

Complete ABAP®

- ▶ An everyday reference for ABAP programmers of all levels
- ▶ Learn language elements, syntax, concepts, and more
- ▶ Explore modularization, modifications, and enhancements

Kiran Bandari



Rheinwerk
Publishing



SAP PRESS is a joint initiative of SAP and Rheinwerk Publishing. The know-how offered by SAP specialists combined with the expertise of Rheinwerk Publishing offers the reader expert books in the field. SAP PRESS features first-hand information and expert advice, and provides useful skills for professional decision-making.

SAP PRESS offers a variety of books on technical and business-related topics for the SAP user. For further information, please visit our website: <http://www.sap-press.com>.

James Wood, Joseph Rupert
Object-Oriented Programming with ABAP Objects (2nd edition)
2016, 470 pages, hardcover and e-book
www.sap-press.com/3597

Paul Hardy
ABAP to the Future (2nd edition)
2016, 801 pages, hardcover and e-book
www.sap-press.com/4161

Kerem Koseoglu
Design Patterns in ABAP Objects
2017, 387 pages, hardcover and e-book
www.sap-press.com/4277

Gahm, Schneider, Swanepoel, Westenberger
ABAP Development for SAP HANA (2nd edition)
2016, 641 pages, hardcover and e-book
www.sap-press.com/3973

Kiran Bandari

Complete ABAP®



Bonn • Boston

Dear Reader,

Flexing your linguistic muscle can be both a rewarding and nerve-wracking experience. When traveling in Spain, I may have, for example, accidentally asked for no octopus (pulpo) in my orange juice instead of no pulp (pulpa). The truth of the matter is, learning a new language is no easy task—programming not excluded.

Whether you're an ABAP newbie or need a refresher on key concepts, this book offers a 360-degree view of everything you need to know about ABAP. From introductions to ABAP keywords to techniques and best practices every good ABAP developer should know, this heavyweight guide is your key to mastering the ABAP language. And since the best way to learn a language is via the help of a native speaker, you'll be in good company with the expertise of author Kiran Bandari.

What did you think about *Complete ABAP*? Your comments and suggestions are the most useful tools to help us make our books the best they can be. Please feel free to contact me and share any praise or criticism you may have.

Thank you for purchasing a book from SAP PRESS!

Sarah Frazier

Editor, SAP PRESS

Rheinwerk Publishing
Boston, MA

editorial@rheinwerk-publishing.com

<http://www.sap-press.com>

Notes on Usage

This e-book is **protected by copyright**. By purchasing this e-book, you have agreed to accept and adhere to the copyrights. You are entitled to use this e-book for personal purposes. You may print and copy it, too, but also only for personal use. Sharing an electronic or printed copy with others, however, is not permitted, neither as a whole nor in parts. Of course, making them available on the Internet or in a company network is illegal as well.

For detailed and legally binding usage conditions, please refer to the section [Legal Notes](#).

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy:

Imprint

This e-book is a publication many contributed to, specifically:

Editor Sarah Frazier

Acquisitions Editor Kelly Grace Weaver

Copyeditor Melinda Rankin

Cover Design Graham Geary

Photo Credit Shutterstock.com/167078399/© Africa Studio

Production E-Book Kelly O'Callaghan

Typesetting E-Book SatzPro, Krefeld (Germany)

We hope that you liked this e-book. Please share your feedback with us and read the [Service Pages](#) to find out how to contact us.

The Library of Congress has cataloged the printed edition as follows:

Names: Bandari, Kiran, author.

Title: Complete ABAP / Kiran Bandari.

Description: 1st edition. | Bonn ; Boston : Rheinwerk Publishing, 2016. |

Includes index.

Identifiers: LCCN 2016016863 | ISBN 9781493212729 (print : alk. paper) | ISBN

9781493212743 (print and ebook : alk. paper) | ISBN 9781493212736 (ebook)

Subjects: LCSH: ABAP/4 (Computer program language) | SAP ERP.

Classification: LCC QA76.73.A12 B36 2016 | DDC 005.1/17--dc23 LC record available at <https://lccn.loc.gov/2016016863>

ISBN 978-1-4932-1272-9 (print)

ISBN 978-1-4932-1273-6 (e-book)

ISBN 978-1-4932-1274-3 (print and e-book)

© 2017 by Rheinwerk Publishing, Inc., Boston (MA)

1st edition 2017

Contents

Acknowledgments	21
Preface	23

1 Introduction to ERP and SAP 29

1.1	Historical Overview	29
1.2	Understanding an ERP System	32
1.2.1	What Is ERP?	32
1.2.2	ERP vs. Non-ERP Systems	33
1.2.3	Advantages of an ERP System	35
1.3	Introduction to SAP	36
1.3.1	Modules in SAP	36
1.3.2	Types of Users	37
1.3.3	Role of an ABAP Consultant	38
1.3.4	Changing and Adapting the Data Structure	40
1.4	ABAP Overview	43
1.4.1	Types of Applications	43
1.4.2	RICEF Overview	43
1.5	System Requirements	48
1.6	Summary	48

2 Architecture of an SAP System 49

2.1	Introduction to the Three-Tier Architecture	49
2.2	SAP Implementation Overview	52
2.2.1	SAP GUI: Presentation Layer	52
2.2.2	Application Servers and Message Servers: Application Layer	54
2.2.3	Database Server/RDBMS: Database Layer	61
2.3	Data Structures	64
2.3.1	Client Overview	64
2.3.2	Client-Specific and Cross-Client Data	65
2.3.3	Repository	68
2.3.4	Packages	68
2.3.5	Transport Organizer	70
2.4	Summary	75

3 Introduction to the ABAP Environment 77

- 3.1 SAP Environment 78
 - 3.1.1 ABAP Programming Environment 78
 - 3.1.2 Logging On to the SAP Environment 78
 - 3.1.3 Elements of the SAP Screen 79
 - 3.1.4 Transaction Codes 82
 - 3.1.5 Navigating and Opening Transactions 83
- 3.2 ABAP Workbench Overview 87
 - 3.2.1 ABAP Editor 88
 - 3.2.2 Function Builder 90
 - 3.2.3 Class Builder 91
 - 3.2.4 Screen Painter 92
 - 3.2.5 Menu Painter 95
 - 3.2.6 ABAP Data Dictionary 96
 - 3.2.7 Object Navigator 98
- 3.3 Eclipse IDE Overview 99
- 3.4 Summary 105

4 ABAP Programming Concepts 107

- 4.1 General Program Structure 108
 - 4.1.1 Global Declarations 108
 - 4.1.2 Procedural 109
- 4.2 ABAP Syntax 110
 - 4.2.1 Basic Syntax Rules 110
 - 4.2.2 Chained Statements 111
 - 4.2.3 Comment Lines 112
- 4.3 ABAP Keywords 113
- 4.4 Introduction to the TYPE Concept 114
 - 4.4.1 Data Types 115
 - 4.4.2 Data Elements 130
 - 4.4.3 Domains 134
 - 4.4.4 Data Objects 137
- 4.5 ABAP Statements 141
- 4.6 Creating Your First ABAP Program 143
- 4.7 Summary 149

5 Structures and Internal Tables 151

5.1	Defining Structures	152
5.1.1	When to Define Structures	154
5.1.2	Local Structures	155
5.1.3	Global Structures	158
5.1.4	Working with Structures	162
5.1.5	Use Cases	163
5.2	Internal Tables	164
5.2.1	Defining Internal Tables	164
5.2.2	Types of Internal Tables	167
5.2.3	Table Keys	171
5.2.4	Working with Internal Tables	175
5.2.5	Control Break Statements	182
5.3	Introduction to Open SQL Statements	187
5.3.1	Database Overview	189
5.3.2	Selecting Data from Database Tables	198
5.3.3	Selecting Data from Multiple Tables	200
5.4	Processing Data from Database via Internal Tables and Structures	203
5.5	Introduction to the Debugger	205
5.6	Practice	209
5.7	Summary	209

6 User Interaction 211

6.1	Selection Screen Overview	212
6.1.1	PARAMETERS	214
6.1.2	SELECT-OPTIONS	221
6.1.3	SELECTION-SCREEN	229
6.1.4	Selection Texts	230
6.2	Messages	231
6.2.1	Types of Messages	231
6.2.2	Messages Using Text Symbols	233
6.2.3	Messages Using Message Classes	235
6.2.4	Dynamic Messages	237
6.2.5	Translation	238
6.3	Summary	239

7 Modularization Techniques 241

- 7.1 Modularization Overview 242
- 7.2 Program Structure 245
 - 7.2.1 Processing Blocks 246
 - 7.2.2 Event Blocks 258
 - 7.2.3 Dialog Modules 260
 - 7.2.4 Procedures 261
- 7.3 Events 261
 - 7.3.1 Program Constructor Events 262
 - 7.3.2 Reporting Events 262
 - 7.3.3 Selection Screen Events 269
 - 7.3.4 List Events 270
 - 7.3.5 Screen Events 271
- 7.4 Procedures 272
 - 7.4.1 Subroutines 274
 - 7.4.2 Function Modules 284
 - 7.4.3 Methods 293
- 7.5 Inline Declarations 300
 - 7.5.1 Assigning Values to Data Objects 301
 - 7.5.2 Using Inline Declarations with Table Work Areas 301
 - 7.5.3 Avoiding Helper Variables 302
 - 7.5.4 Declaring Actual Parameters 302
- 7.6 Summary 303

8 Object-Oriented ABAP 305

- 8.1 Introduction to Object-Oriented Programming 305
 - 8.1.1 Classes 308
 - 8.1.2 Methods 313
 - 8.1.3 Instance and Static Components 320
 - 8.1.4 Events 322
- 8.2 Encapsulation 325
 - 8.2.1 Component Visibility 326
 - 8.2.2 Friends 329
 - 8.2.3 Implementation Hiding 332
- 8.3 Inheritance 335
 - 8.3.1 Inheriting Components 336
 - 8.3.2 Abstract Classes and Methods 341
 - 8.3.3 Final Classes and Methods 344

8.3.4	Composition	345
8.3.5	Refactoring Assistant	346
8.4	Polymorphism	347
8.4.1	Static and Dynamic Types	348
8.4.2	Casting	350
8.4.3	Dynamic Binding with the CALL Method	354
8.4.4	Interfaces	357
8.5	Working with XML	362
8.5.1	XML Overview	362
8.5.2	XML Processing Concepts	364
8.6	Summary	366

9 Exception Handling 369

9.1	Exceptions Overview	369
9.2	Procedural Exception Handling	370
9.2.1	Maintaining Exceptions Using Function Modules	370
9.2.2	Maintaining Exceptions Using Methods	373
9.2.3	Maintaining Exceptions for Local Classes	374
9.3	Class-Based Exception Handling	375
9.3.1	Raising Exceptions	376
9.3.2	Catchable and Non-Catchable Exceptions	378
9.3.3	Defining Exception Classes Globally	383
9.3.4	Defining Exception Classes Locally	386
9.4	Messages in Exception Classes	387
9.4.1	Using the Online Text Repository	387
9.4.2	Using Messages from a Message Class	390
9.4.3	Using the MESSAGE Addition to Raise an Exception	393
9.5	Summary	393

10 ABAP Data Dictionary 395

10.1	Database Tables	396
10.1.1	Creating a Database Table	399
10.1.2	Indexes	409
10.1.3	Table Maintenance Generator	414
10.1.4	Foreign Keys	418
10.1.5	Include Structure	422
10.1.6	Append Structure	424
10.2	Views	426
10.2.1	Database Views	427

10.2.2	Projection Views	429
10.2.3	Maintenance Views	431
10.2.4	Help Views	433
10.2.5	ABAP CDS Views	434
10.3	Data Types	438
10.3.1	Data Elements	439
10.3.2	Structures	443
10.3.3	Table Types	446
10.4	Type Groups	450
10.5	Domains	451
10.6	Search Helps	455
10.6.1	Elementary Search Helps	457
10.6.2	Collective Search Helps	461
10.6.3	Assigning a Search Help	463
10.6.4	Search Help Exits	464
10.7	Lock Objects	465
10.8	Summary	469

11 Persistent Data 471

11.1	Working with Data in Databases	472
11.1.1	Open SQL	474
11.1.2	Logical Unit of Work	485
11.2	ABAP Object Services	492
11.2.1	Persistence Service Overview	493
11.2.2	Building Persistent Classes	493
11.2.3	Working with Persistent Objects	498
11.3	File Interfaces	498
11.3.1	Working with Files in the Application Server	499
11.3.2	Working with Files in the Presentation Layer	501
11.4	Data Clusters	503
11.4.1	Exporting Data Clusters to Databases	505
11.4.2	Importing Data Clusters	506
11.5	Security Concepts	506
11.6	Summary	508

12 Dialog Programming 509

12.1	Screen Events	510
12.2	Screen Elements and Flow Logic	513
12.2.1	Components of a Dialog Program	515

12.2.2	Screens	520
12.2.3	Screen Elements	525
12.3	Basic Screen Elements	529
12.3.1	Text Fields	530
12.3.2	Checkboxes and Radio Buttons	531
12.3.3	Push Button	533
12.4	Input/Output Fields	534
12.5	List Box	536
12.6	Table Controls	537
12.6.1	Create a Table Control without a Wizard	538
12.6.2	Create a Table Control with a Wizard	542
12.7	Tabstrip Controls	544
12.8	Subscreens	545
12.9	Working with Screens	547
12.9.1	Screen Flow Logic	548
12.9.2	GUI Status	550
12.9.3	GUI Title	552
12.9.4	Modifying Screen Fields Dynamically	553
12.9.5	Field Help and Input Help	555
12.9.6	Screen Sequence	556
12.9.7	Assigning Transaction Codes	558
12.10	Control Framework	560
12.10.1	Using Container Controls	561
12.10.2	Implementing Custom Controls	562
12.11	Practice	564
12.11.1	Application Flow	566
12.11.2	Delete Functionality	567
12.11.3	Validations and Autofills	567
12.12	Summary	568

13 List Screens 569

13.1	Program Types	570
13.1.1	Executable Programs	571
13.1.2	Module Pool Programs	572
13.1.3	Function Groups	572
13.1.4	Class Pools	573
13.1.5	Interface Pools	573
13.1.6	Subroutine Pools	573
13.1.7	Type Pools	574
13.1.8	Include Programs	574

13.2	Program Execution	574
13.2.1	Executable Program Flow	575
13.2.2	Module Pool Program Flow	576
13.2.3	Calling Programs Internally	577
13.3	Memory Organization	578
13.4	List Events	582
13.4.1	TOP-OF-PAGE	583
13.4.2	END-OF-PAGE	585
13.4.3	AT LINE-SELECTION	586
13.4.4	AT USER-COMMAND	586
13.5	Basic Lists and Detail Lists	588
13.6	Classical Reports	592
13.7	Interactive Reports	592
13.7.1	HIDE	592
13.7.2	READ LINE	596
13.7.3	GET CURSOR	597
13.7.4	DESCRIBE LIST	597
13.8	Practice	598
13.9	Summary	599

14 Selection Screens 601

14.1	Defining Selection Screens	602
14.2	Selection Screen Events	604
14.3	Input Validations	606
14.4	Selection Screen Variants	608
14.4.1	Creating a Variant	609
14.4.2	Variant Attributes	613
14.4.3	Table Variables from Table TVARVC	614
14.4.4	Dynamic Date Calculation	616
14.4.5	Dynamic Time Calculation	617
14.4.6	User-Specific Variables	618
14.5	Executing Programs in the Background	619
14.6	Displaying and Hiding Screen Elements Dynamically	621
14.7	Calling Programs via Selection Screens	623
14.8	Summary	623

15 ALV Reports 625

15.1	Standard ALV Reports Using the Reuse Library	626
15.1.1	List and Grid Display: Simple Reports	627

15.1.2	Block Display	636
15.1.3	Hierarchical Sequential Display	640
15.2	Interactive Reports	644
15.2.1	Loading a Custom SAP GUI Status	645
15.2.2	Reacting to User Actions	649
15.2.3	Printing TOP-OF-PAGE	650
15.3	ALV Reports Using the Control Framework	650
15.4	ALV Object Model	653
15.4.1	Table Display	654
15.4.2	Hierarchical Display	655
15.4.3	Tree Object Model	659
15.5	Summary	661

16 Dynamic Programming 663

16.1	Field Symbols	665
16.1.1	Using Field Symbols to Make Programs Dynamic	666
16.1.2	Defining Field Symbols	673
16.1.3	Assigning a Data Object	674
16.1.4	Checking if a Field Symbol is Assigned	677
16.1.5	Unassigning a Field Symbol	678
16.1.6	Casting	678
16.2	Data References	679
16.2.1	Defining Reference Variables	680
16.2.2	Getting Data References	681
16.2.3	Anonymous Data Objects	682
16.2.4	Assignment between Reference Variables	684
16.3	Runtime Type Services	685
16.3.1	Runtime Type Information	686
16.3.2	Runtime Type Creation	687
16.4	Dynamic Token Specification	691
16.5	Dynamic Procedure Calls	693
16.6	Dynamic Program Generation	695
16.7	Summary	697

17 Debugging 699

17.1	Classic Debugger	700
17.1.1	Activating and Using the Classic Debugger	701
17.1.2	Field View	705
17.1.3	Table View	705

17.1.4	Breakpoints View	706
17.1.5	Watchpoints View	707
17.1.6	Calls View	708
17.1.7	Overview View	708
17.1.8	Settings View	708
17.1.9	Additional Features	710
17.2	New Debugger	714
17.2.1	UI and Tools	714
17.2.2	Layout and Sessions	717
17.3	AMDP Debugger	718
17.4	Using the Debugger to Troubleshoot	719
17.5	Using the Debugger as a Learning Tool	721
17.6	Summary	722

18 Forms 723

18.1	SAPscripts	725
18.1.1	Overview and Layout	725
18.1.2	Creating the Form Layout	729
18.1.3	Maintaining Window Details	736
18.1.4	Processing Forms with Function Modules	741
18.2	Smart Forms	749
18.2.1	Overview and Layout	750
18.2.2	Maintaining the Global Settings	753
18.2.3	Maintaining Elements	756
18.2.4	Driver Program	774
18.3	SAP Interactive Forms by Adobe	777
18.3.1	Form Interface	778
18.3.2	Form Context and Layout	785
18.3.3	Driver Program	797
18.3.4	Downloading the Form as a PDF	799
18.4	Summary	800

19 Interfaces 803

19.1	Batch Data Communication	804
19.1.1	Direct Input	806
19.1.2	Batch Input	807
19.2	Business Application Programming Interface	819
19.2.1	Business Object Types and Business Components	819
19.2.2	BAPI Development via BAPI Explorer	820

19.2.3	Standardized BAPIs	823
19.2.4	Standardized Parameters	824
19.2.5	Implementing BAPIs	826
19.3	EDI/ALE/IDocs	835
19.3.1	Electronic Data Interchange	836
19.3.2	Application Link Enabling	841
19.3.3	Intermediate Documents	844
19.3.4	System Configurations	856
19.3.5	Inbound/Outbound Programs	863
19.4	Legacy System Migration Workbench	867
19.4.1	Getting Started	868
19.4.2	Migration Process Steps	869
19.5	Web Services	880
19.5.1	Creating a Web Service	884
19.5.2	Consuming Web Services	888
19.6	OData Services	895
19.6.1	Data Model Definition	897
19.6.2	Service Maintenance	901
19.6.3	Service Implementation	903
19.6.4	READ	905
19.7	XSL Transformations	908
19.7.1	Serialization	909
19.7.2	Deserialization	910
19.8	XML and JSON Data Representation	911
19.9	WebSockets (ABAP Channels and Messages)	913
19.9.1	Creating an ABAP Messaging Channel	914
19.9.2	Creating a Producer Program	916
19.9.3	Creating a Consumer Program	917
19.10	Summary	920

20 Modifications and Enhancements 921

20.1	Customization Overview	921
20.2	Modification Overview	923
20.3	Using Modification Assistant	924
20.3.1	Modifications to Programs	925
20.3.2	Modifications to Class Builder	927
20.3.3	Modifications to Screen Painter	928
20.3.4	Modifications to Menu Painter	929
20.3.5	Modifications to ABAP Data Dictionary	929

20.3.6	Modifications to Function Modules	930
20.3.7	Resetting to Original	931
20.4	Using Modification Browser	932
20.5	Enhancements Overview	933
20.6	User Exits	935
20.7	Customer Exits	936
20.7.1	Create a Customer Exit	939
20.7.2	Function Module Exits	942
20.7.3	Screen Exits	942
20.7.4	Menu Exits	944
20.8	BADIs	946
20.8.1	Overview	946
20.8.2	Defining a BAdI	948
20.8.3	Implementing a BAdI	955
20.8.4	Implementing a Fallback Class	958
20.8.5	Calling a BAdI	959
20.9	Enhancement Points	960
20.9.1	Explicit Enhancements	961
20.9.2	Implicit Enhancements	963
20.10	Business Transaction Events	966
20.10.1	Implementing a BTE	967
20.10.2	Testing a Custom Function Module	971
20.11	Summary	972

21 Test and Analysis Tools 973

21.1	Overview of Tools	974
21.2	ABAP Unit	976
21.2.1	Eliminating Dependencies	977
21.2.2	Implementing Mock Objects	979
21.2.3	Writing and Implementing Unit Tests	980
21.3	Code Inspector	987
21.4	Selectivity Analysis	990
21.5	Process Analysis	992
21.6	Memory Inspector	995
21.6.1	Creating Memory Snapshots	995
21.6.2	Comparing Memory Snapshots	996
21.7	Table Call Statistics	997
21.8	Performance Trace	1000
21.8.1	Activating and Filtering a Performance Trace	1001
21.8.2	SQL Trace	1004

21.8.3	RFC Trace	1006
21.8.4	Enqueue Trace	1007
21.8.5	Buffer Trace	1008
21.9	ABAP Trace/Runtime Analysis	1009
21.9.1	Running ABAP Trace	1009
21.9.2	Analyzing the Results	1011
21.10	Single-Transaction Analysis	1014
21.11	Dump Analysis	1017
21.12	Summary	1019
The Author		1021
Index		1023
Service Pages		I
Legal Notes		II

Acknowledgments

Writing an acknowledgement page is like preparing a wedding invitation list; you just have too many people to include.

This book took over a year to write and includes all the learnings from my decade long career in ABAP. First, I'd like to thank Ram Mohan and Raj Narayan for introducing me to this wonderful world of ABAP. A special thanks to my colleagues, who gave their perspective on the depth of discussion for each topic of this book, and my students, who helped me with their feedback to keep the discussions simple enough to comprehend.

I'd like to express my gratitude to Kelly Grace Weaver for giving me the opportunity to author this book for SAP PRESS and brainstorming with me to get the topics for this book finalized.

Special thanks to my editor, Sarah Frazier, for all her support, patience, and encouragement to see this project through. Her input, corrections, and attention to detail have helped me tremendously to expand the coverage of each topic.

Writing a book this big that expands over a thousand pages takes a toll on your personal life. I am grateful to my family for their blessings, especially my late father without whose teachings I wouldn't be what I am today.

I'd especially like to thank my dear wife, Dr. Reena Sinha, who ensured I stayed fit and healthy despite long hours and late nights spent to meet the deadlines. This book would not have been possible without her constant moral support and motivation. She literally took care of everything else while I focused on writing this book and forgave me (with little tantrums, of course!) each time I failed to keep my promises of spending our free time together.

Acknowledgments

There were many difficult times during this project which tested my perseverance. Finally, I want to add a quote that inspired me to push myself each time I felt like giving up:

A hero is no braver than an ordinary man, but he is brave five minutes longer.

– Ralph Waldo Emerson

Kiran Bandari

Preface

Advanced Business Application Programming, or ABAP, is a programming language used to develop applications in an SAP environment. ABAP contains numerous language elements that are stored in different libraries, and ABAP programs are developed in development environments as part of SAP NetWeaver AS ABAP. All application development objects are created within the SAP NetWeaver AS ABAP environment.

ABAP was developed to meet the business processing needs of ERP systems, and has evolved over time to meet other enterprise requirements as well. More information on the evolution of SAP ERP and ABAP will be discussed in [Chapter 1](#) and [Chapter 2](#).

With the advent of SAP NetWeaver AS ABAP 7.5 and higher, new innovations have emerged that make the lives of developers that much easier. In this introduction, we'll discuss the primary objective of this book and how it addresses both standard practices and new innovations in ABAP. We will also discuss who this book is targeted at as well as its structure and methodology.

The Objective of this Book

The objective of this book is to offer complete coverage on the most important elements of ABAP, explaining both the concepts at hand and then providing examples of their practical application.

The book is divided into chapters based on the major concepts in ABAP programming, and are ordered according to how one would learn ABAP. For example, we explain the elements of the ABAP Data Dictionary that are relevant in their respective chapters: data elements are discussed when explaining data types, table types are explored when discussing procedures, and so on. The idea is to provide a book that focuses on your learning process.

Target Audience

This book is an everyday reference for ABAP developers of all program levels. While the beginning chapters tend to cater to a more beginner audiences, the most advanced chapters provide refreshers and insights that even seasoned ABAP developers will be interested in reviewing. Taking a good look at the table of contents will allow you to identify which topic areas you may need more insight into. This book also provides a detailed index if you have a particular subject in mind you'd like to jump to.

Structure of this Book

This book starts off with basic concepts and then transitions into more advanced topics. Understanding the prerequisites discussed in the beginning chapters is important for grasping the more advanced topics explained later on in the book.

The following overview offers details on individual chapters and the overall structure of the book:

- ▶ **Chapter 1, Introduction to ERP and SAP**, provides an overview of enterprise resource planning (ERP) systems and the advantages this type of software offers. We then look at the different SAP modules, both functional and technical, used in SAP. From there, we discuss how you can customize your SAP system per your business requirements using custom development, modifications, and enhancements. Finally, we begin shifting the focus to the main topic of this book, ABAP, and the system requirements necessary for carrying out the different examples discussed throughout this book.
- ▶ **Chapter 2, The Architecture of an SAP System**, offers a “nuts-and-bolts” look at the SAP system architecture. The topics in this chapter will help you clearly understand the type of environment your ABAP programs will be run in. We'll look at the different architectural layers: presentation, application, and database. Finally, we'll close out our discussion with an overview of data structures and the use of clients and packages.
- ▶ **Chapter 3, Introduction to the ABAP Environment**, discusses two different integrated development environments (IDEs) that can be used when working

with ABAP: SAP GUI and Eclipse. This chapter begins by looking at the different elements of an SAP screen and how to use transaction codes to navigate between these screens. We then dive into the tools found in the ABAP Workbench: the ABAP Editor, ABAP Data Dictionary, Screen Painter, Menu Painter, and Object Navigator. We round out our discussion with a look at Eclipse and its setup.

- ▶ **Chapter 4, ABAP Programming Concepts**, introduces the basics of ABAP: key concepts, general program structures, syntax, keywords, and more. In addition, we'll also look at using data types, data elements, data objects, and domains. We end the chapter with steps to create your first ABAP program.
- ▶ **Chapter 5, Structures and Internal Tables**, explores complex data types and lays the foundation for working with the data stored in a typical database. We begin the chapter by looking at local and global structures and how to define them. We then move on to working with internal tables, before jumping into an introduction to Open SQL. The chapter ends with a practice example that highlights the concepts taught throughout the chapter.
- ▶ **Chapter 6, User Interaction**, takes us beyond the basics and introduces user interaction within our ABAP code. The chapter begins with instructions for using a special kind of dynpro: *selection screens*. Implementing selection screens enables users to provide selection criteria for a report program, while also providing an interface for data transfer between two executable programs. We then move on to using messages to communicate with users from ABAP programs.
- ▶ **Chapter 7, Modularization Techniques**, teaches how to implement modularization into your code to provide, among other things, better readability and organization in your programs. Specifically, this chapter looks at the structure of an ABAP program, including processing blocks, events, and procedures. The chapter ends by discussing a feature introduced in SAP NetWeaver 7.4: inline declarations.
- ▶ **Chapter 8, Object-Oriented ABAP**, takes a walk on the object-oriented side of ABAP. As we will discuss, there are two programming approaches to ABAP: procedural and object-oriented (OOP). While we discuss both techniques throughout the book, this chapter focuses on the general concepts of OOP. This includes key features such as encapsulation, inheritance, polymorphism, and more.

- ▶ **Chapter 9, Exception Handling**, explains how to implement messages to users when an error has occurred in the system (for example, inputting an incorrect field value). The chapter will look at both procedural and OOP (class-based) exception handling approaches.
- ▶ **Chapter 10, ABAP Data Dictionary**, reviews concepts previously mentioned about the ABAP Data Dictionary thus far and offers a complete discussion on topics not yet covered. This includes more detailed information on database tables, views, data types, domains, type groups, search helps, and lock objects.
- ▶ **Chapter 11, Persistent Data**, explains how persistent data is used in ABAP programs. ABAP programs generally work with data that is read from a persistent storage source, such as a database, meaning that data exists even when the program is not reading it. In this chapter, we'll look at how to work with data in a database, persistent object services, file interfaces, and data clusters. We'll end the chapter with a look at key security concepts to keep in mind when dealing with persistent data.
- ▶ **Chapter 12, Dialog Programming**, provides steps for creating screens, flow logic, menu bars, and more for an application. The end of the chapter offers a practice example where you can implement what you learned into a real application.
- ▶ **Chapter 13, List Screens**, explains one of the most basic ways to develop reports in ABAP: list screens. There are three types of screens in ABAP: selection screens (discussed previously in [Chapter 6](#) and to be discussed in [Chapter 14](#)), general dialog screens, and list screens. This chapter provide steps for creating list screens for developing reports using program types, program execution, list events, and more. We again end this chapter with a series of exercises.
- ▶ **Chapter 14, Selection Screens**, offers more details on selection screens and how you can use them to display information to application users. Specifically, we'll dive into the events, input validations, variants, and batch jobs used with selection screens. We'll also provide tips for displaying and hiding screen elements dynamically and steps for calling a program.
- ▶ **Chapter 15, ALV Reports**, discusses how to develop reports using procedural and OOP ALV function modules. ALV function modules are another way to create reports in ABAP.
- ▶ **Chapter 16, Dynamic Programming**, explains how to enable dynamic programs in ABAP. Dynamic programs return results based on user inputs. In this

chapter, we'll look specifically at using field symbols, data references, specifications, procedure calls, and more, dynamically.

- ▶ **Chapter 17, Debugging**, takes a close look at the two primary debuggers used in ABAP: the Classic Debugger and the New Debugger. In addition to learning how to implement debugging via these two debuggers, we'll also take a brief look at the new AMDP Debugger for ABAP Managed Database Procedures (AMDP). We'll end the chapter with a discussion on using debugging tools for troubleshooting and for learning purposes.
- ▶ **Chapter 18, Forms**, explores the three main types of form technology that can be used to create forms in ABAP: SAPscripts, smart forms, and SAP Interactive Forms by Adobe. Each section describes the step-by-step process for creating forms. We then compare and contrast the advantages and disadvantages of these three technologies.
- ▶ **Chapter 19, Interfaces**, focuses on interface development. Each section is devoted to a different type of interface, including LSWM, web and OData services, ALE/EDI/IDocs, XSL transformations, and more. We begin by explaining the concept behind each interface technology, then dive into the steps for working with them.
- ▶ **Chapter 20, Modifications and Enhancements**, provides information on how to modify and enhance repository objects. We begin with an overview of the Modification Assistant and Modification Browser, and how they can be used to modify objects within the ABAP Workbench. Moving on to enhancements, we'll look at how to best implement user exits, customer exits, BAdIs, and more.
- ▶ **Chapter 21, Test and Analysis Tools**, concludes our book with a look at the tools for testing and analyzing your ABAP code. As any good developer will tell you, it is important to test your code to ensure that it is error-free. Therefore, we'll look at a number of tools to guarantee this, including ABAP Unit for creating unit tests, Code Inspector for analyzing program code, Memory Inspector for creating memory snapshots of the memory consumed by an ABAP program, and many more.

It is our hope that this book serves as a roadmap for your journey as an ABAP developer. Now, roll up your sleeves and get ready to work!

We'll begin our journey with a basic introduction to ERP systems and SAP. From there, we'll begin to paint a picture of what it means to be an ABAP consultant and begin looking at an overview of ABAP and system requirements.

1 Introduction to ERP and SAP

This book is targeted for people who wish to learn the ABAP programming language inside and out, irrespective of their prior programming exposure. Our aim is to start from the very basics of programming in the initial chapters and introduce you to advanced concepts in the later chapters. We are not assuming you to have any prior programming experience, so whether you are an ABAP consultant or a seasoned SAP functional consultant, or even if you are completely new to SAP, you should find it easy to follow this book.

This chapter begins by introducing the basic concepts of ERP and SAP. [Section 1.1](#) discusses how the SAP system has evolved over the years. Then, in [Section 1.2](#), we provide a deeper understanding of ERP systems in general, before diving into SAP-specific information in [Section 1.3](#), including the different modules within SAP. [Section 1.4](#) introduces the focus of this book, ABAP, before listing the system requirements necessary to work with ABAP in [Section 1.5](#).

1.1 Historical Overview

In 1972, SAP introduced its first system, *R/1*. *R* represents *real-time data processing*. This system was used for financial accounting and did not include the presentation, application, or database layers that you would find in the SAP system today. In the late 1970s, SAP introduced the *R/2* system, which had two layers: the presentation layer and the application plus database layer. These systems followed a two-tier architecture.

As technology evolved, SAP moved towards a three-tier architecture with the introduction of SAP R/3 in the 1990s. The SAP R/3 system comprised three layers: the presentation layer, application layer, and database layer. With the release of SAP R/3, SAP named its releases with versions such as R/3 1.0, R/3 2.0, R/3 3.1i, R/3 4.0B (until 2001, with R/3 4.6D), with an underlying component called SAP Basis.

SAP R/3 4.7 was the first release that came with a technology platform called *SAP Web Application Server*, which included Business Server Pages (BSP) applications and a J2EE engine. Since then, SAP has added technology platform naming to its product line. The SAP Web Application Server is a natural evolution to support web infrastructure (for the World Wide Web), so it performs all the duties of an application server, similar to the previous releases, and also acts as a web server to support web infrastructure. BSP is the server pages technology developed by SAP, similar to JavaServer Pages from Sun Microsystems (JSP) and Active Server Pages by Microsoft (ASP).

SAP then started adding the platform version to the product naming. This led to some confusion when, for example, you could have your SAP R/3 4.x system (denoting the version of your SAP R/3 system) on SAP Web Application Server 6.10 (denoting the version of your platform).

With the introduction of platforms (the first being SAP Web Application Server), SAP systems began using specific components that represented the platform, such as Internet Communication Manager (ICM), SAP Gateway, and so on, and other components related to business functions that showed the functionality of the system. SAP then introduced enterprise editions along with SAP Web Application Server 6.20, such as SAP R/3 Enterprise 4.70. The enterprise editions provided new functionality to split components based on business functions and also provided the ability to use extensions to extend the functionality of these core business functions.

SAP included the release year with the product names, such as mySAP ERP 2004. mySAP ERP 2004 signaled a complete architectural change in SAP R/3 Enterprise, which was replaced with SAP Enterprise Central Component (SAP ECC). SAP then introduced SAP NetWeaver 2004, which also contained SAP Web Application Server 6.40.

To add to the confusion, SAP started referring to mySAP ERP 2004 as SAP ECC 5.0, which puzzled customers when trying to refer to SAP products. For example,

at that moment, both mySAP ERP2004 and SAP ECC 5.0 could refer to the same system, and SAP NetWeaver 2004 and SAP Web Application Server 6.40 could refer to the same platform. The mySAP 2004 and SAP ECC 5.0 systems were also found under the umbrella term SAP ERP 5.0.

In 2005, mySAP ERP 2005 came out, which was referenced as SAP ECC 6.0, resided on the SAP NetWeaver 2004 (SAP Web Application Server 7.00) platform, and was also called SAP ERP 6.0.

Table 1.1 captures the evolution of SAP systems over the years.

Year	Version
1972	SAP R/1 (<i>R</i> represents real-time; 1 represents one layer)
Late 1970s	SAP R/2 (<i>R</i> represents real-time; 2 represents two layers)
1992	SAP R/3 (<i>R</i> represents real-time; 3 represents three layers)
2001	Introduction of SAP Web Application Server (technology platform)
2002	SAP R/3 Enterprise 4.70 (SAP Web Application Server 6.20)
2003	mySAP ERP 2003 (SAP Web Application Server 6.30)
2004	mySAP ERP 2004 ECC 5.0 ERP 5.0 (SAP Web Application Server 6.40; SAP NetWeaver 2004)
2005	mySAP ERP 2005 ECC 6.0 ERP 6.0 (SAP Web Application Server 7.00; NetWeaver 2004s/7)
2006	SAP NetWeaver 7.1
2011	SAP NetWeaver 7.3
2013	SAP NetWeaver 7.4
2015	SAP NetWeaver 7.5

Table 1.1 Evolution of SAP Systems

Enhancement packages provided additional functionality to customers without touching the core components. Customers activate the new business functions that they want to use via the Switch Framework. The enhancement packages contain new functions for both generic and industry-specific processes.

These enhancement packages are delivered to customers at regular intervals (usually every two years), bundled with all the new business software functionalities

that SAP has delivered in the forms of service packs (SP) since the last enhancement pack.

For example, SAP ERP 6.0 EHP 7 offers over one hundred new functions compared to SAP ERP 6.0 EHP 6 and is the first enhancement package (EHP) to be optimized for both the SAP HANA database and traditional databases.

Table 1.2 lists the enhancement packages released for the SAP ERP 6.0 system.

Year	Enhancement Package
2006	EHP 1
2007	EHP 2
2007	EHP 3
2008	EHP 4
2010	EHP 5
2011	EHP 6
2014	EHP 7
2016	EHP 8

Table 1.2 Overview of Enhancement Packages Released for SAP ERP 6.0

Now that you have a better understanding of the evolution of SAP software and naming conventions, in the next section we'll dive deeper into ERP system software.

1.2 Understanding an ERP System

SAP provides enterprise resource planning (ERP) software, and ABAP is the programming language used to develop applications in SAP systems. Understanding how an SAP ERP system works is vital to understanding ABAP programming. In this section, we will look at what ERP software encompasses, ERP vs. non-ERP systems, and the advantages of using ERP software.

1.2.1 What Is ERP?

ERP business management software integrates various functions of a business. An ERP system typically implements a common database in which all the data from

various business functions of an organization are stored. This allows applications to access the data seamlessly across business processes. To understand ERP, let's look at how a non-ERP system works.

1.2.2 ERP vs. Non-ERP Systems

In a typical non-ERP system, every department in an organization uses its own database and standalone applications. To better understand this, let's look at a simple example of how a typical business works and the challenges a business faces with a non-ERP setup. This should help you appreciate the advantages an ERP system provides.

Let's assume you're running a business that sells laptop computers. Your organization has various departments to run the business—for example, say that you have the following departments that take care of different business activities:

- ▶ **Sales department**
Captures customer orders
- ▶ **Materials department**
Procures various raw material from different vendors to build the product
- ▶ **Production department**
Produces the product
- ▶ **Logistics department**
Delivers the product to the customer
- ▶ **Finance department**
Manages accounts

These activities—from taking a customer order to delivering the product—comprise a single business transaction within the system. Each business transaction involves multiple departments. If we take the example of a made-to-order product, the business transaction flow will be as follows:

1. The sales department captures the order as per the customer requirement.
2. The materials department procures the required raw materials based on the order information.
3. The production department manufactures or assembles the product from the raw materials.

4. The logistics department delivers the product to the customer address captured by the sales department while creating the order.
5. The accounts department addresses all the costs involved in this business transaction and payments received.

As shown in this example, all departments rely on information from each other to complete a business transaction. However, in a non-ERP setup, the system is not designed for the seamless transfer of data across departments. Instead, each department is connected to its own database, as illustrated in [Figure 1.1](#).

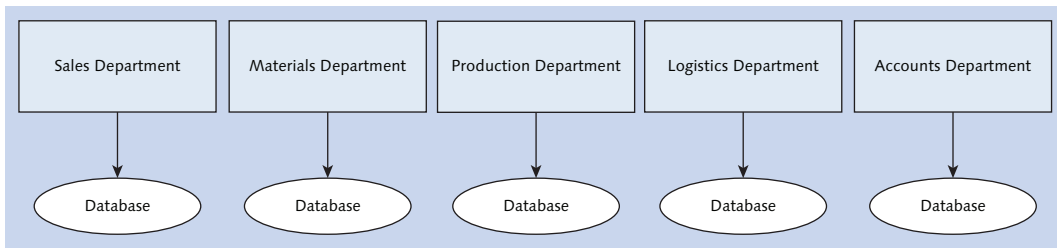


Figure 1.1 Typical Non-ERP System Layout

This setup requires pushing the data manually to other departments before they can act on it. In other words, the data is not exchanged in real time across business functions. In this example, the materials department can't start with procurement until the order details are sent manually by the sales team. This causes delays and hinders automated business processes.

For the purposes of our example, let's say the materials team members update their system to indicate the unavailability of a key raw material, or the logistics team members update their system to indicate a temporary unavailability in a particular location. The sales team can't take advantage of this information while taking the order or before committing to a delivery date for the customer; sales has to check with other departments manually before committing to the date. The software application that the sales team uses to capture the order doesn't automatically validate material availability, because it doesn't have access to data across departments.

[Figure 1.2](#) illustrates a typical ERP system layout. As you can see, because all business functions across the organization access a common database, the information can flow seamlessly.

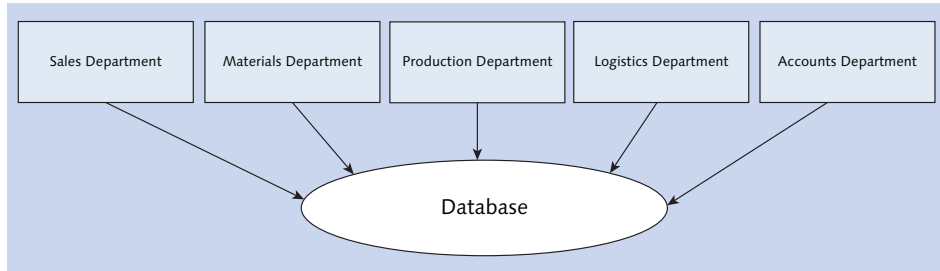


Figure 1.2 Typical ERP System Layout

1.2.3 Advantages of an ERP System

An ERP system allows you to build software applications that can access information across the business functions of an organization, enabling the scope for automation. With an ERP setup, a sales order application can, for example, automatically query the material availability data and the logistics data to provide automatic validation for the delivery date.

An ERP setup can throw an error if a sales person enters a delivery date that can't be met. You can develop an online order-tracking application for customers, because an ERP system can access data across departments to provide an accurate status update. An ERP system provides real-time data access: Data updated by one department is instantly available for others.

A common database allows for developing ERP software with tightly integrated business applications. In an ERP system, you typically have thousands of applications across business functions, seamlessly integrated with one another, as opposed to single, standalone applications for individual business functions. The database consists of hundreds of tables to store data from different business functions, usable by any application.

Of course, each ERP system implements various security measures, such as checking for user authorizations, to prevent any unauthorized access of data. We will discuss the security concepts that can be implemented in an SAP system in [Chapter 11](#).

Now that you have an overview of ERP software, in the next section we'll look at SAP specifically; SAP is currently the market leader in ERP software.

1.3 Introduction to SAP

SAP provides a market-leading ERP software that is used by hundreds of thousands of customers, including a suite of integrated applications across industrial sectors. SAP offers various products, such as Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), Product Lifecycle Management (PLM), Supply Chain Management (SCM), Supplier Relationship Management (SRM), and more. This book focuses on the SAP ERP system—more specifically, SAP ERP 6.0 on the SAP NetWeaver 7.5 platform.

The business functions within SAP ERP's software are called *modules*. As an SAP consultant, you will be working in one or more modules. We'll look at the different types of modules in this section. We'll also discuss the types of users that take advantage of these modules and then look at your specific role as an ABAP consultant. Finally, we'll wrap up this section by looking at what types of modifications, enhancements, and customizations are available to customize software for users.

1.3.1 Modules in SAP

One of the reasons that SAP is popular is because it is very flexible and easily customizable. To achieve this flexibility, an SAP system is divided into various modules. Each SAP module maps to a business process of a business unit. For example, the SAP ERP MM module maps to the Material Management business unit of the organization, the HR module maps to the Human Resources business unit of the organization, and so on.

All SAP modules can be integrated with one another or with third-party systems. For example, you can integrate SAP ERP PP with an HR module from PeopleSoft. Based on the organization, you can implement either one or several SAP modules and integrate them with other ERP systems.

SAP modules are classified into two categories: functional or technical.

Functional Modules

Functional modules cover business processes and functional areas that carry out day-to-day activities of a business, such as the following:

- ▶ Financial Accounting (FI)
- ▶ Controlling (CO)
- ▶ Human Resource (HR)
- ▶ Sales and Distribution (SD)
- ▶ Materials Management (MM)
- ▶ Plant Maintenance (PM)
- ▶ Production Planning (PP)
- ▶ And more

Technical Modules

Technical modules cover the technical side of the SAP system that are required to customize the system based on business needs, including setting up and maintaining the SAP system.

Technical modules include the following:

- ▶ SAP Basis
- ▶ ABAP
- ▶ SAP BusinessObjects Business Intelligence (BI)/SAP Business Warehouse (BW)
- ▶ SAP Exchange Infrastructure (XI)/SAP Process Integration (PI)
- ▶ SAP Enterprise Portal
- ▶ SAP HANA
- ▶ And more

1.3.2 Types of Users

In the previous section, we looked at the functional and technical modules available in SAP. These modules are accessed by different users.

There are three types of SAP users:

- ▶ End users
- ▶ Functional consultants
- ▶ Technical consultants

End users are people who use the SAP system in a production environment to carry out their day-to-day business activities. These users include sales people using the SAP system to create sales orders and HR managers managing human capital, for example.

Consultants are subject matter experts (SME) who help to customize and maintain the SAP system for a business. Consultants also address any issues or customization requests from end users.

In general, a consultant will gain expertise in a particular modules. The consultant's area of expertise determines if he is a *technical consultant* or a *functional consultant*. If you work as an ABAP consultant, you would be a technical consultant with expertise in ABAP programming. Similarly, if you had domain experience in finance, you would be a functional consultant in a finance module. However, many consultants gain expertise in one of the functional modules and one of the technical modules. They are *techno-functional consultants*.

It has become increasingly important for functional consultants to learn ABAP to become techno-functional consultants. Functional consultants with a background in ABAP can work more effectively with ABAP teams and develop a holistic technical understanding of SAP applications.

In the next section, we'll take a deeper look at the role an ABAP consultant plays.

1.3.3 Role of an ABAP Consultant

Because this book focuses on ABAP and ABAP development, let's discuss the role of an ABAP consultant in an organization that has implemented an SAP system.

Even though having a good understanding of business processes helps with application development, it isn't absolutely necessary to know the intricacies of the business to succeed as an ABAP developer. As ABAP developers, we don't usually interact with end users directly. When an end user has a requirement or an issue with an SAP application, he or she will contact the respective functional Center of Excellence (CoE).

A functional CoE consists of functional consultants with expertise in a respective domain who will evaluate a customer requirement and decide if it requires any technical changes. If the user's issue or requirement necessitates any technical changes, then the functional CoE will assign the task to the ABAP CoE. It is the

responsibility of the functional consultant to prepare a functional requirement document explaining the requirement and listing all the essential information to make the changes needed in the application.

For example, if a user asks for some extra information to be shown in a report output, the functional specification will provide the required information, such as the name of the report program that should be changed and the database tables where the extra information can be found to show in the report output.

Once the functional specification is received by the ABAP CoE, it will be assigned to an ABAP consultant, who will prepare a technical specification based on the functional specification and carry out the necessary changes to the application in the development system. The technical specification will capture the required changes from a technical perspective, such as which part of the program code is changed and any extra lines of code added to the program.

Once the changes are made, the application will be tested by the functional consultant and the end user in the quality assurance system before transporting the changes to the production system.

Systems within SAP

We talk about different systems, such as the development system, quality assurance system, and production system, and the transportation mechanism to transport the objects between systems in the discussion of the SAP system landscape in [Chapter 2](#).

The typical work of an ABAP consultant involves production support, enhancements, and new projects. *Production support* involves fixing bugs when an existing application develops a problem and needs troubleshooting. For *enhancements*, consultants enhance an existing application to provide added functionality. For *new projects*, consultants develop new applications to meet business requirements.

The proficiency required from an ABAP consultant varies with the requirement. The more experience you gain in ABAP, the more confident you become in handling complex business requirements. To succeed as an ABAP consultant, a good understanding of ABAP programming language and general understanding of business processes is crucial. ABAP consultants are not restricted to any one business function; they can work on requirements across business functions.

One advantage of being an ABAP consultant, compared to being a consultant in other technical modules, is that with years of experience ABAP consultants gain

not only technical expertise but also good functional knowledge. In [Section 1.4](#), we'll look at various types of developments ABAP consultants work on.

1.3.4 Changing and Adapting the Data Structure

SAP system software is business software delivered with predefined standard SAP applications and configurations to meet the needs of various industry sectors. It is not necessary to change the *repository objects* (programs, applications, etc.) to use an SAP system. However, every company within an industry sector may have its own company-specific requirements. During implementation, SAP software needs to be adapted to fit these enterprise requirements. We refer to this process as *customization*.

There are three types of customization:

▶ **Modification**

The standard SAP code is modified per specific business requirements.

▶ **Customer enhancement**

The standard SAP applications are enhanced without changing the source code of the standard SAP program.

▶ **Customer development**

Customers develop applications in the SAP system.

In the following subsections, we will look at each of these ABAP customization methods.

Modifications

Changes to standard SAP-delivered objects (e.g., programs and table definitions) are called *modifications*, and repository objects delivered by SAP are changed in this process. In other words, with modifications, you directly change the source code of a standard SAP program to extend its functionality. These changes will be lost when the system is upgraded or when a support package is imported. To prevent this, you will need to check these modifications against the new repository when next upgrading your system or when importing support packages. This can escalate the cost and time for such upgrade projects. We always recommend keeping modifications to a minimum and as a last resort in customization.

As an example of implementing a modification, let's say you have a standard SAP application to create a sales order. This standard SAP application allows you to capture certain details while creating the sales order. Your company requires additional details to be captured while creating the sales order as well, which are not available with the standard SAP application. One way of customizing the sales order application is to modify the standard SAP sales order application screen to add your custom fields. However, these custom fields will be lost when you next upgrade the standard SAP application.

We will provide further details about the modification process in [Chapter 20](#).

Customer Enhancements

To create a customer enhancement, you add customer-specific objects to the repository without actually changing the original source code of the program. While developing the standard SAP applications, SAP anticipated the need to extend program functionality. There are specific openings in the code where you can supplement the SAP standard system with your own objects.

You can enhance a standard SAP application only if SAP has anticipated the need and provided an opening or hook to add customer-specific objects. SAP provides a variety of enhancement techniques, such as user exits/customer exits, Business Add-Ins (BAdIs), and explicit and implicit enhancement points to extend the functionality of a standard SAP object.

Not every standard SAP object can be enhanced, and the enhancement options provided may not meet your company-specific requirements. However, most of the standard SAP programs are flexible and support many enhancements.

For example, considering the same scenario of adding custom fields to a standard SAP sales order application, if SAP has provided a screen enhancement to the standard SAP application, you can implement the enhancement to add custom fields. Because SAP calls these enhancements strategically in the standard program, the custom fields will not be lost in an upgrade. By implementing an enhancement, you extend the program without modifying the standard code.

We will discuss the available enhancement techniques further in [Chapter 20](#).

Customer Developments

If you can't find a suitable enhancement option, don't wish to modify a standard SAP program, or can't find a suitable standard SAP object that fulfills your company-specific requirement, you can create your own repository objects (such as tables, programs, transactions, etc.). All customer developments normally take place in a customer namespace, meaning that the names of all objects created by a customer should start with Y or Z.

For example, let's say your company has to implement a new business solution for a specific region due to a change in local laws of that region and there is no standard SAP solution that meets the requirement. In such scenarios, you can develop your custom applications to implement the business requirements. These custom applications will be in a customer namespace and will coexist with standard SAP applications.

Figure 1.3 depicts the customization overview. As you can see, modifications fall under the SAP namespace, because modifications change the standard SAP program directly, whereas enhancements and customer developments fall under the customer namespace and are immune to being changed during upgrades.

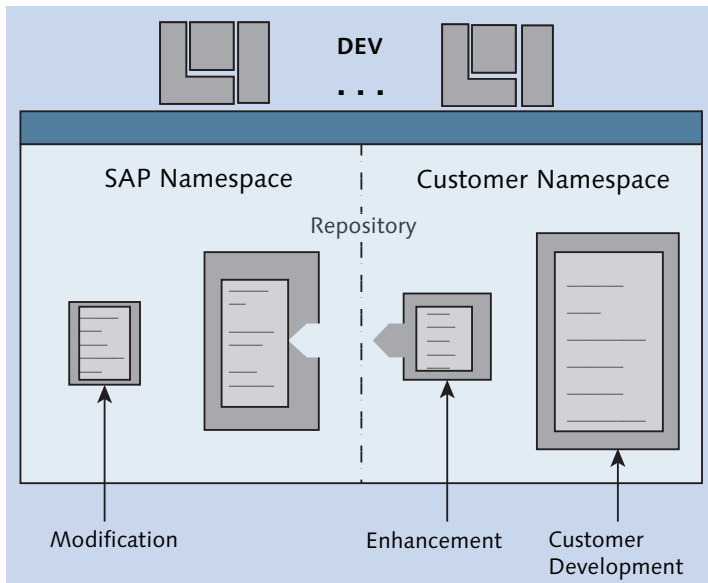


Figure 1.3 Customization Overview

Now that we've explored the roles of ABAP consultants and the changes and adaptations that can be made to the data structure in an SAP system, let's turn our attention to ABAP as a whole.

1.4 ABAP Overview

ABAP is a programming language used to develop applications in SAP. To work with modifications, enhancements, or customer developments, you need knowledge of application development within ABAP.

Let's look at a few of the objects we can develop with ABAP.

1.4.1 Types of Applications

Originally designed for report development within SAP, ABAP has been extended enormously. Currently, we can develop a wide variety of applications in ABAP, both in the traditional SAP GUI environment and in the web environment. These typical developments include reports, forms, interfaces, transactions, and more.

ABAP is a hybrid programming language that supports both procedural and object-oriented programming techniques to meet any complex business requirement.

1.4.2 RICEF Overview

Reports, interfaces, conversions, extensions, and forms (RICEF) is the general acronym used to highlight the different kinds of objects developed in ABAP. The following sections look at each of these areas.

Reports

Reports provide read-only access to a database. ABAP was originally named *Allgemeiner Berichts-Aufbereitungs-Prozessor*, German for *general report creation processor*, before it was rechristened as *Advance Business Application Programming*.

As the original name suggests, ABAP was mostly used for creating reports in SAP in the beginning. Even though ABAP has extended and matured over the decades

and allows you to create many complex applications and interfaces, reports are still common developments within ABAP.

During the reporting process, data is updated in the SAP system by transactions, and is processed by report programs to generate various kinds of reports for users. For example, if you want to check available stock, then you'll need a report program that will pull out this data from the database and present it to you as a report output.

Figure 1.4 illustrates a typical report program. As you can see, data is updated in the SAP database from across the business functions, and the report program can process that data and present it to the user as a report output.

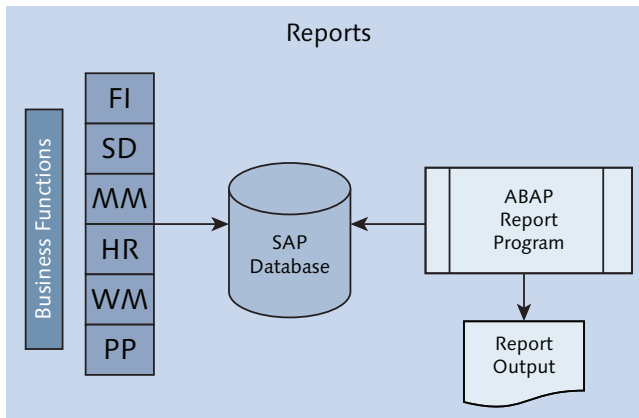


Figure 1.4 Report Programs

Interface Programs

Interface programs allow you to exchange data with other systems, such as a legacy system or a partner system. You typically develop two kinds of interface programs: inbound interface programs and outbound interface programs. For *inbound interface programs*, data is received from an external system and updated in the SAP system. For *outbound interface programs*, data is extracted from the SAP system and pushed to external systems.

Figure 1.5 illustrates inbound and outbound interface program processes in SAP. As you can see, in the inbound interface the data is extracted from the external system and the file is transferred to the SAP system. An ABAP program will process this file and update the data in the SAP system. In an outbound interface, the

data is extracted from the SAP system by an ABAP program, which is then pushed to the external system. The external program will process this data and update the system.

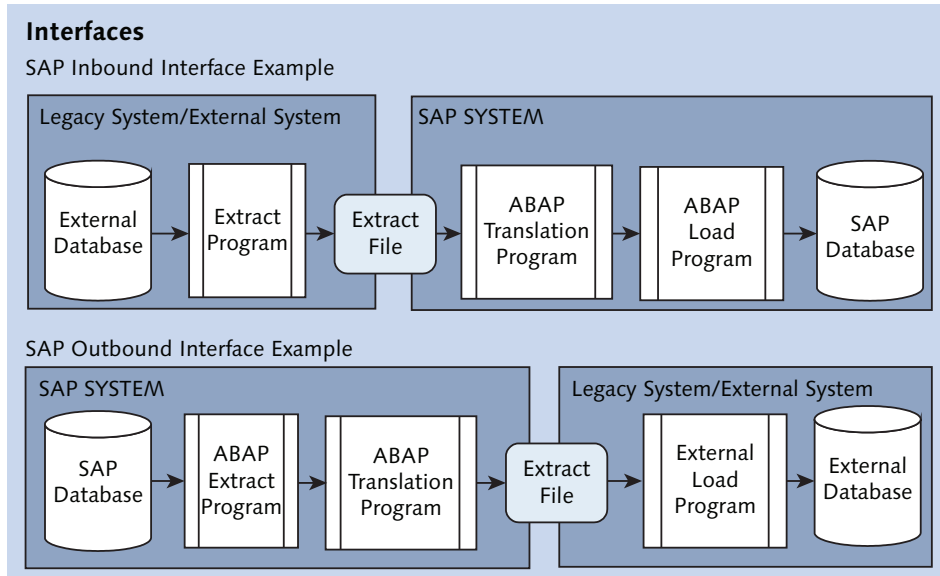


Figure 1.5 Interface Programs

Interfaces are a key concept, and SAP provides various technologies to interact with external systems. We will explore different interface technologies in [Chapter 19](#).

Conversions

Conversion programs move data from a legacy system to an SAP system. Usually, these are one-time-use programs for the initial load of data to an SAP system before go-live.

For example, once you implement a new SAP system, you may want to move all your existing data for customers or materials maintained in your legacy system. In such a case, you would develop conversion programs to process, validate, and update this data in the SAP system.

[Figure 1.6](#) illustrates conversion program processes in SAP.

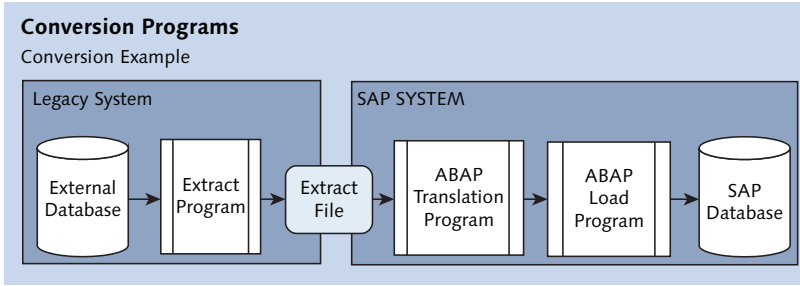


Figure 1.6 Conversion Programs

Similar to inbound interface programs, for conversion programs, data is extracted from the legacy system and transferred to the SAP system, which is then validated by an ABAP load program to post in the SAP system.

SAP provides a tool called the Legacy System Migration Workbench (LSMW) that integrates various conversion techniques to migrate the data from the legacy system to the SAP system. You'll learn more about conversion programs and LSMW in [Chapter 19](#).

Extensions

Extensions are programs that extend the functionality of a standard SAP system. These include modifications, enhancements, new developments, and third-party add-on programs.

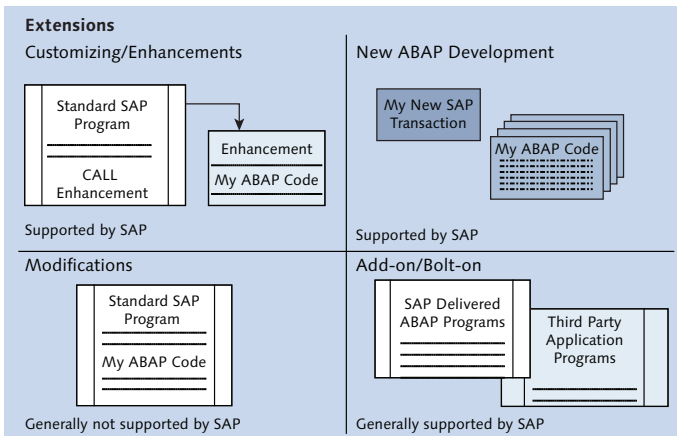


Figure 1.7 Extensions

As shown in [Figure 1.7](#), you can extend the functionality of standard SAP software by implementing SAP-delivered enhancements, modifying standard SAP programs, developing custom programs in a customer namespace, or using third-party extensions.

Forms

Forms are like report programs that provide read-only access to the database. However, forms provide more flexibility in their layout design. Forms are used with documents for which the user wants a physical version (e.g., purchase orders or invoices). These documents should meet legal requirements and should have a specific format.

SAP provides various tools that are integrated with the ABAP Workbench to allow you to design a form layout to meet legal and customer requirements (e.g., SAPscripts, smart forms, or Adobe forms).

As shown in [Figure 1.8](#), a typical form layout contains various windows to print different data from the document. The layout is usually designed to meet the legal requirements for the document to print various data in a specific format.

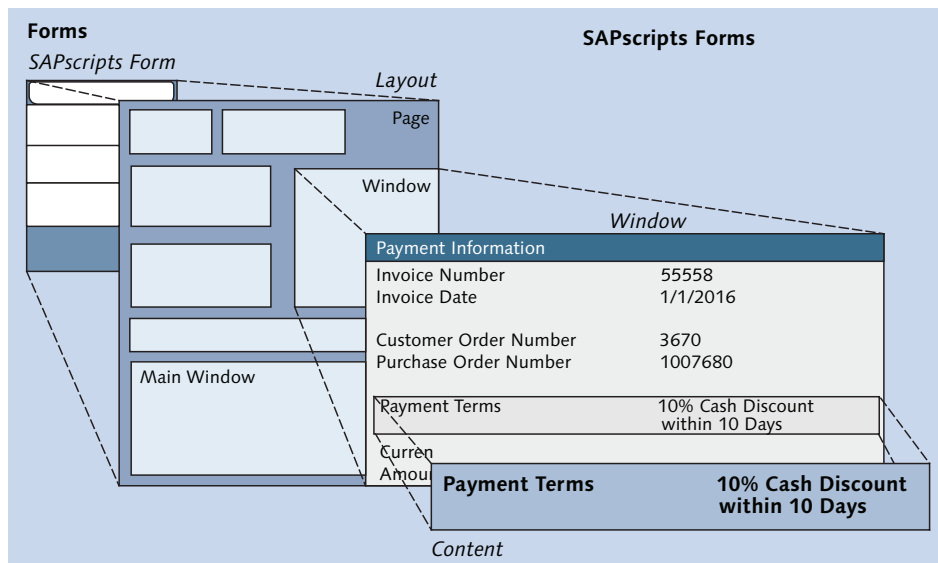


Figure 1.8 Forms

This brief overview of ABAP provided a glimpse into some of the topics that we'll cover in this book. The final section of this chapter looks at the system requirements you'll need to meet to follow along with the examples in this book.

1.5 System Requirements

To work with ABAP development and follow the examples in this book, you need access to an SAP system and a client-side program (SAP GUI or Eclipse IDE). SAP provides the Internet Demonstration and Evaluation System (IDES) for training.

IDES in the SAP ERP system represents a model company and contains application data for various business scenarios that can be run in the SAP system. The business processes in the IDES system are designed to reflect real-life business requirements and have access to many realistic characteristics. To follow the examples in this book, you need IDES for release 7.4.

There are several ways of accessing IDES data:

- ▶ Local IDES system
- ▶ Remote access to an SAP IDES system from either SAP or third-party vendors
- ▶ Access to an IDES system on your notebook

1.6 Summary

In this chapter, we walked through the evolution of SAP and the various product naming conventions enacted by SAP over the years. We discussed the different user types and the role of an ABAP consultant.

Next, we talked about the need for customization and the various ways in which an SAP system can be customized. Finally, we looked at the development objects provided in ABAP. We will explore each of those development objects in greater detail as we progress in this book.

In the next chapter, we'll take a closer look at the architecture of an SAP system.

A technical overview of the SAP system architecture before you begin coding will allow you to better understand the environment in which your programs run.

2 Architecture of an SAP System

Understanding the SAP system architecture from a technical point of view will help you understand how ABAP programs run. In this chapter, we'll explore the system architecture and look at the various layers that make up the SAP system.

We'll start with an introduction to the three-tier architecture of an SAP system in [Section 2.1](#), covering the presentation layer, application layer, and database layer that makes up the system. In [Section 2.2](#), we'll discuss the individual components that make up each of these different layers.

[Section 2.3](#) concludes the chapter with a discussion on data structures, which will help you understand how data is typically stored in an SAP system.

2.1 Introduction to the Three-Tier Architecture

In [Chapter 1](#), we discussed SAP's move towards a three-tier architecture since the introduction of the SAP R/3 system. Here, *R* stands for *real-time* (instantaneous data exchange), and *three* represents *three-tier architecture*.

Three-tier architecture is a client-server architecture in which the user interface (presentation layer), business logic (application layer), and data storage (database layer) are developed and maintained as independent modules (often on separate platforms).

[Figure 2.1](#) illustrates the three layers of the ABAP architecture. As you can see, the presentation layer communicates with the application layer, and the application layer communicates with the database layer. The presentation layer cannot communicate with the database layer directly, which further enhances security for business logic, because the client doesn't have direct access to the database.

Compared to two-tier architecture, the introduction of an additional layer in a three-tier system brings its own complexities to an implementation, but it allows for a highly scalable system. In addition, in a three-tier system you can maintain and update each layer independently. For example, in web server architecture, typically the presentation tier consists of a web browser (Internet Explorer, Google Chrome, etc.), the application tier consists of a web server (Apache, IIS, etc.), and the database tier consists of a database (MSSQL, MySQL, etc.).

Here, each tier exists as independent software and can be changed or updated without affecting the other tiers. Therefore, the user can update his or her browser independently without worrying about breaking anything in the application or the database tier. All the user needs to worry about is whether the browser is compatible with the web server or not. Similarly, the web server and the database can be updated or replaced independently without affecting the software in any other tiers. This independence provides flexibility to update the system in response to changes in either requirements or technology.

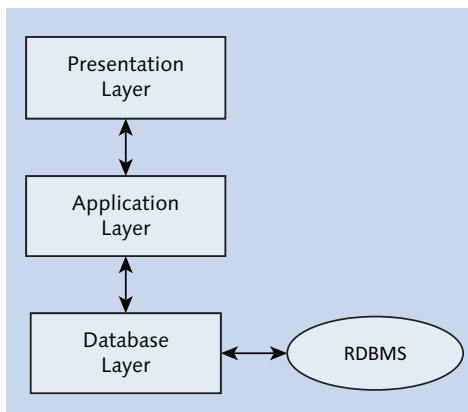


Figure 2.1 Three-Tier Architecture

The following list describes the different layers shown in [Figure 2.1](#):

► **Presentation layer**

The *presentation layer* consists of the user interface, which runs on a client machine that uses a standard graphical user interface (GUI). The user interface can be a web browser or propriety software like SAP GUI. Because the presentation layer can be developed independently depending on the requirements

and system configuration, it can support various devices (PCs, smartphones, etc.) and operating system environments.

The GUI communicates with the application layer by taking user inputs, sending requests to the application server, and displaying the results received from the application server. For example, when you open a website in your web browser, you are requesting a resource from the web server. The web server responds to your request, and the browser presents the result in a user-friendly way.

We'll discuss the presentation layer further in [Section 2.2.1](#).

► **Application layer**

The *application layer* is where all the major work is done by ABAP programs. This is the software layer that hosts various modules of business functions and executes program code. It can consist of one or multiple application servers and a messaging server, which is responsible for communication within the layer.

When you run an application, the execution takes place in the application layer. The application layer can communicate with the database to process the data requested by the user.

We will explore the various components of the application layer in [Section 2.2.2](#).

► **Database layer**

The *database layer* is where all the persistent data is stored. The database is typically managed by a *relational database management system (RDBMS)*. The RDBMS provides a declarative method to maintain data and queries, and the system takes care of describing the structures for storing the data.

In an RDBMS model, the data is stored in one or more *tables*. Each table consists of rows and columns, with each row identified by a unique key. Tables are related to one another with *foreign keys*. If the key field of one table exists as a non-key field in another table, then the two tables have a foreign key relation. Foreign keys allow you to define a parent-child relationship between different tables of the database. You'll learn more about RDBMS in [Chapter 5](#).

SAP supports various database systems, like SAP HANA, Oracle, MSSQL, MaxDB, and IBM DB/2.

We will discuss the database layer further in [Section 2.2.3](#).

2.2 SAP Implementation Overview

When implementing an SAP system, organizations typically implement at least three separate systems: one for development, one for quality assurance/testing, and one for a production environment. Each of these SAP systems have three layers—namely, the presentation layer, application layer, and database layer.

In this section, we'll discuss each individual layer in more detail and each layer's various components.

2.2.1 SAP GUI: Presentation Layer

There are two options for the presentation layer: a system with SAP GUI installed to access the SAP system and work with classic SAP GUI-based applications, or a web browser to work with modern web-based applications. SAP GUI is a program installed on a user's workstation. You can also use a web browser to access web-based applications without installing SAP GUI.

The presentation interface allows users to interact with the SAP system and access various SAP applications. The requests from the presentation interface are sent to the application server for processing. The results are then presented to the user. The presentation interface (SAP GUI or web browser) captures the user actions on screen, and also takes care of formatting the output received from the application server.

Depending on the operating system on the user's workstation, the user may choose to install the compatible SAP GUI version. For example, if you're using a Windows PC, you can install SAP GUI for the Windows environment. Similarly, if you are using a Mac- or Linux-based system, you can install SAP GUI for the Java environment. If you simply wish to use a web browser to access the classic SAP GUI-based applications, you can install SAP GUI for HTML.

Figure 2.2 depicts the following SAP GUI versions:

- ▶ **SAP GUI for the Windows environment**

SAP GUI for Windows is the most popular version. Via an EXE file, the user installs the GUI on his or her workstation. SAP GUI for Windows supports all the functionality to develop and access all SAP GUI-based applications.

- ▶ **SAP GUI for the Java environment**

SAP GUI for Java allows users to work with SAP GUI-based applications when

operating Mac or Linux systems. Certain application development features are not supported in this version of SAP GUI, such as the graphical PC editor. We recommend using SAP GUI for Windows to take advantage of all development features.

► **SAP GUI for HTML (Web GUI)**

SAP GUI for HTML is SAP's attempt to enable all SAP GUI-based applications for the web. For these applications, the GUI maps SAP screens to HTML pages.

SAP GUI for HTML requires the server to run Internet Transaction Server (ITS). ITS performs one-to-one mapping of SAP screen fields (i.e., input fields, text fields, etc.) to SAP GUI for HTML fields. Even though SAP GUI for HTML is the fastest way to enable SAP applications for the web, it isn't very popular with end users.

Because web browsers communicate using the HTTP/HTTPS protocol, the Internet Communication Manager (ICM) ensures that communication between the SAP system and the outside world—via HTTP, HTTPS, and SMTP protocols—works properly, as shown in [Figure 2.2](#). Without the ICM component, it isn't possible to use a web browser to communicate with the application server.

SAP GUI sends requests to the dispatcher in the application server (see [Figure 2.2](#)), which takes care of assigning the task to an available *work process* and also takes care of returning the result to the GUI.

We will discuss the different components of an application server further in [Section 2.2.2](#).

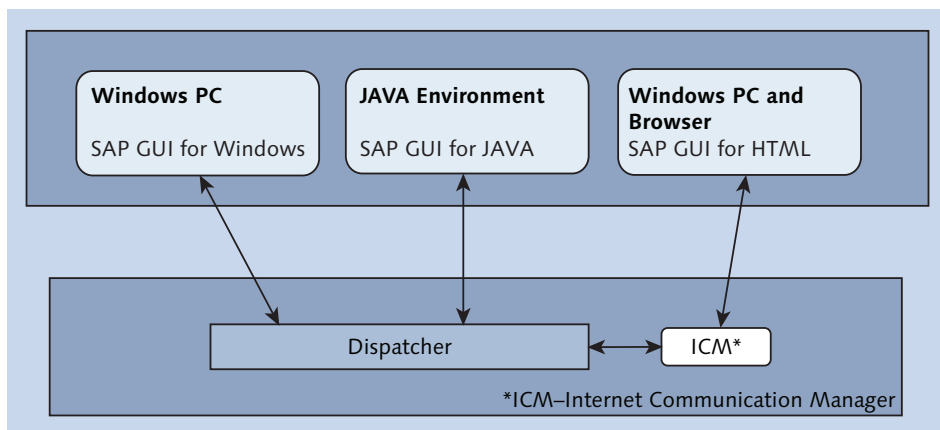


Figure 2.2 SAP GUI

2.2.2 Application Servers and Message Servers: Application Layer

ABAP programs run in the application layer, and all program execution takes place in the application server. The application layer includes one or more application servers and a message server. The message server is responsible for communication between the application servers. The application server interprets ABAP programs by providing the runtime environment to execute ABAP code.

The application server consists of various components, as shown in [Figure 2.3](#).

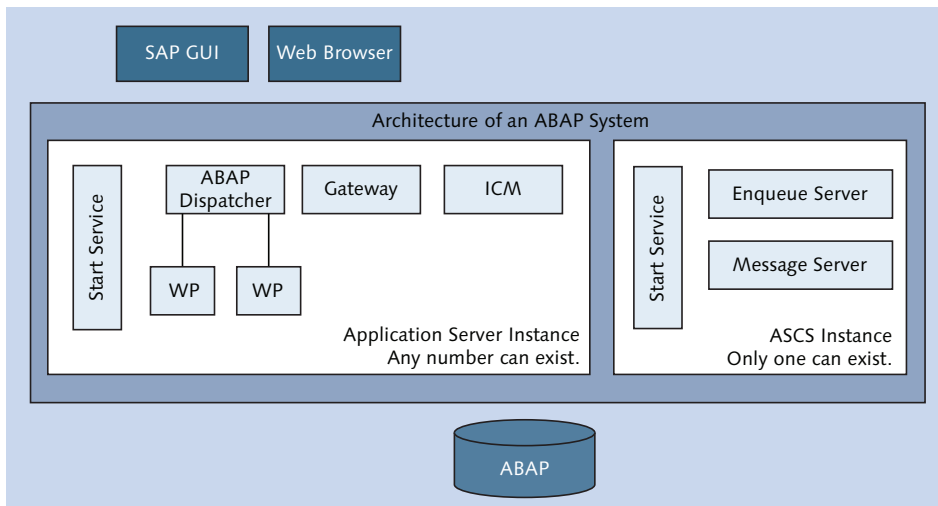


Figure 2.3 ABAP Application Server

SAP NetWeaver is the application server that provides the runtime environment for both ABAP and Java programs. It can be installed as a pure ABAP system, a pure Java system, or a dual-stack system:

- ▶ **Pure ABAP system**

A pure ABAP system supports developing and running ABAP-based applications, but offers limited support for running SAP Java-based applications.

- ▶ **Pure Java system**

A pure Java system supports the Java platform and Java Enterprise Edition (Java EE) applications. It provides full infrastructure support for developing and running Java EE applications, but it doesn't provide support for ABAP applications.

► **Dual-stack system**

A dual-stack system offers the best of both worlds by providing support for both ABAP-based and Java EE-based applications.

ABAP consultants are not directly involved with the installation or maintenance of an application server. This is the role of an SAP Basis consultant. However, because ABAP programs run in the application server, understanding the various components of the application server (and their roles) help to understand the backend process that takes place when an ABAP program is executed.

Discussing all of the components in an application layer is beyond the scope of this book. Therefore, we'll limit our discussion to some of the important components that make up the application layer.

The various components of an SAP system are collectively called *instances*. These instances process user requests, and we can have any number of application server instances.

Some of the major components of the application server instance are as follows:

- ICM
- ABAP Dispatcher
- SAP Web Dispatcher
- Gateway
- SAP start service
- Work processes

Each application server instance can have multiple work processes but can't have more than one ICM, dispatcher, gateway, and SAP start service.

Along with multiple application server instances, there is a special instance called the *ABAP System Central Services (ASCS) instance*. The ASCS instance includes the message server, the enqueue server, and a separate start service. The ASCS instance handles managing locks, exchanging messages, and load balancing.

The *message server* takes care of load balancing and communication between multiple instances, whereas *the enqueue server* manages application locks. Application locks ensure that no two users edit the same data at the same time.

Let's look the various components of an application server instance in a bit more detail.

Internet Communication Manager

The *ICM* enables the application server to communicate with the outside world using web protocols like HTTP, HTTPS, or SMTP. The *ICM web-enables* the application server. This allows users to use a web browser to make requests to the application server, as opposed to using SAP's proprietary GUI.

ICM runs as a separate process, started and monitored by the ABAP Dispatcher. With ICM, the application server can process requests from the Internet, such as Web Dynpro-based applications.

ABAP Dispatcher

Each application server has a dispatcher. Requests are processed via work processes. The role of the dispatcher is to assign requests to work processes. A user makes a request, the request is sent to the dispatcher, and the dispatcher assigns the request to a freely available work process.

Once the request is assigned to the work process, the work process executes the request and sends back the result. The work process then becomes free, ready to take up another request.

When a user logs in to an SAP system, the logon request goes to the message server, which performs load balancing and connects the user to the dispatcher of an instance with the least load. Once connected, the dispatcher of that instance handles all the requests by the user.

Work Processes

In [Figure 2.3](#), "WP" stands for *work processes*. User requests are processed through work processes. There may exist multiple work processes per application server instance.

As shown in [Figure 2.4](#), when a user makes a request, the request is sent to the ABAP Dispatcher. The ABAP Dispatcher, in turn, assigns the request to the available work process for processing. Once the work process completes the request,

it sends back the result, and the work process becomes free again to take a new request.

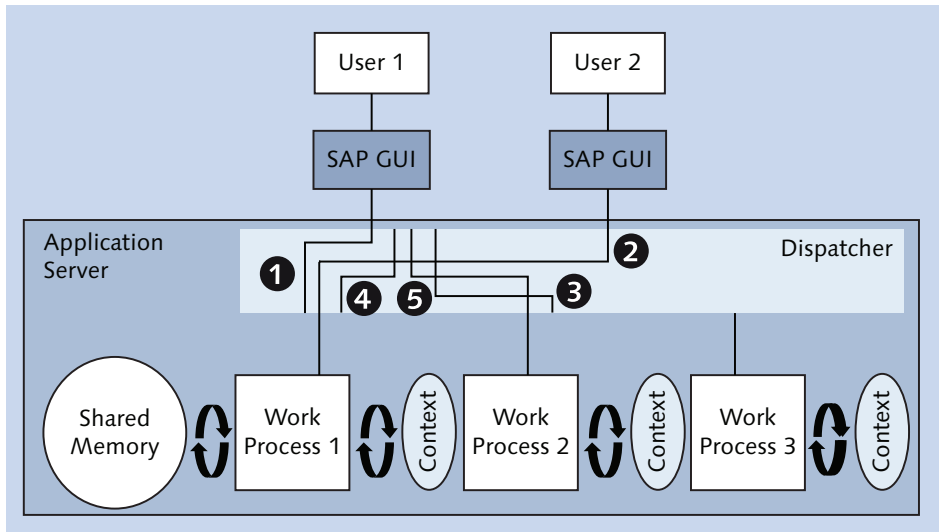


Figure 2.4 Dispatcher and Work Process

Work processes are not dedicated to any user session. They can take requests from any user. For example, as shown in [Figure 2.4](#), User 1 executes a report. This request is assigned to Work Process 1 by the dispatcher. Work Process 1 processes the request and sends the result back to User 1.

At this point, if User 2 makes a request, the dispatcher can assign it to Work Process 1, because that work process is free to take a new request. If User 1 makes a new request while Work Process 1 is processing the request from User 2, then the dispatcher will simply assign the User 1 request to Work Process 2. If a new request is made by either User 1 or User 2 while both Work Process 1 and Work Process 2 are busy, then the dispatcher will assign this task to the next available work process.

You can see how the work process statuses change by opening Transaction SM50 (see [Figure 2.5](#)). If you run any program and check Transaction SM50, you'll notice that the request is processed by a dialog work process (DIA) with a RUNNING status. The work processes that are free to take requests have a WAITING status.

Numb..	Type	Process ID	Process Status	Info.	F
0	DIA	4.656	Waiting		
1	DIA	5.836	Running		
2	DIA	5.852	Waiting		
3	DIA	4.680	Waiting		
4	DIA	4.688	Waiting		
5	DIA	3.292	On Hold	Debugging	
6	UPD	4.704	Waiting		
7	BTC	4.712	Waiting		
8	BTC	4.720	Waiting		
9	BTC	4.728	Waiting		
10	SPO	4.736	Waiting		
11	UPD2	4.744	Waiting		

Figure 2.5 Transaction SM50: Work Process Overview

As shown in [Figure 2.5](#), there are different types of work processes that handle different types of requests. [Table 2.1](#) lists the available work process.

Work Process Type	Purpose
DIA (Dialog)	Handles dialog requests (i.e., requests made from users through screens)
UPD/UPD2 (Update)	Handles high- and low-priority updates to the database
BTC (Background processing)	Handles background jobs
SPO (Spool)	Handles spool requests

Table 2.1 Work Process Overview

The SAP Basis administrator can configure the number of work processes for each type by using Transaction RZ10.

Gateway

The gateway reader enables communication with external systems using remote function calls (RFCs), which are based on TCP/IP. RFC services enable external applications to communicate with the SAP system. RFCs can be made in ABAP programs or by using interfaces with external programs (e.g., RFC-enabled function modules).

In addition to using RFCs between systems, gateways can also be used between processes of an instance.

Start Service

A *start service* starts the SAP system instance. It is implemented as a service in Windows and as a daemon in Unix. A start service helps start and stop instances, monitor the runtime state, and read logs.

A start service also provides technical information, such as information about active sessions or network ports.

Message Server

Only one message server can exist per SAP system. A *message server* helps different application server instances communicate with each other.

A message server also handles load balancing. As discussed earlier, when the user logs into an SAP system, the message server connects the user to the application server instance with the least load. Note that the message server handles the load balancing for SAP GUI and RFC connections, whereas the load balancing in a web environment is handled by the SAP Web Dispatcher.

SAP Web Dispatcher

SAP Web Dispatcher is useful in web environments. It lies between the Internet and the SAP system, which means that all HTTP(S) requests are routed through SAP Web Dispatcher. It can be placed in front of multiple SAP systems and configured to route specific requests to specific systems for better load balancing. You can also configure it to accept or reject connections, enabling better security.

SAP Web Dispatcher can reject certain URLs, allowing for restricted access to the system through URL filtering. It can also be used as a web cache to improve application response time.

Enqueue Server

The *enqueue server* manages locks. It can be part of the classical ABAP central instance, or it can be installed as a standalone server that isn't part of SAP NetWeaver AS ABAP.

The enqueue function provides logical locking functionality that helps to prevent multiple users from updating the same data at the same time. For example, enqueue functions are used to prevent multiple users from editing the same

document at the same time. The moment one user enters the change mode to edit a document, enqueue functions can prevent other users from entering the change mode to edit the same document.

User Context

User context is where contextual information about the user is stored. When a user makes a request, the work process uses the information about the user available in the user context area. The user context area stores information like the user's personal settings, authorizations, and date and currency formats. The work process applies this information to each request by the user.

For each user, the user context is loaded once into the application server on each user login and deleted when the user logs off from the system. The data in the user context is not refreshed until the next login for that user.

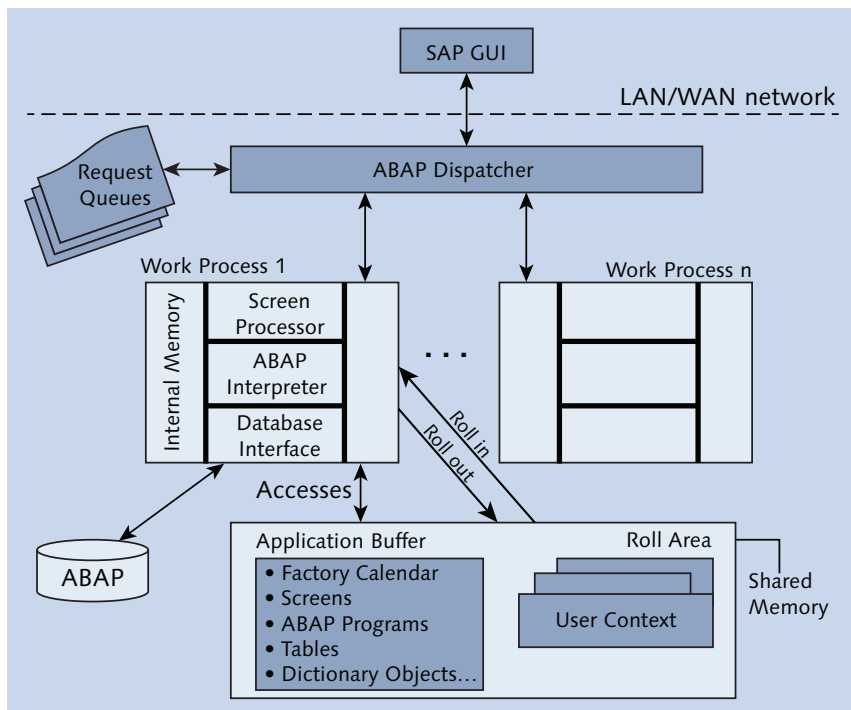


Figure 2.6 Roll-In and Roll-Out of User Context

As shown in [Figure 2.6](#), before processing the request from the user, the work process will roll in the user context, apply it to the request, and roll out the user context once the request is processed. This allows the work process to handle requests from any user and apply user-specific context to each request.

2.2.3 Database Server/RDBMS: Database Layer

Each SAP system consists of a central database in which the entire dataset is stored. All the data is stored here, including application data, administration data, customizing settings, and ABAP source code. The database layer consists of the RDBMS.

SAP supports a wide range of database systems, such as SAP HANA, Oracle, Microsoft SQL Server, IBM DB2, SAP liveCache, and SAP MaxDB. SAP NetWeaver 7.5 is optimized for SAP HANA. It enables the use of SAP HANA as the underlying database and facilitates the optimization of applications for SAP HANA.

The RDBMS allows you to spread data across multiple tables and link the tables using foreign keys. A table consists of columns (fields) and rows (records). Each table includes a key, which uniquely identifies the row.

For example, say you want to store an item's details in the database table. Let's call this table `TB_ITEM`. The table consists of four columns: `ITEM_ID`, `DESCRIPTION`, `QUANTITY`, and `UoM` (unit of measure), as shown in [Table 2.2](#).

ITEM_ID	DESCRIPTION	QUANTITY	UoM
1	Pen	10	PC
2	Pencil	20	Box
3	Paper	10	KG

Table 2.2 Example `TB_ITEM` Showing Rows and Columns

Because each row in [Table 2.2](#) can be uniquely identified by the `ITEM_ID` field, we will keep `ITEM_ID` as the key field of the table.

This looks fine for basic information storage. However, since our SAP system supports multiple languages, say that users want to see the item descriptions in different languages. How will you expand this table to accommodate the descriptions in different languages? One way is to add a `LANGUAGE` field to the table and maintain the descriptions in multiple languages, as shown in [Table 2.3](#).

ITEM_ID	LANGUAGE	DESCRIPTION	QUANTITY	UOM
1	English	Pen	10	PC
1	German	Stift	10	PC
2	English	Pencil	20	Box
2	German	Bleistift	20	Box
3	English	Paper	10	KG
3	German	Papier	10	KG

Table 2.3 TB_ITEM Expanded to Accommodate Language Key

In [Table 2.3](#), the `ITEM_ID` field is no longer unique. The primary key of the table should always be unique, so the key of this table should be expanded to include the `LANGUAGE` field as well. Therefore, the key of [Table 2.3](#) is a combination of the `ITEM_ID` and `LANGUAGE` fields. Individually, the data in `ITEM_ID` field can repeat n number of times and the data in `LANGUAGE` field can repeat n number of times, but the combination should always be unique.

As you may have noticed, in [Table 2.3](#) we're repeating the same data multiple times to accommodate various languages. When more languages are added to the mix, you end up with redundant data. What if we could separate the descriptions and maintain them in a separate table that can be linked to this table with foreign keys?

[Table 2.4](#) stores the item details, and [Table 2.5](#) stores the descriptions in multiple languages. This arrangement of tables avoids data redundancy, and because the tables are now linked through a foreign key we can query [Table 2.5](#) for the description in the preferred language using the `ITEM_ID` field of [Table 2.4](#).

ITEM_ID	QUANTITY	UOM
1	10	PC
2	20	Box
3	10	KG

Table 2.4 TB_ITEM without Description

ITEM_ID	LANGUAGE	DESCRIPTION
1	English	Pen
1	German	Stift
2	English	Pencil
2	German	Bleistift
3	English	Paper
3	German	Papier

Table 2.5 New Storing Item Descriptions

This is how you model data in an SAP system. For example, the material master data is stored in table `MARA`, the material description in multiple languages is stored in table `MAKT`, and the plant data for material is stored in table `MARC`. With this arrangement, you can query different tables to get different information about the material using the foreign key concept.

In this section, we discussed the different layers of SAP's three-tier architecture in detail. [Figure 2.7](#) summarizes the three layers of the SAP system.

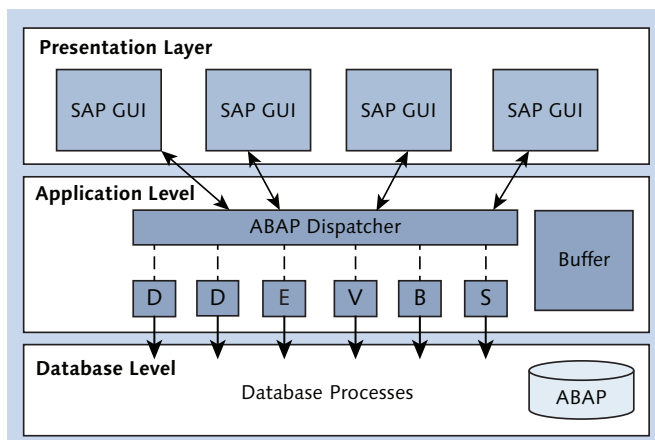


Figure 2.7 Overview of Three Layers

As shown in [Figure 2.7](#), the application server includes a special area called the *buffer*. A buffer acts like a cache in which data that is frequently accessed but seldom changed can be stored. This allows you to improve the performance of the applications, because the data can be accessed from the local buffer. A buffer also

helps to reduce the database and network load, as the application server need not access the database repeatedly for the same information.

Buffering data is recommended when data in the tables does not change frequently and having access to up-to-date data is not critical for the application. If any data changes in the database table, a background process will update the buffer at a fixed interval. This means that buffering a frequently changing table is counterproductive.

In the next section, we'll look at data structures and clients.

2.3 Data Structures

The SAP system typically contains various data, such as master data, user data, and transaction data. Each SAP system further consists of multiple *clients*. A *client* represents an organizational unit in an SAP system.

In SAP systems, the following terms must be differentiated with regard to the data structure:

- ▶ **Client-specific data**

Data that exists within a specific client of the system

- ▶ **Cross-client (client-independent) data**

Data that exists across all the clients of the system.

In other words, when we store any data in an SAP system, the data can be specific to a client or it can be accessed centrally from any client.

In this section, we will discuss the concept of a client, the significance of using different clients, and how our decisions about clients influence the data stored in an SAP system.

2.3.1 Client Overview

Each SAP system can be divided into multiple clients, and each client is identified by a three-digit numeric value. Each client has its own master data, user data, and transactional data. The data in a client is known as *client-specific data*.

Each SAP system logon must include a client ID (see [Figure 2.8](#)), which indicates a logon to a specific client.

SAP

New password: 3 digit client ID

Client: 100

User:

Password: *****

Logon Language: EN

Figure 2.8 Client ID on Logon Screen

2.3.2 Client-Specific and Cross-Client Data

As mentioned previously, data in an SAP system is either client-specific or cross-client. Client-specific data is visible only in the client in which the data was created, whereas cross-client data is accessible from any client in the system.

Because the data in an SAP system is always stored in a database table, the data is restricted to a client by maintaining a client column (field) in the application table. These tables are called client-specific tables and the client columns (see Figure 2.9) are called MANDT.

Dictionary: Display Table

Transparent Table: MARA Active

Short Description: General Material Data

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3		Client
MATNR	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MATNR	CHAR	18		Material Number
.INCLUDE	<input type="checkbox"/>	<input type="checkbox"/>	EMARA	STRU	0		Data Division MARA
ERSDA	<input type="checkbox"/>	<input type="checkbox"/>	ERSDA	DATS	8		Created On
ERNAM	<input type="checkbox"/>	<input type="checkbox"/>	ERNAM	CHAR	12		Name of Person who Created the O

Client field/column in table to restrict the data to a client. This makes the data client dependent.

Transparent Table: TVARV Active

Short Description: Table of variables in selection criteria

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
NAME	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	RVARI VNAM	CHAR	30		ABAP: Name of Variant Variable
TYPE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	RSSCR_KIND	CHAR	1		ABAP: Type of selection
NUMB	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	TVARV_NUMB	NUMC	4		ABAP: Current selection number
SIGN	<input type="checkbox"/>	<input type="checkbox"/>	TVARV_SIGN	CHAR	1		ABAP: ID: I/E (include/exclude values)

Missing client field/column in table. This makes the data cross-client.

Figure 2.9 Client Field in Table

The `MANDT` field stores the current client number in which the data is generated. While accessing the data from the client-specific tables, the system automatically matches the user's current client with the `MANDT` field of the table, restricting the data to the current client.

For example, say a purchase order is created while logged on to client 100. The relevant purchase order application-related database tables update with the new purchase order data, and the `MANDT` fields of those tables will be updated with the client number (100 in this case) automatically. If you try to open the purchase order in another client—say, client 200—the system won't show the purchase orders created in client 100, because it will try to match the `MANDT` field with 200.

For cross-client tables, the `MANDT` field isn't part of the table key, so the data isn't restricted to the client. The data can be accessed from any client of the system. To find out if the data in a table is client-dependent or cross-client, check if the table has `MANDT` as the first field and as part of the key, as shown in [Figure 2.9](#).

For example, say that the material master table `MARA` has `MANDT` as the first field and part of the table key. This makes table `MARA` client-specific. If you create a new material in client 100, then the material data is visible only in client 100. You can't access this data from other clients of the system. Similarly, if the material data is changed in a particular client, then the change won't affect the data in other clients of the system. Application data is generally client-specific, and customizing data is generally cross-client.

The client concept entails splitting your SAP system into multiple clients and separating the data by either restricting it to a particular client or keeping it cross-client.

When you log on to an SAP system, you're logging onto a particular client. Any activities carried out in the system are always carried out in one specific client. When planning the SAP system landscape, you can create various clients for various activities. For example, you can have one client for development, one client for testing (QA), and one client for production.

Some of the most important client roles are as follows:

► **Customizing and development client (CUST)**

The SAP system software is standard business software and is adapted to company-specific business requirements when implemented. To customize the SAP

software and carry out various development activities, each SAP system has a CUST client.

▶ **Quality assurance client (QTST)**

Once the developments are completed in CUST, they need to be tested thoroughly before the application can be used in production. The client in which these tests are carried out is called QTST. The QTST client is generally refreshed with production data at regular intervals so that applications can be tested with more realistic production data.

▶ **Production client (PROD)**

The production environment uses a different client, PROD. PROD is where actual business activities take place. There are no development or testing activities in this client in order to ensure smooth business operations.

These three clients (CUST, QTST, and PROD) are central clients that exist in every system landscape. In general, these three clients are maintained in separate systems and SAP instances, with some transportation mechanism in place to move objects between clients.

Each system will have precisely one client for each of these client roles. The development system will have a CUST client, the quality system will have a QTST client, and the production system will have a PROD client.

For example, ABAP programs are cross-client. When you create a program in one client, it is made automatically available in all the clients of that system. If you change the program in a CUST client, the changes automatically will be reflected in the QTST and PROD clients (if all the clients are set up in the same SAP instance). If all clients are set up in the same instance, you won't have the opportunity to test your program changes. By keeping the clients in separate systems, you can make your changes in the development system, transport them to the quality system for testing, and once you're satisfied, import them to the production system.

In order to maintain consistency across clients, we recommend that all the Customizing settings be carried out in a single Customizing client and transported (with the Change and Transport System [CTS]) to other clients. This helps prevent situations in which applications behave differently in different clients due to different customization settings.

We also recommend that no Customizing settings or workbench developments be carried out in the QTST or PROD clients. If you establish the appropriate client settings using Transaction SCC4 to prevent any development activities in the QTST and PROD clients, this ensures that development activities are always carried out in the CUST client and transported to other clients for testing and subsequent use by end users in the production environment.

2.3.3 Repository

The *repository* is a central store for all ABAP Workbench development objects, such as ABAP programs, screens, menus, function modules, and ABAP Data Dictionary objects.

As discussed in [Section 2.2.3](#), all data, including the source code of ABAP programs and table definitions, is stored in database tables. These tables are not restricted by the client field (MANDT), making them cross-client. Therefore, if you create a table or a program in one client, it is automatically available in all clients. All repository objects are cross-client.

In other words, a table is cross-client, but the data in a table can be made client-specific or cross-client by including or ignoring the client field.

2.3.4 Packages

Repository objects are organized into *packages*. Packages contain related development objects, like programs or tables.

If you create multiple reports, a class, and a few tables for a project, keeping these semantically linked objects in one package helps to identify the related development objects for a project. This allows you to easily identify the dependent objects that need to be changed when the project requirement changes. Packages also transport all related objects together between clients.

Packages can be broadly classified as transportable or non-transportable:

► **Transportable package**

A *transportable package* is a package in which transport attributes are defined. The objects in the transport package can be transported to the target system.

When developing a new application, you start in the development client and transport it to the testing client for testing. Finally, you transport the

application to the production client for end users to use. In this process, you keep all the development objects and customization settings of the application in a transportable package to facilitate their transport to the target system using CTS.

► **Non-transportable package**

A package with no defined transport attributes is called a *non-transportable package*. The objects in a non-transportable package can't be transported between clients and are called *local objects*. A local object—for example, a test program—is not intended to be transported or used in a production environment.

Figure 2.10 shows the development objects grouped into the standard SAP package named VF. All objects are grouped into different folders based on the type of object for easy identification. You easily can look up all related objects for a given application. For example, if you want to find all the related transactions, tables, enhancements, and so on for billing documents, you can open the VF package in the Object Navigator (Transaction SE80), as shown in Figure 2.10. You can check the package to which a transaction belongs by using Transaction SE93.

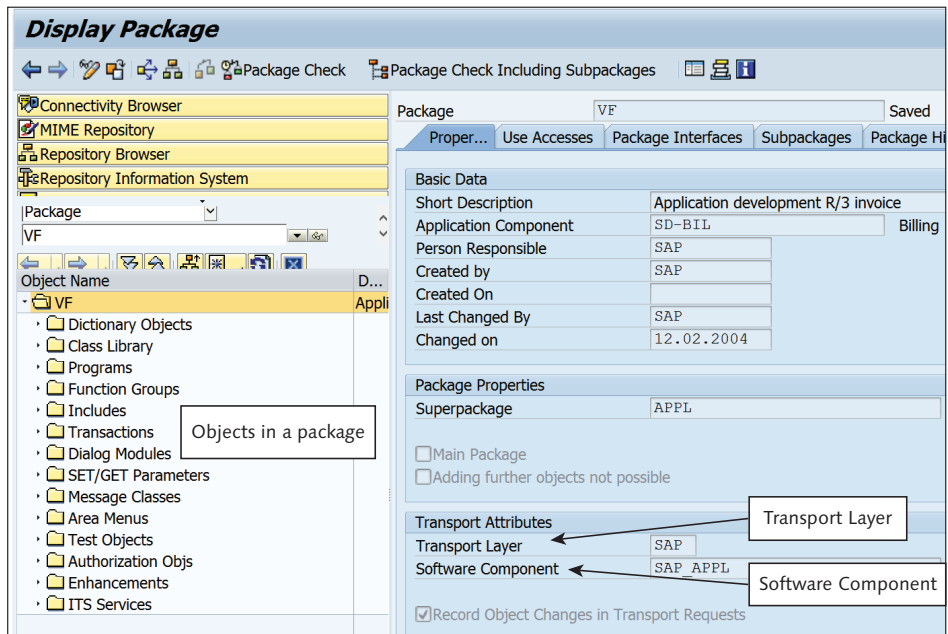


Figure 2.10 Checking the Package

2.3.5 Transport Organizer

Objects are transported between clients using CTS. The *Transport Organizer* is a tool within CTS for managing the objects that change during the development and customization phases. The Transport Organizer delivers the changed objects to the target client (QTST or PROD).

The CTS uses *requests* and *tasks* to manage objects. These objects are added to a task, and the tasks are grouped into requests. To transport an object, all the tasks in a request are released, and then the request itself is released. Once the request is released, the objects are transported to the target system automatically (or manually by the administrator). The target system to which the object is transported depends upon the transport routes of the package to which the object is assigned.

[Figure 2.11](#) shows the requests and tasks in the Transport Organizer.

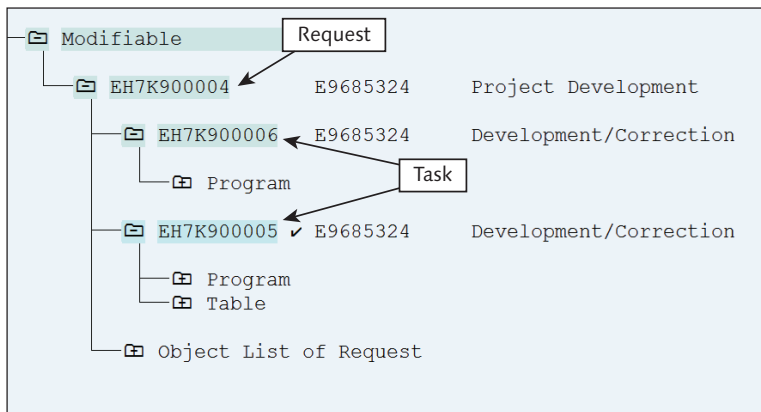


Figure 2.11 Requests and Tasks in Transport Organizer

[Table 2.6](#) lists some of the important Transport Organizer transaction codes.

Transaction	Description
SE01	Transport Organizer (extended view)
SE09/SE10	Search transport request by user
STMS	Transport Management system
SE03	Transport Organizer tools

Table 2.6 Important Transport Organizer Transactions

Certain naming conventions are followed when creating packages. [Table 2.7](#) lists the naming conventions used for packages.

Package Naming Convention	Explanation
Names beginning with a letter from A to S or from U to X.	<ul style="list-style-type: none"> ▶ Package names beginning with these letters are reserved for standard SAP objects; customer objects cannot be created in these packages. ▶ The objects in these packages can be transported as the Transport Organizer records changes to the objects in these packages. As shown in Figure 2.10, these packages belong to the SAP transport layer and are assigned to an SAP software component (e.g., SAP_BASIS, SAP_APPL, etc.).
Names beginning with Y or Z	<ul style="list-style-type: none"> ▶ Customer objects are created in these packages; the objects in these packages can be transported as the Transport Organizer records changes to the objects in these packages. ▶ These packages are assigned to the software component HOME.
Packages belonging to a prefix namespace	<ul style="list-style-type: none"> ▶ The objects in the prefix namespace begin with a forward slash (/). Like other repository objects (such as programs, tables, etc.), packages can also belong to a prefix namespace. ▶ Customer objects are created in these packages; the objects in these packages can be transported as the Transport Organizer records changes to the objects in these packages.
Names beginning with T (private test packages)	<ul style="list-style-type: none"> ▶ These packages allow you to create objects in a customer namespace (Y or Z) or objects from the prefix namespace (objects starting with /). ▶ This package doesn't belong to a transport layer, so the objects in this package can't be transported using the Transport Organizer. However, if the package is linked to the Transport Organizer, the objects are recorded in nontransportable local requests.

Table 2.7 Package Naming Conventions

Package Naming Convention	Explanation
Names beginning with T (private test packages) (Cont.)	<ul style="list-style-type: none"> ▶ Objects in these packages can be transported to other systems only by special transport requests, such as transport of copies or relocation transports. Newly installed SAP systems contain a private test package by default with the name TEST. This package is not connected to the Transport Organizer. ▶ These packages are assigned to the software component LOCAL.
Names beginning with \$ (local packages)	<ul style="list-style-type: none"> ▶ Objects in the customer namespace or prefix namespace can be created in these packages. ▶ The objects in these packages can't be transported; these packages don't belong to a transport layer, and the Transport Organizer doesn't record any changes to these objects. These packages are used for local objects. ▶ A newly installed SAP system contains the local package (\$TMP) by default.

Table 2.7 Package Naming Conventions (Cont.)

Often, you'll be using an existing package to save your development objects. The package used to save the repository object depends on the transport layer (the client to which the repository object will be transported) and the application area (the business function to which the repository object belongs).

Organizing and transporting development objects varies widely across organizations. Always check the internal process and coding standards followed by your organization.

Follow these steps to create a new package:

1. Open Transaction SE21 or Transaction SPACKAGE to access the Package Builder tool.
2. On the PACKAGE BUILDER: INITIAL SCREEN, select the PACKAGE radio button and enter the name of the package. Make sure to follow the namespace conventions, as listed in [Table 2.7](#). Once you enter the name of the package, click the CREATE button shown in [Figure 2.12](#).

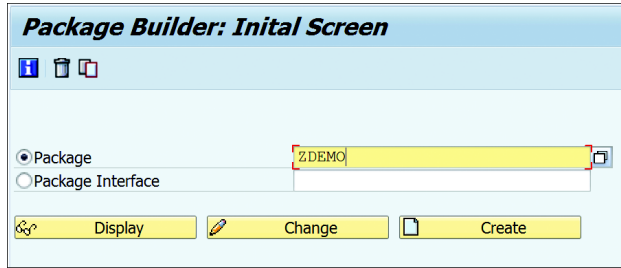


Figure 2.12 Package Builder Initial Screen

3. After clicking the CREATE button, a dialog box will appear. In the CREATE PACKAGE dialog box, maintain the attributes of the package, as shown in [Figure 2.13](#), and click OK. Only the SHORT DESCRIPTION field is mandatory.

The following are the important attributes:

- ▶ **PACKAGE:** Provide the name of the package.
- ▶ **SHORT DESCRIPTION:** Provide a short description for your package.
- ▶ **APPLICATION COMPONENT:** Use the F4 help to select the appropriate application component based on the development objects that will be saved in this package.
- ▶ **SOFTWARE COMPONENT:** This is selected as HOME for customer packages.
- ▶ **TRANSPORT LAYER:** Select the transport layer based on the target system for the objects. You may have to consult your manager or SAP Basis administrator to select the correct transport layer.
- ▶ **SUPERPACKAGE:** Packages can be nested, so you can embed one package into another in a package hierarchy. This allows for better modularization. Provide the name of the superpackage in this field if you want to nest the current package as a subpackage in an existing package.
- ▶ **PACKAGE TYPE:** Select the package type that you wish to create. You can create the package as a STRUCTURE PACKAGE, MAIN PACKAGE, or DEVELOPMENT PACKAGE:
 - **STRUCTURE PACKAGE:** This forms the top level in the package hierarchy. You can add main packages to a structure package.
 - **MAIN PACKAGE:** A main package acts as a container for development objects that are semantically linked. A main package contains subpackages (of

either other main packages or standard development packages). Development objects are always added to the subpackage and never directly to the main package itself. You can group several main packages to form a structure package.

- DEVELOPMENT PACKAGE: Development objects are always added to a development package.
- ▶ PACKAGE ENCAPSULATED: Setting this flag encapsulates the package. When a package is encapsulated, only development elements exposed in package interfaces of the package are visible to the outside world. Package interfaces allow you to expose the development objects to other packages.

Package	ZDEMO
Short Description	Test Package
Application Component	BC-ABA
Software Component	HOME
Transport Layer	ZEC3
Superpackage	
Package Type	Development Package
Package encapsulated	<input type="checkbox"/>

Figure 2.13 Package Attributes Window

4. Once you've completed filling out the fields in [Figure 2.13](#), select the green checkmark to continue.
5. In the dialog box that appears, assign a transport request to save the package. You can either assign an existing transport request by clicking the OWN REQUESTS button or create a new request, as shown in [Figure 2.14](#). To create a new request, simply select the CREATE icon. This will open the CREATE REQUEST dialog box, where you can fill out the fields for your new request.

Tip

If you're creating something in the SAP system and the system prompts you to assign a package, it means you're creating a *repository object*; keep this in mind if you need to identify if what you're creating is a repository object or not. Remember that all repository objects (except for SAPscripts) are cross-client.

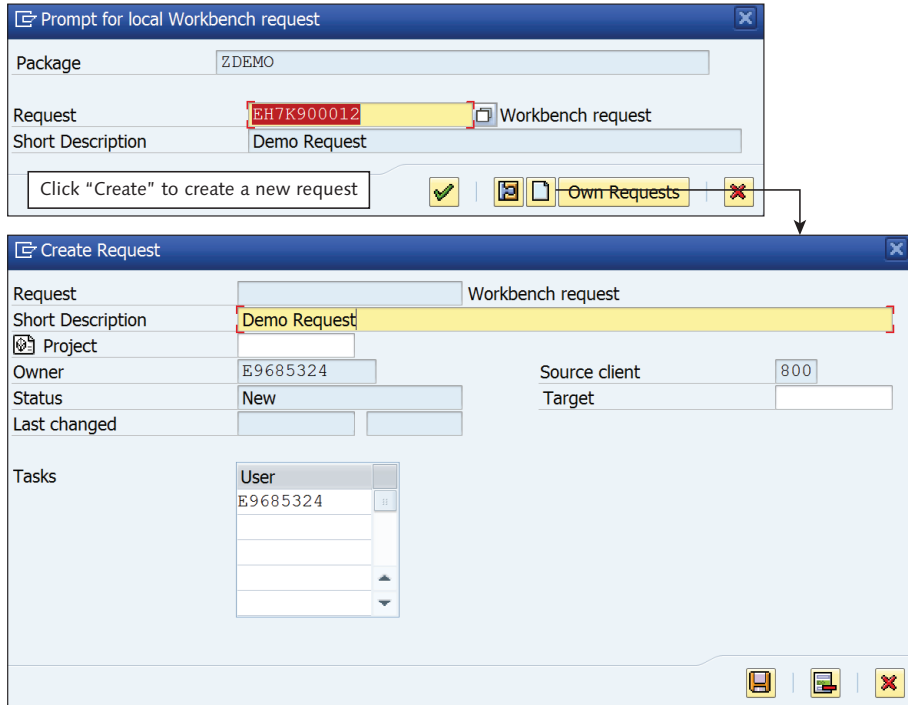


Figure 2.14 Creating a Request

2.4 Summary

In this chapter, we explored the technical aspects of an SAP system. Even though SAP Basis administrators perform most of the setup and configuration for the components discussed in this chapter, having a basic understanding of these components helps when developing ABAP programs.

We discussed the three layers of an SAP system—namely, the presentation layer, application layer, and database layer. We also explored the different components that make up these systems, such as different SAP GUI versions and the components of the application server and database server.

We also looked into the client concept and discussed the data structure of an SAP system and how the data can be made specific to a client or available across clients.

Finally, we provided an overview of repository objects and how they're grouped into packages. This overview lays the foundation to discuss development objects.

In the next chapter, we'll begin to introduce you to the ABAP environment and the ABAP Workbench tools.

Now that you've walked through the SAP system, let's turn our attention to the ABAP environment. In this chapter, we'll introduce native IDE and the Eclipse IDE plug-in.

3 Introduction to the ABAP Environment

SAP provides an integrated development environment (IDE) called *ABAP Workbench*, a collection of tools used to develop, test, and run ABAP programs. After the release of SAP NetWeaver 7.31 SPS 4, SAP introduced the *ABAP Development Tools* (ADT) for SAP NetWeaver, also known as *ABAP in Eclipse*, which adds Eclipse capabilities to ABAP Workbench (Transaction SE80) functionality.

In this chapter, we will showcase SAP GUI and the Eclipse IDE and introduce the ABAP Workbench tools that we'll use to develop various ABAP programs throughout this book.

In [Section 3.1](#), we'll begin by looking at the SAP environment in general, including information on the ABAP programming environment and steps for logging onto the SAP system. We'll also look at the different elements of an SAP screen and how to use transaction codes to get from point A to point B.

[Section 3.2](#) will dig into the different ABAP tools found in ABAP Workbench. This includes the ABAP Editor, Function Builder, Class Builder, Screen Painter, Menu Painter, ABAP Data Dictionary, and Object Navigator. Each of these tools will be used and discussed throughout the remainder of the book, so it's important to gain a good understanding of their functions and purpose.

We'll end the chapter with an overview of the Eclipse IDE in [Section 3.3](#), including steps for installation and setup.

3.1 SAP Environment

As discussed in earlier chapters, you can use SAP GUI or a web browser (SAP Web GUI for traditional applications or for Web Dynpro applications) to logon to an SAP system and run applications. However, ABAP Workbench tools are fully supported in a desktop environment for both SAP GUI and Eclipse IDE. To avoid confusion and to follow a consistent theme throughout the book, we will be using SAP GUI for Windows in this book for all examples. You should be able to use SAP GUI for Java or Eclipse IDE to follow the examples without any difficulty.

In this section, we'll explain how to logon to an SAP system and navigate through the SAP screens. We'll explore the different ways to run SAP applications and will look at various elements of an SAP screen.

3.1.1 ABAP Programming Environment

Unlike in other programming languages (Java, C/C++, etc.), ABAP source code is not stored in files; instead, all ABAP source code is saved in the underlying database of the SAP instance. There is no separate compiler for ABAP code; the ABAP runtime environment itself interprets and executes ABAP code.

The database interface in the work process communicates with the underlying database. The database interface converts Open SQL statements in the ABAP program to Native SQL statements (native to the underlying database). The ABAP runtime system ensures that the ABAP programs are operating system- and database-independent, making ABAP programs portable. In other words, you don't have to worry about changing your program code if the underlying database or operating system environment changes.

In the next few sections, we'll explore these environments and their on-screen elements in greater detail.

3.1.2 Logging On to the SAP Environment

The SAP logon screen shows four input fields, as shown in [Figure 3.1](#). Along with username (USER) and PASSWORD, you also enter the ID of the CLIENT you wish to logon onto. SAP supports multiple languages, so you can choose a LOGON LANGUAGE from any of the installed languages in your SAP system.

To use a specific language, input its two-digit SAP language code on the logon screen—for example, “EN” for English, “DE” for German, and so on. The logon screen also includes a NEW PASSWORD button to change your password.

Figure 3.1 SAP Logon Screen

Once you fill in the details on the logon screen, press to log on to the system.

3.1.3 Elements of the SAP Screen

Upon successful logon, you'll see the default SAP EASY ACCESS screen (see [Figure 3.2](#)); this entry screen is always displayed after logon and is used for opening various SAP applications. The SAP EASY ACCESS screen includes a customizable SAP menu with a tree structure in which all the installed applications are organized into various folders.

The navigation area of the SAP EASY ACCESS screen is divided into the SAP MENU, USER MENU, and FAVORITES:

- ▶ **SAP MENU**
The SAP MENU is delivered standard with SAP.
- ▶ **USER MENU**
The USER MENU can be customized to display user-specific applications based on the user's role in the organization. For example, a finance user might see finance-related applications and a sales user may see sales-related applications by default in the SAP MENU.

► FAVORITES

Frequently accessed applications can be added as favorites under the FAVORITES folder for quick access.

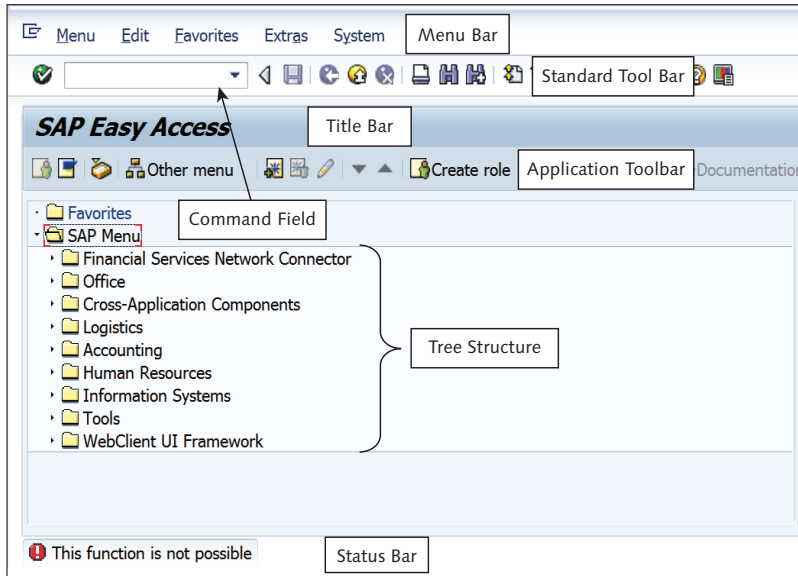


Figure 3.2 SAP Easy Access Screen with Screen Elements

Figure 3.2 shows the various screen elements of a standard SAP screen as accessed by the SAP GUI.

Now, let's walk through each of the standard screen elements identified in Figure 3.2.

Menu Bar

The menu bar is where various menu items for applications are located. You can use the menu bar to carry out additional activities with the application. When developing your own ABAP applications, you can design the menu bar by using the Menu Painter tool (Transaction SE41). You'll learn more about Menu Painter in Section 3.2.5 and how to use it in Chapter 12.

The menu bar consists of an application-specific menu, system menu, and help menu. The application-specific menu consists of menu items that are specific to

each application and are created when developing the application. For example, if you want the user to download report output to a spreadsheet, you can add a menu item and related functionality to do so. The system menu and help menu are available by default and consist of standard system functions and links to help resources, respectively.

Standard Toolbar

The standard toolbar contains a standard set of icons for every application. The standard icons include SAVE, BACK, EXIT, CANCEL, PRINT, FIND, FIND NEXT, and so on.

You can't remove icons from or add icons to this toolbar. However, you can enable or disable certain icons for your application according to your business requirements. The standard toolbar is customized via the Menu Painter tool during application development.

Title Bar

The title bar displays the title of the application, allowing you to identify the application you're running easily. Based on the type of program, the title is displayed either from the program attributes or from your program code, using the ABAP keyword `SET TITLEBAR`.

You'll learn more about setting the application title and using the ABAP keyword `SET TITLEBAR` in [Chapter 12](#).

Application Toolbar

The application toolbar displays all the application-specific icons. The application toolbar is located directly below the title bar. This area can be used to provide various icons that enhance the functionality of the application.

It's standard design practice to use an application toolbar to duplicate functions from the menu bar. This allows for quick access to some of the most commonly used functions for the current screen and system task. The buttons available in the application toolbar change from application to application as per each application's design.

You can display any of the available icons in the application toolbar by using the Menu Painter tool. To check the possible icons that can be displayed in the application toolbar, see table [ICON](#). For example, you can display the PDF icon in the application toolbar of your report output to allow for downloading the report output quickly as a PDF file.

Status Bar

The status bar displays all status messages. This area is used to show various messages to the users while they run your application. For example, you can display an error message when an invalid entry is made in the input field of the screen, or you can show a status message to give the user a status update while your application runs.

The status bar also displays other information, such as the server you're logged into, the client database you're accessing, and the user ID with which you're logged in.

You'll learn more about the types of messages that can be displayed for the user in [Chapter 6](#).

Command Field

The command field can be used to open applications (programs) quickly by using their transaction codes. Applications (also known as *transactions*) can be associated with a transaction code, and a user can directly enter the transaction code in the command field to run the application.

Certain commands, such as `/n` to terminate the application, `/h` to turn on the debugging, and others, can be entered in the command field to carry out different actions. [Table 3.1](#) shows some of the popular commands. In the next section, we'll look specifically at the transaction codes typically used in ABAP development.

3.1.4 Transaction Codes

To make it easy to open an application, each SAP application is associated with a transaction code. Applications are linked to a transaction code in Transaction SE93 (Maintain Transaction). Once a program is associated with a transaction

code, the user can directly input the transaction code in the command field to run the program. Letters A to X are reserved for standard SAP applications; transaction codes for custom applications should be created in the customer namespace Y or Z.

SAP follows a standard naming convention for most of the standard SAP applications. Transaction codes ending with a 1 generally are used for creation, transaction codes ending with a 2 are for editing, and transaction codes ending with 3 are for displaying. The following are some example transaction codes that hold to this pattern:

- ▶ MM01 (Material Master Create)
- ▶ MM02 (Material Master Change)
- ▶ MM03 (Material Master Display)
- ▶ VF01 (Billing Document Create)
- ▶ VF02 (Billing Document Change)
- ▶ VF03 (Billing Document Display)
- ▶ ME21N (Purchase Order Create)
- ▶ ME22N (Purchase Order Change)
- ▶ ME23N (Purchase Order Display)

Transaction codes ending with *N* are called *enjoy transactions*. Enjoy transactions were released with the SAP ERP release, which was also known as the *Enjoy* release. These transactions implement the Control Framework elements, such as docking containers, custom containers, ALV grids, and picture controls. These transactions replaced older transactions. For example, Transaction ME21N is an SAP enjoy transaction for Transaction ME21.

Next, we'll explain how transactions can be used to navigate within the system and how to open them.

3.1.5 Navigating and Opening Transactions

There are couple of ways to open a transaction (application). You can use the tree structure in the SAP MENU on the SAP EASY ACCESS screen, or you can input the transaction code in the command field directly.

Each SAP window that you open is called a *session*, and each application runs in a session. Each session occupies memory in the application server, so to conserve system memory the SAP Basis administrator can restrict the maximum sessions that are allowed per user login. The maximum allowed is sixteen sessions per login, but the limit is usually set to six sessions per login.

Various transactions are organized into different folders in the SAP MENU on the SAP EASY ACCESS screen (see [Figure 3.3](#)).

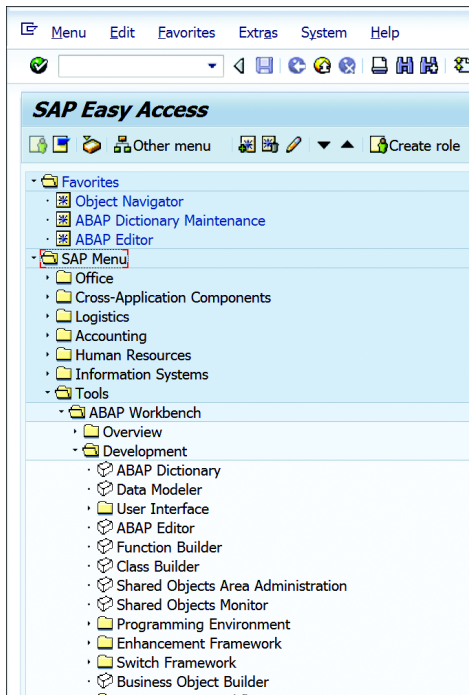


Figure 3.3 SAP Menu on SAP Easy Access Screen

The menu shown in [Figure 3.3](#) can be customized based on a user's role in the organization. System administrators usually assign role-specific menus to users while setting up user accounts in the SAP system. For example, a sales executive might see only sales-related transactions in his SAP menu.

A user can also add frequently used transactions to the FAVORITES folder for easy access. For example, as an ABAP consultant, you often use the ABAP Editor. To access the ABAP Editor, you can navigate to the menu path SAP MENU • TOOLS •

ABAP WORKBENCH • DEVELOPMENT and double-click ABAP EDITOR (see [Figure 3.3](#)). This can be time-consuming each time you want to open the ABAP Editor. Instead, you can simply add ABAP EDITOR to your favorites by right-clicking the FAVORITES folder in the SAP EASY ACCESS screen and selecting INSERT TRANSACTION from the context menu (see [Figure 3.4](#)).

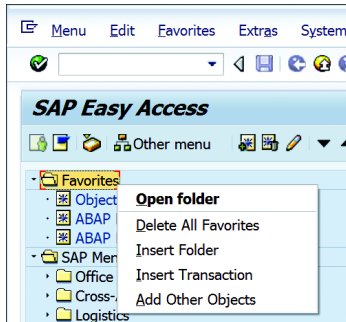


Figure 3.4 Adding Transactions to Favorites

Another, faster way to open transactions is to enter the transaction code directly into the command field. This is useful when you want to open another transaction from any session without having to open the SAP EASY ACCESS screen.

If you're on the SAP EASY ACCESS screen, you can directly input the transaction code in the command field and then press `[Enter]` or click the green checkmark to the left of command field. However, if you've already opened a transaction and want to open a new transaction from the existing transaction screen, then you should preface the transaction code with `/N` or `/O`. Entering the transaction code alone will not open the application unless you're on the SAP EASY ACCESS screen.

`/N` and `/O` are used for the following purposes:

- ▶ **`/N`**
Preface the transaction code with `/N` to terminate the current transaction and open the new transaction in the same session.
- ▶ **`/O`**
Preface the transaction code with `/O` to keep the current session intact and open the new transaction in a new session.

For example, on the SAP EASY ACCESS screen, you can enter the transaction code "SE38" to open the ABAP Editor. This will terminate the SAP EASY ACCESS screen and open the ABAP Editor in the same session.

Now, if you want to open Transaction SE11 (ABAP Data Dictionary) from the ABAP Editor, enter "/NSE11" or "/OSE11" in the command field, depending on whether you want to terminate ABAP Editor and open ABAP Data Dictionary in the same session or keep ABAP Editor intact and open ABAP Data Dictionary in a new session.

Table 3.1 lists some of the common command field formats that can be used. Please note the list is not exhaustive.

Common Command Field Options	Description
/nxxxx	Call a transaction in the same session (window)
/*xxxx	Call transaction in the same session (window); the initial screen is skipped
/oxxxx	Call transaction in an additional session
/n	To end the current transaction (<i>caution</i> : unsaved changes are lost without warning)
/i	Delete the current session
/o	Generate a session list
/ns000	End the current transaction and return to the starting menu
/nend	Log off from the system
/nex	Log off from the system without a confirmation prompt (<i>caution</i> : unsaved changes are lost without warning)
/h	Turn on debugging
Note: xxxx represents transaction code	

Table 3.1 Some Common Command Field Options

You can easily check the transaction code associated with each transaction by enabling a setting to display the technical name. On the SAP EASY ACCESS screen, follow the menu path EXTRAS • SETTINGS, or press **[Shift] + [F9]**, and select the DISPLAY TECHNICAL NAMES checkbox (see [Figure 3.5](#)).

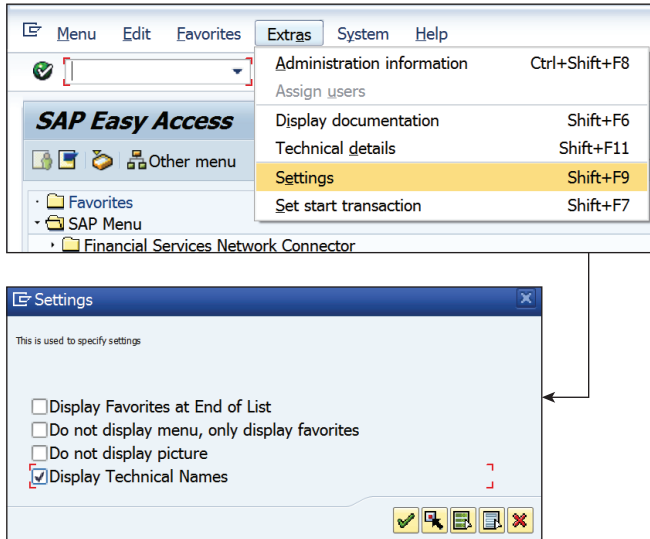


Figure 3.5 Enabling Technical Names

After this setting is enabled, when you open the tree structure in SAP MENU, you should see the transaction code displayed for each application (see [Figure 3.6](#)).

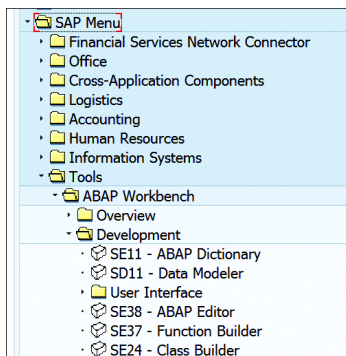


Figure 3.6 Transaction Code Displayed for Each Application

3.2 ABAP Workbench Overview

As an ABAP developer, you'll use the ABAP Workbench tools to work with various development objects, such as programs, functions, classes, tables, and so on.

In this section, we'll explore some of the important ABAP Workbench tools that you'll use to develop ABAP programs. We'll explore other ABAP Workbench tools as we progress through this book.

ABAP Workbench tools are integrated with development objects, and the ABAP Workbench provides forward navigation support to move between objects. For example, if you're editing the program code in the ABAP Editor and come across a statement in which a function module is called, simply double-clicking the function module name will open the function module in the Function Builder tool.

Because you use different ABAP Workbench tools to design and develop various objects in an ABAP application, instead of juggling between ABAP Workbench tools in different screens, SAP provides an application called *Object Navigator* (Transaction SE80).

Object Navigator organizes all the development objects in a tree-like hierarchy, which makes it easy to edit the application's different objects. We'll look more closely at the structure of the Object Navigator screen in [Section 3.2.7](#).

Let's explore the ABAP Workbench tools and see how they're integrated into the Object Navigator to provide an IDE.

3.2.1 ABAP Editor

One of the main components of ABAP Workbench is ABAP Editor, in which you write and edit ABAP code. The main transaction to open ABAP Editor is Transaction SE38, but the editor also is integrated into other ABAP Workbench tools, like Class Builder, Function Builder, and Screen Painter.

In ABAP Editor, you can choose one of three different modes:

- ▶ **FRONT-END EDITOR (NEW) (source code mode)**
The new front-end editor is a universal programming tool optimized for ABAP since the release SAP NetWeaver 7.0 and is available with SAP GUI for Windows 6.40 patch level 10 or higher. It is fully integrated with the SAP environment and provides various features beyond those in the old editor, such as code completion, code templates, changing the color of ABAP keywords and the background color, auto-filling parameter interfaces for procedures, and so on.
- ▶ **FRONT-END EDITOR (OLD) (plain text mode)**
The old front-end editor works in plain text mode and provides a basic feature

set. When using this mode, the source code is loaded into the frontend and edited locally. Since the source code is edited locally, all the functions that do not require interaction with backend are performed faster. This editor is available in SAP GUI for Windows and SAP GUI for Java.

► **BACK-END EDITOR (line-based mode)**

The back-end editor works in a line-based mode in which only one line at a time is edited. Even though this editor is not very user-friendly, it can be used in any operating system environment (unlike the front-end editor).

You can select a mode by following the menu path **UTILITIES • SETTINGS** from any ABAP Workbench tool to open the **USER-SPECIFIC SETTINGS** dialog box (see [Figure 3.7](#)).

On this screen, select the **ABAP EDITOR** tab and then the **EDITOR** tab. Under the **EDITOR** tab, select the radio button for your preferred editor.

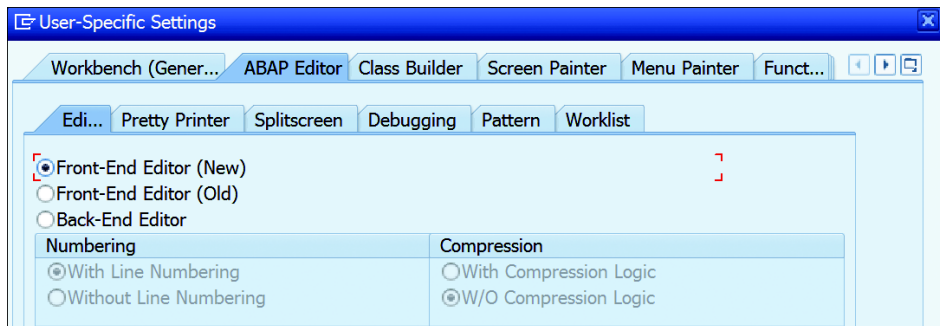


Figure 3.7 ABAP Editor Settings

We recommend using the new front-end editor; it offers many powerful features for writing and editing ABAP code, such as syntax highlighting, code hints, and auto-completion. [Figure 3.8](#) highlights some of the features of the new editor that are not available in other editor modes.

As shown in [Figure 3.8](#), the new front-end editor supports bookmarking for fast navigation, highlights the changes to the source code when editing, provides code hints and auto-completion of language elements and structures, and displays element blocks that easily can be expanded or collapsed for legibility.

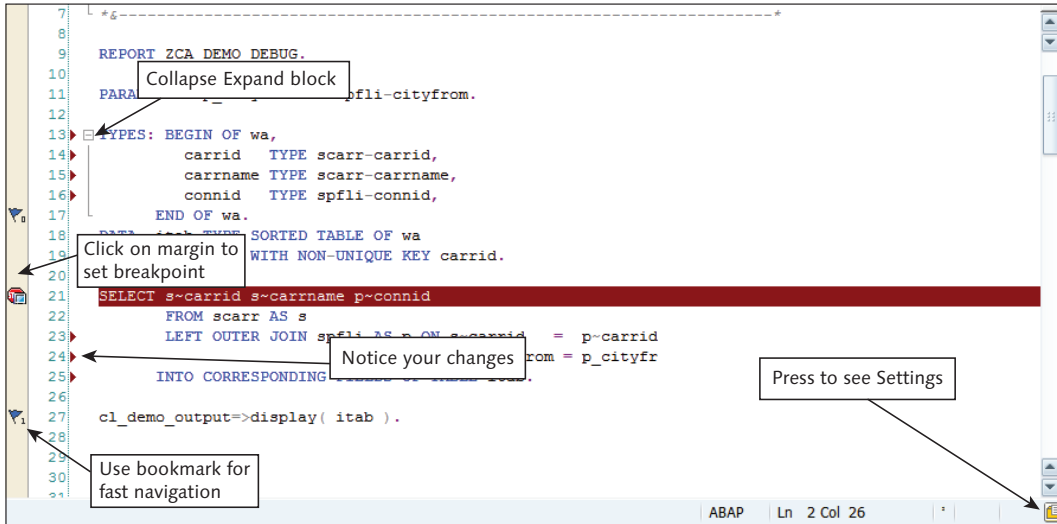


Figure 3.8 New Editor Mode Features

You can also quickly set a breakpoint by double-clicking the margin line. From within the settings, you can create user-defined source code templates to speed up the development process and avoid manually writing repetitive code.

3.2.2 Function Builder

Function modules are reusable software blocks that can be called from ABAP programs. The Function Builder is the ABAP Workbench tool that allows you to create and maintain function modules globally in the system. These function modules can be called by any ABAP program or remotely by external programs.

Function modules are called in ABAP programs or called remotely by external programs and never executed by the user directly. However, Function Builder provides a test environment to test the function modules without actually calling them in a program.

Function Builder also serves as a central library to search for existing function modules to use in your programs. You can also maintain documentation for each function module during development in the Function Builder, allowing other

developers to understand the functionality and parameter interface of the function module when they use the module in their own programs.

Access Function Builder via Transaction SE37 or from the Repository Browser (Transaction SE80). [Figure 3.9](#) shows Function Builder.

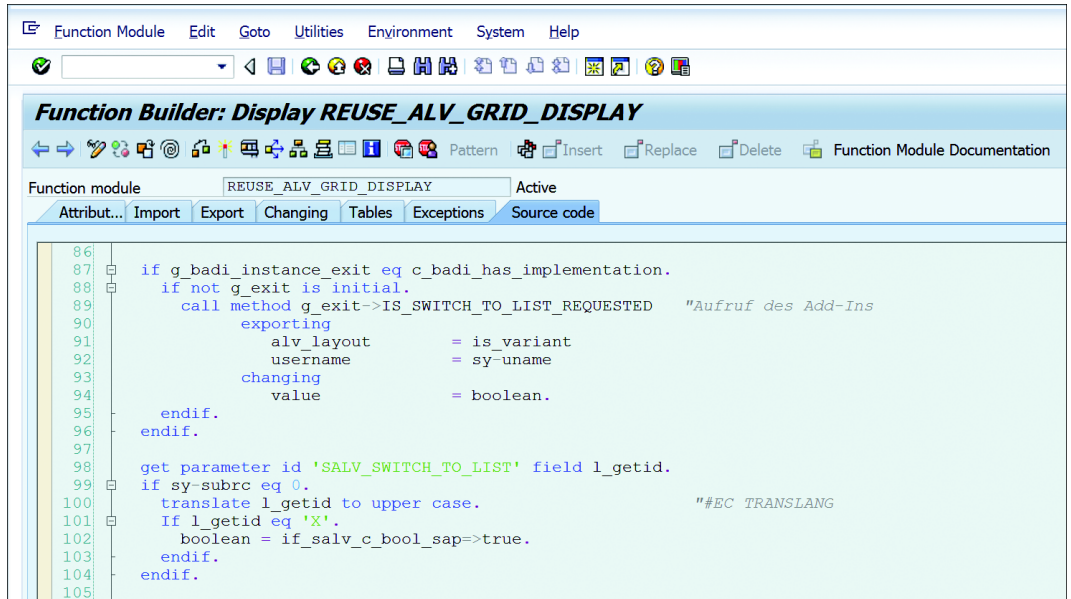


Figure 3.9 Function Builder

We'll explain Function Builder in more detail and you'll use it to develop function modules in [Chapter 7](#).

3.2.3 Class Builder

Similar to how Function Builder is used to create and maintain function modules, Class Builder is an ABAP Workbench tool used to create, define, change, and test global ABAP classes and interfaces. ABAP classes are procedures that implement object-oriented programming concepts (see [Chapter 8](#)).

ABAP classes can be defined locally in an ABAP program or globally in the system using the Class Builder tool. Class Builder allows you not only to define and maintain global classes but to maintain relations such as inheritance between objects,

redefine methods in subclasses, and maintain attributes, methods, and events for the class. Class Builder also provides the runtime to test classes and interfaces.

Access Class Builder via Transaction SE24 or from the Repository Browser (Transaction SE80). [Figure 3.10](#) shows Class Builder. As you can see, the Class Builder includes multiple tabs to maintain various components of the class, such as methods, attributes, interfaces, and more.

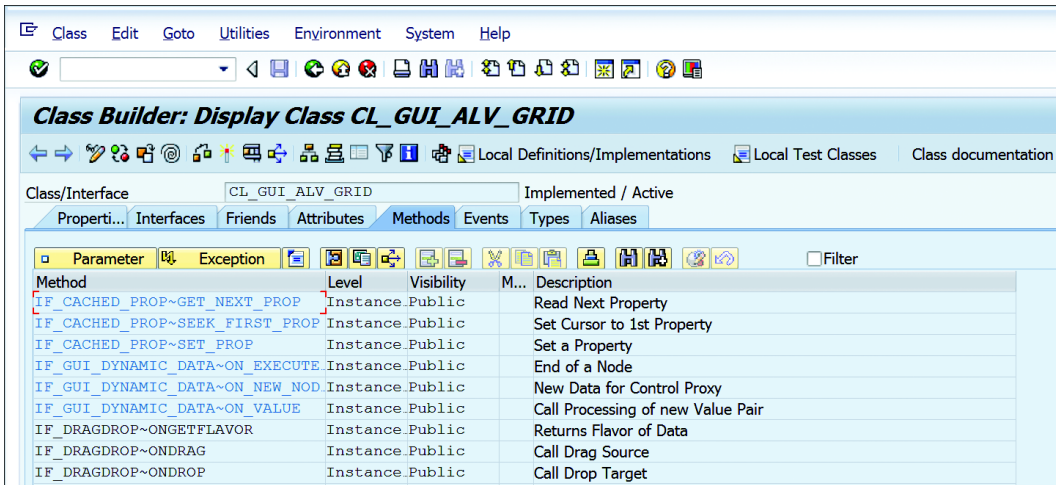


Figure 3.10 Class Builder

We'll explain Class Builder in more detail and you'll use it to implement global classes in [Chapter 7](#).

3.2.4 Screen Painter

Screen Painter is an ABAP Workbench tool used to create screens for ABAP programs and to maintain the screen flow logic. In ABAP, three types of screen can be defined for ABAP programs: selection screens, list screens, and general screens. Selection screens and list screens are automatically generated using specific ABAP keywords. General screens are designed and maintained using Screen Painter.

Screen Painter provides a graphical layout editor mode in SAP GUI for Windows, in which various screen elements, like input fields, text fields, and dropdown list boxes, can be defined easily via drag-and-drop functionality.

For operating systems that don't support the graphical layout editor mode (e.g., SAP GUI for Java), an alphanumeric mode is also available to work with screen elements.

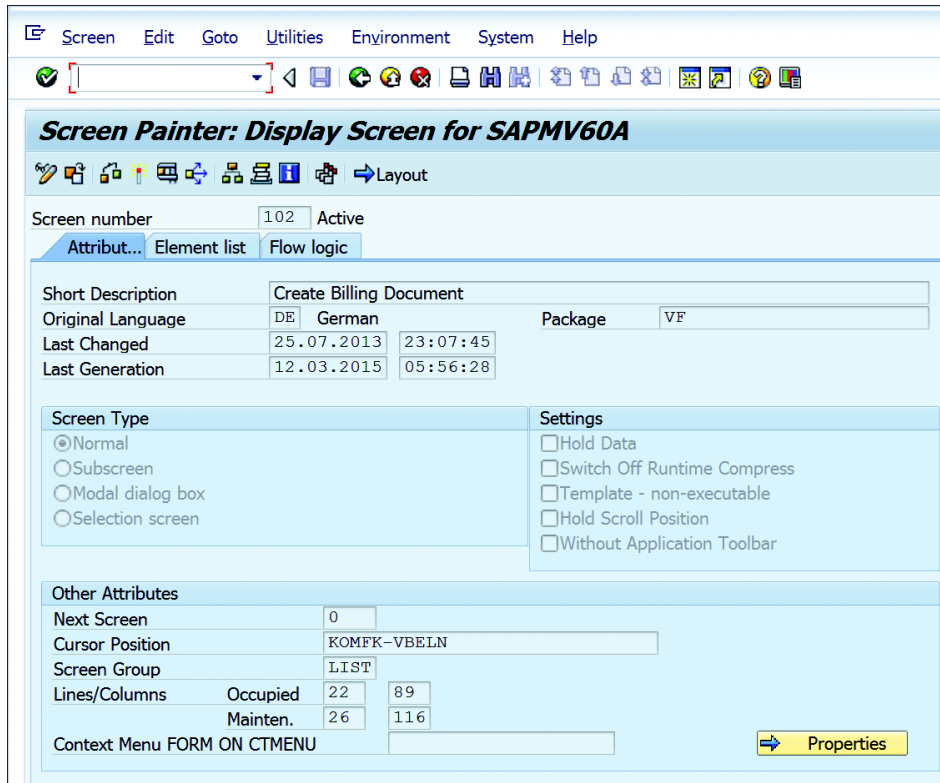


Figure 3.11 Screen Painter

As shown in [Figure 3.11](#), Screen Painter includes three tabs:

► ATTRIBUTES

In the ATTRIBUTES tab, you can adjust the screen attributes, such as the description, type of screen, and default behavior. There are three types of screens: the normal screen, the subscreen, and the modal dialog box. A *normal screen* is shown in the SAP GUI window, a *subscreen* is shown in a specific area within the normal screen window, and a *modal dialog box* is shown as a pop-up window. You create the required type of screen depending on your requirements under SCREEN TYPE in the ATTRIBUTES tab.

► ELEMENT LIST/SCREEN LAYOUT

Under the ELEMENTS LIST tab, you maintain the screen elements and the layout. You can maintain a variety of screen elements, such as input/output fields, text fields, radio buttons, checkboxes, and push buttons. You can also maintain more complex screen elements, such as tab strip controls and table controls. This tab is automatically filled with the screen element list when you design the screen using the graphical layout editor.

► FLOW LOGIC

In the FLOW LOGIC tab, you can maintain the flow logic of the screen. The runtime environment triggers the code in the flow logic based on the screen events. For example, a process before output (PBO) event is triggered before the screen is shown to the user, so any code that you want to execute before the screen is shown (such as disabling a field) can be handled by maintaining a PBO event in the flow logic of the screen.

The graphical layout editor is only available in SAP GUI for Windows. This editor is shown in [Figure 3.12](#).

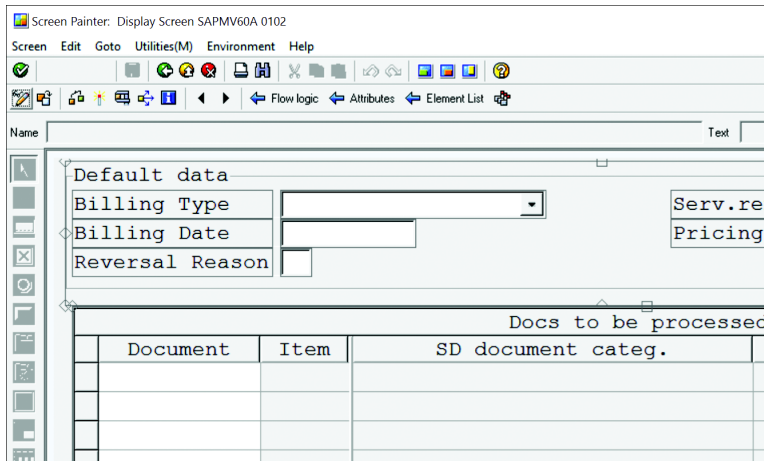


Figure 3.12 Layout Editor in Graphical Mode

Graphical Layout Editor

Screen Painter's graphical layout editor is part of SAP GUI and works only with the SAP NetWeaver Application Server 6.10 with SAP GUI release 6.10 or later. In SAP NetWeaver 4.0, it works with SAP GUI release 6.20 or later, patch 56, or release 6.40, patch 8.

3.2.5 Menu Painter

Menu Painter is an ABAP Workbench tool with which you can design user interfaces for ABAP programs. As discussed in [Section 3.1.3](#), various screen elements—such as items in the menu bar, the standard toolbar, and the application toolbar—can be designed using Menu Painter. For example, you can define the BACK, EXIT, and CANCEL buttons for your screen or define application-specific menu options in the menu bar and define corresponding icons in the application toolbar for quick access. These items are collectively called the *GUI statuses* of the screen.

[Figure 3.13](#) shows the Menu Painter screen on which various UI elements for a screen can be maintained. You can maintain the elements for the menu bar, standard toolbar, application toolbar, and function keys that can be used as keyboard shortcuts by the user.

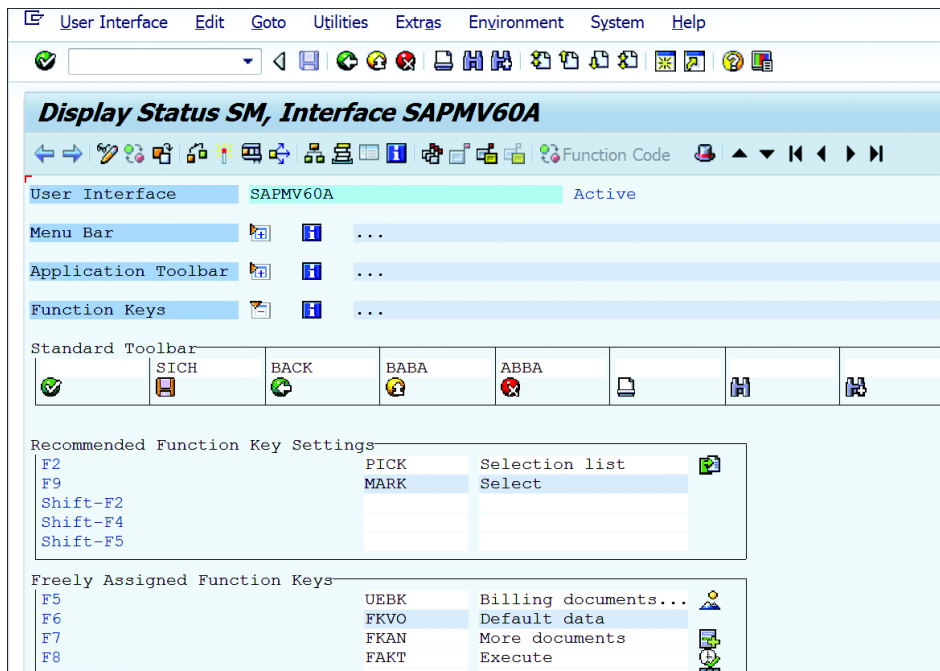


Figure 3.13 Menu Painter

We'll explore Menu Painter further when we discuss dialog programming in [Chapter 12](#).

3.2.6 ABAP Data Dictionary

ABAP Data Dictionary (see [Figure 3.14](#)) is a tool provided by SAP to centrally create and maintain data definitions. You can use ABAP Data Dictionary to create and maintain global user-defined types, like data elements, structures, and table types that can be used in any ABAP program.

You can use ABAP Data Dictionary to create objects (tables and views) in the underlying relational database, which facilitates central description of data without redundancy.

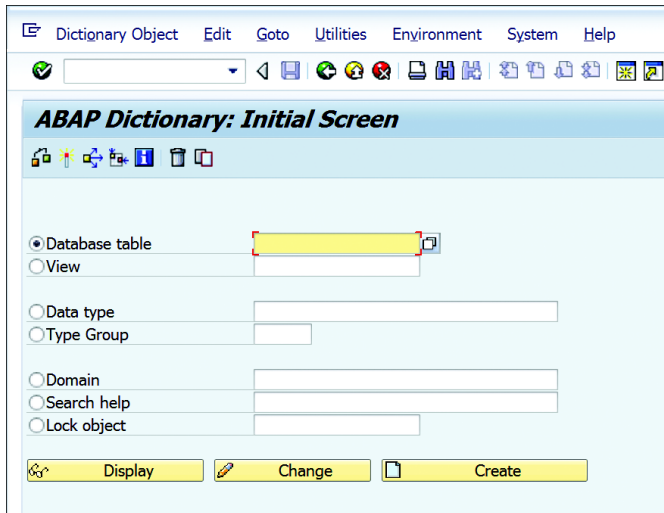


Figure 3.14 ABAP Data Dictionary: Initial Screen

In ABAP programs, Open SQL statements process data from the underlying relational database. Open SQL statements consist of the Data Manipulative Language (DML) part of SQL that helps perform certain actions, like reading, modifying, or updating data in database tables.

To perform certain actions, like creating or deleting database tables or creating table indexes, you use the Data Definition Language (DDL) part of SQL, which requires using Native SQL statements. However, you only use Open SQL statements in ABAP applications; it's highly recommended not to use Native SQL statements to define database tables or perform any actions on the underlying relation database.

Instead, use ABAP Data Dictionary to create and maintain tables and views or perform any actions that would otherwise require the use of Native SQL statements. This not only helps to avoid redundancy but also helps clarify how the logical structure of the objects used in application development are mapped to the underlying relational database.

All ABAP Data Dictionary objects are globally visible in the system and maintained centrally. For example, if many programs are using a particular data element and the definition of the data element needs to be changed due to a change in business requirements, the change will be automatically reflected for all the programs using the data element. Therefore, no program needs to be manually updated, because the runtime system automatically adjusts to the changes in ABAP Data Dictionary and ensures that all the applications and tools always access current data.

The most important object types in ABAP Data Dictionary are as follows:

► **Tables**

Tables are created in the underlying relational database using definitions from ABAP Data Dictionary. When a table is activated in ABAP Data Dictionary, the system uses the structure of the ABAP Data Dictionary table to create a physical table in the underlying database with the same structure as the table in ABAP Data Dictionary.

► **Views**

Multiple tables can be combined to create a logical view. The structure of the view is created in ABAP Data Dictionary, and the view of the database is then created using this structure.

► **Data types**

You can create global data types in ABAP Data Dictionary to centrally maintain the definition of data objects that use these global types. When a data type is changed in ABAP Data Dictionary, ABAP Data Dictionary automatically reflects the change in all the programs using that data type.

► **Lock objects**

Lock objects provide logical locking functionality in ABAP applications to prevent more than one user from accessing or manipulating the same data at the same time. When you create a lock object in ABAP Data Dictionary, ABAP Data Dictionary generates a function module that can be used in ABAP programs to

implement the logical locking functionality to restrict simultaneous data access by multiple users.

► **Domains**

Domains are used to maintain the technical attributes (type and length) for fields. When multiple fields share the same technical attributes, you can maintain a domain centrally and assign the domain to each field that shares the same technical description.

A value range for a table or structure component can be defined by maintaining a value range in the domain when the component refers to this domain. Domains are always assigned to data elements.

3.2.7 Object Navigator

The Object Navigator provides an IDE in which all the ABAP Workbench tools can be used centrally. For example, when developing an ABAP application that requires defining screens, GUI statuses, and ABAP Data Dictionary objects, these respective tools can be opened in the Object Navigator itself without having to leave the screen to access individual tools.

The Object Navigator helps organize your development objects during development and makes it easy to edit all related objects in an application in one place.

As shown in [Figure 3.15](#), the Object Navigator includes two main areas:

► **Navigation area**

The navigation area displays all the development objects of the application. All the objects are organized into folders with a tree structure.

► **Tool area**

The tool area is where the development objects are opened. It automatically opens the respective ABAP Workbench tool to open the object. For example, when you create or edit a screen, the Object Navigator automatically opens the screen using Screen Painter in the tool area.

Now that you've seen a general overview of the objects in the ABAP Workbench, the final section in this chapter will look at the other IDE that can be used with ABAP: Eclipse.

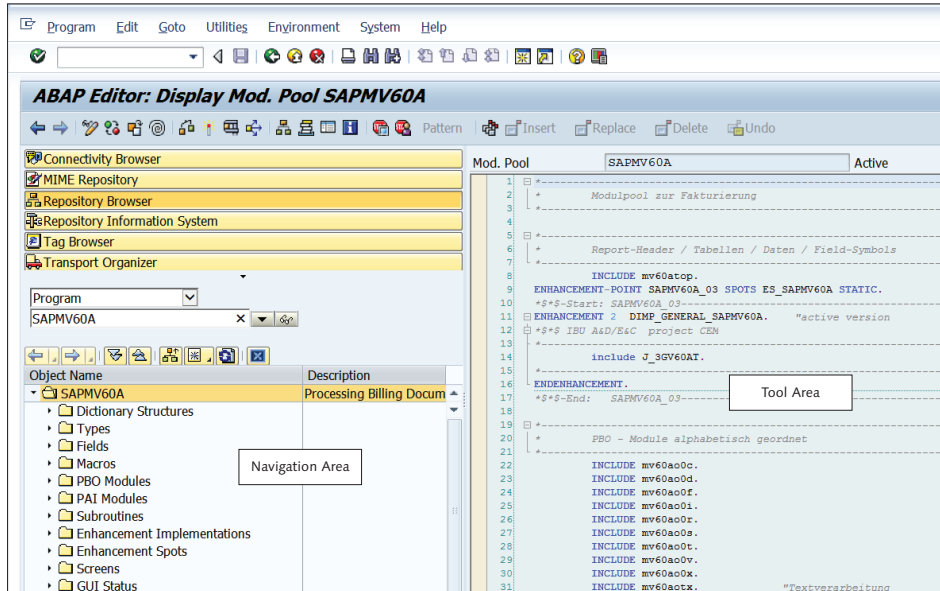


Figure 3.15 Object Navigator

3.3 Eclipse IDE Overview

SAP provides various development tools and functionalities for building and extending its business applications. With changing and evolving technologies, SAP has grown its portfolio of development tools.

Eclipse is a popular IDE that offers a range of benefits for Java application development. With SAP NetWeaver 7.31 SPS 4, SAP introduced ABAP Development Tools (ADT) for SAP NetWeaver, also known as *ABAP in Eclipse*. ADT provides the functionality of the ABAP Workbench (Transaction SE80) with Eclipse capabilities, aligning ABAP with SAP's other Eclipse-based development environments.

The learning curve involved is negligible, and if you're familiar with using the traditional ABAP Workbench tools in the SAP GUI environment, you should feel right at home with Eclipse.

The first step toward using ABAP in Eclipse is to set up a personal development environment, as opposed to the bundled environment you may be accustomed to using Object Navigator.

However, setting up Eclipse is as easy as installing the SAP GUI. Please note that not every tool has been migrated to the Eclipse platform yet; some tools will be added to future ADT releases. For some of the tools, SAP is evaluating new concepts that fit much better into its platform strategy.

For the tools that are not yet available for ADT, you can use the tight SAP GUI integration for now, which allows you to use existing transactions with context-aware navigation. For example, there are no native dynpro development tools within Eclipse, but you can use the integrated SAP GUI to run the standard ABAP Workbench tools like Screen Painter or Menu Painter within Eclipse.

Follow these steps to install and set up the Eclipse IDE:

1. Download the latest Eclipse release from <http://www.eclipse.org/>. Eclipse is available for Windows, Mac, and Linux platforms. The latest version available at the time of writing (October 2016) is Neon (see [Figure 3.16](#)). Make sure to check and download the latest version of Eclipse.

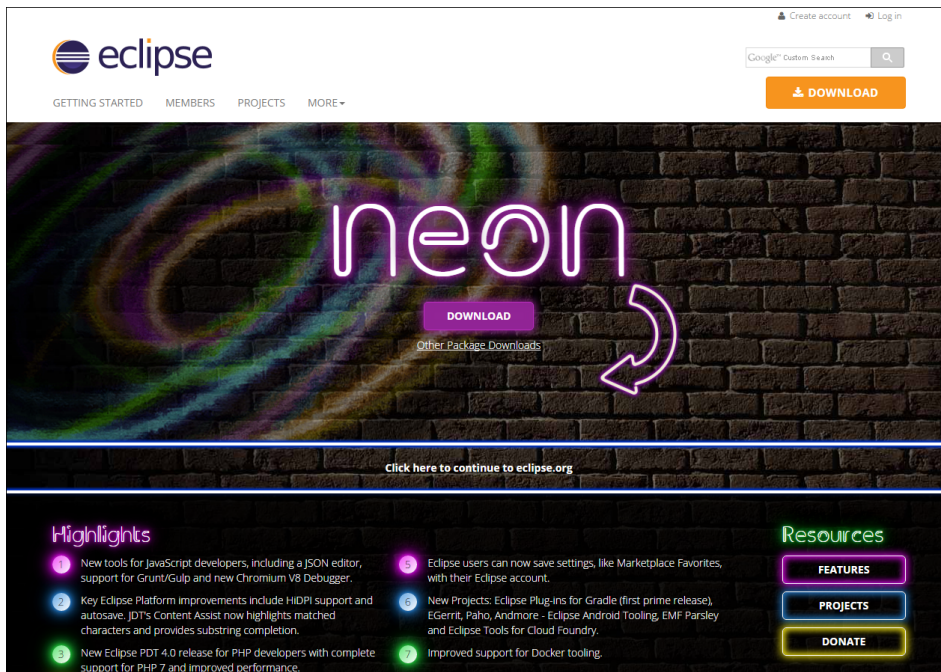


Figure 3.16 Eclipse Website

2. You can download the basic package or any of the predefined packages. Eclipse does not come with ABAP as a predefined package yet. Therefore, you need to download ADT separately from SAP's website. Go to <https://tools.hana.ondemand.com/#abap> for all the required information (see [Figure 3.17](#)). This webpage will provide information on which tools are available for the different Eclipse versions, therefore, your page may differ.

The screenshot shows the SAP Development Tools website. At the top, there is a navigation bar with links for HOME, ABAP, BW, CLOUD, GATEWAY, HANA, HCI, IDM, MOBILE, and SAPUI5. Below the navigation bar, there is a section titled "Eclipse Software Sites" with instructions on how to install tools. A table titled "Tools" lists various development tools and their availability for Eclipse Mars and Neon. The table has columns for "Tool", "Available for Eclipse", "Mars", and "Neon".

Tool		Available for Eclipse	
		Mars	Neon
ABAP	ABAP Development Tools for SAP NetWeaver	✓	✓
BW	Modeling Tools for SAP BW powered by SAP HANA	✓	✓
Cloud	SAP HANA Cloud Platform Tools		
	Latest SDK downloads:		
	• Java Web: neo-java-web-sdk-1.110.17.1.zip	✓	
	• Java EE 6 Web Profile: neo-javaee6-wp-sdk-2.92.5.2.zip		
	• Java Web Tomcat 7: neo-java-web-sdk-2.57.7.1.zip		
Gateway	SAP Mobile Platform Tools	✓	
HANA	SAP HANA Tools	✓	✓
HCI	SAP HANA Cloud Integration Tools	✓	
IdM	SAP Identity Management Configuration Lifecycle Tools	✓	✓
Mobile	SAP Mobile Tools		
SAPUI5	UI Development Toolkit for HTML5	✓	

Below the table, there is a "Troubleshooting" section with a link to "SAP Development Tools for Eclipse Installation, FAQ and Known Issues".

Figure 3.17 SAP Development Tools

3. To add ADT to Eclipse, open the installation wizard from within Eclipse via the menu path **HELP • INSTALL NEW SOFTWARE** (see [Figure 3.18](#)).
4. In the installation wizard, enter the URL in the **WORD WITH** field and click **ADD** to see a list of the available tools (see [Figure 3.19](#)). From the list, select **ABAP DEVELOPMENT TOOLS FOR SAP NETWEAVER** and click **NEXT** to continue.

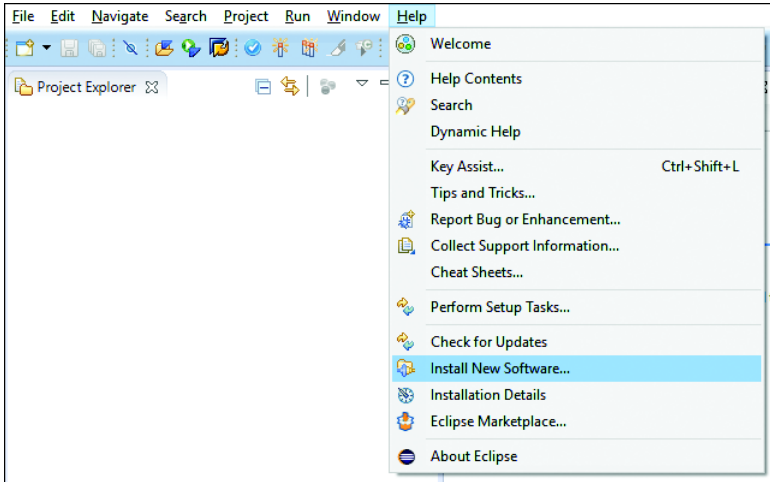


Figure 3.18 Installing ADT in Eclipse

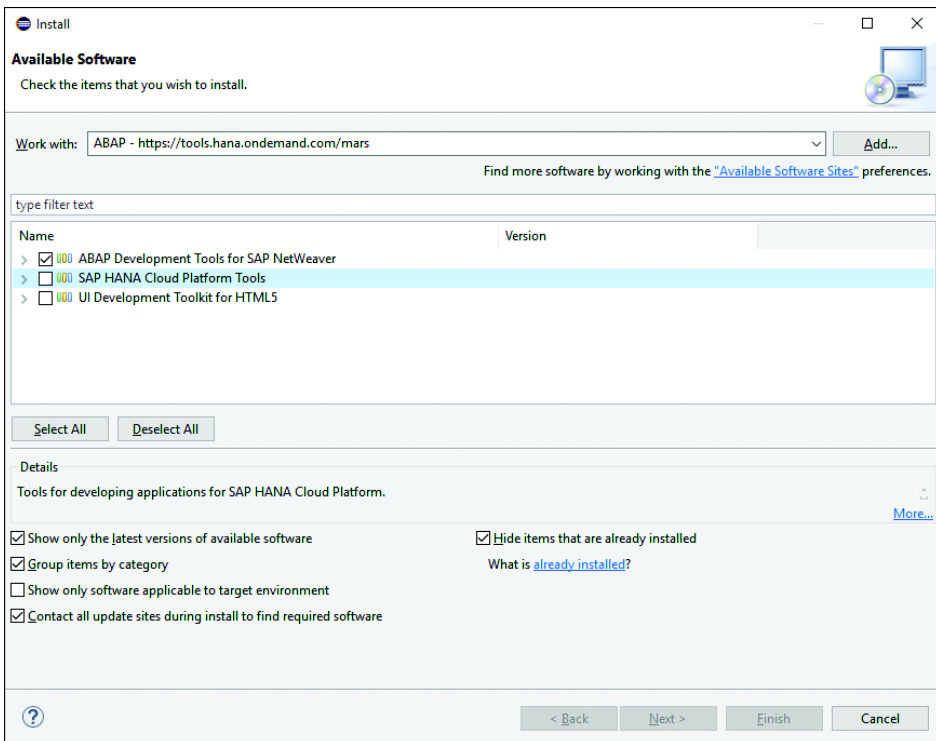


Figure 3.19 Eclipse Installation Wizard

5. Once the installation completes, you can create a new project to start working. From the menu bar, select **FILE • NEW • ABAP PROJECT**.
6. Now, you can load the logon parameters for your SAP system from the existing SAP GUI information or manually maintain the connection parameters. [Figure 3.20](#) shows the **SYSTEM CONNECTION** screen. Click **NEXT** to enter the SAP logon credentials to connect to your SAP system.

Note

Note that your backend SAP system requires certain configurations to be maintained to make a successful connection and work with ADT. SAP Basis administrators maintain these configurations, therefore, please contact your SAP Basis administrator if you need further information.

New ABAP Project

System Connection

Select a connection to an SAP system. You can maintain system connections in SAP Logon

Select connection from SAP Logon Define manually

Connection: **Browse...**

Properties of Connection

Description:

Connection Type:

Secure Network Communication (SNC):

Single Sign-On (SSO):

System ID:

Message Server:

Group:

Application Server:

SAProuter String:

Instance Number:

< Back **Next >** **Finish** **Cancel**

Figure 3.20 Connection Parameters Screen

7. When you create an ABAP project in Eclipse, a connection is established to an existing ABAP backend to gain access to the development objects (such as programs, ABAP Data Dictionary objects, and classes) maintained in the system's ABAP packages.
8. Once you've established a connection, you can use the Eclipse PROJECT EXPLORER (see [Figure 3.21](#)) to open the objects of the backend ABAP packages and add them to your project.

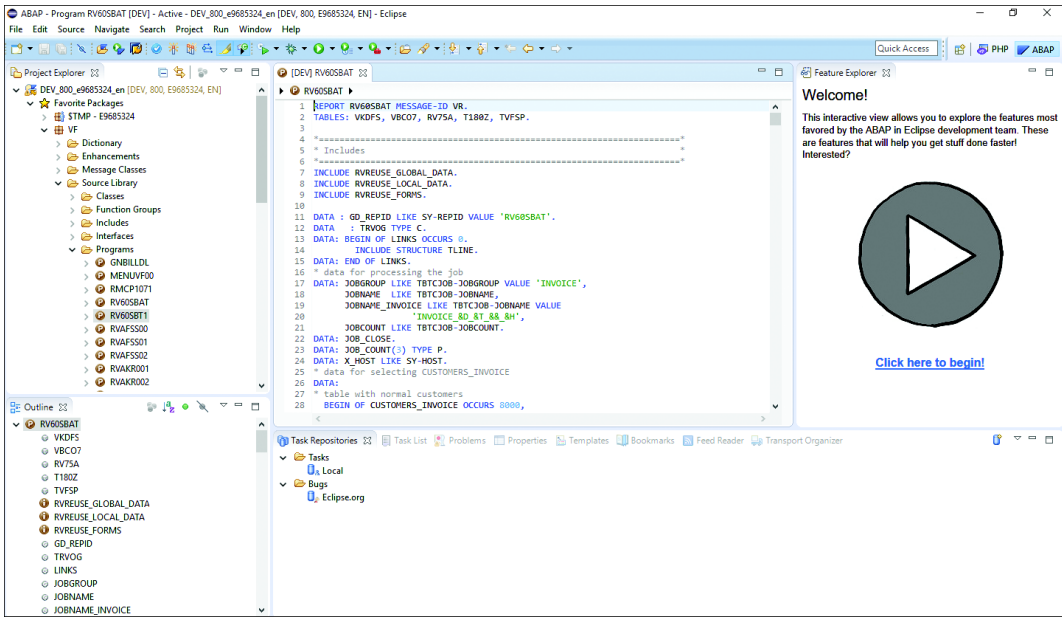


Figure 3.21 Project Explorer

You can now select a development object contained in an ABAP package and display it in a particular tool. For example, you can open a program in the source code editor if you want to modify the code.

The OUTLINE view, which is located beneath the PROJECT EXPLORER view, displays the structure of the selected development object, such as the program variables, and enables you to navigate directly to that element in the source code.

Eclipse IDE provides a quick start guide when you open the editor, which should help you get up to speed with Eclipse. You should find the Eclipse editor quite

familiar; most of the features are similar to the new front-end editor in the traditional SAP GUI that we previously discussed.

3.4 Summary

In this chapter, we looked at both the SAP environment and the ABAP environment. We walked through steps for logging on to the SAP system and then looked at the different elements found on a screen in SAP. We then looked at how to navigate the menu tree in SAP using transaction codes.

At that point, we shifted our focus to the two IDEs: SAP GUI and Eclipse. We explored the ABAP Workbench tools, including ABAP Editor, Function Builder, Class Builder, Screen Painter, Menu Painter, ABAP Data Dictionary, and Object Navigator. We'll use all the tools discussed in this chapter to develop ABAP objects and will have an opportunity to explore more about them when we discuss specific topics in this book.

In the final section of this chapter, we provided an overview of how to set up and use the Eclipse IDE. Many of the settings in Eclipse are similar to those in SAP GUI.

With this basic understanding of the SAP and ABAP environment and the various ABAP Workbench tools in place, in the next chapter we'll dive into basic ABAP programming concepts, such as ABAP syntax, data types, and data objects. At the end of the next chapter, you'll create your first ABAP program.

In this chapter, we'll look at basic ABAP programming language concepts. The concepts discussed in this chapter will lay the foundations to write your own ABAP programs.

4 ABAP Programming Concepts

ABAP programs process data from the database or an external source. As discussed in [Chapter 2](#), ABAP programs run in the application layer, and ABAP program statements can work only with locally available data in the program. External data, like user inputs on screens, data from a sequential file, or data from a database table, must always be transported to and saved in the program's local memory in order to be processed with ABAP statements.

In other words, before any data can be processed with an ABAP program, it must first be read from the source, stored locally in the program memory, and then accessed via ABAP statements; you can't work with the external data directly using ABAP statements. This temporary data that exists in an ABAP program while the program is executed is called *transient data* and is cleared from memory once the program execution ends. If you wish to access this data later, it must be stored persistently in the database; such stored data is called *persistent data*.

This chapter provides information on the basic building blocks of the ABAP programming language in order to understand the different ways to store and process data locally in ABAP programs. In [Section 4.1](#), we begin by looking at the general structure of an ABAP program, before diving into ABAP syntax rules and ABAP keywords in [Section 4.2](#) and [Section 4.3](#), respectively.

[Section 4.4](#) introduces the `TYPE` concept, and explores key elements such as data types and data objects, which define memory locations in ABAP programs to store data locally. In [Section 4.5](#), we discuss the different types of ABAP statements that can be employed, including declarative, modularization, control, call, and operational statements. Finally, in [Section 4.6](#), we conclude this chapter by walking through the steps to create your first ABAP program!

Initially, we'll use a procedural model for our examples to keep things simple; we'll switch to an object-oriented model after we discuss object-oriented programming in [Chapter 8](#).

4.1 General Program Structure

Any ABAP program can be broadly divided into two parts:

► Global declarations

In the global declaration area, global data for the program is defined; this data can then be accessed from anywhere in the program.

► Procedural

The procedural part of the program consists of various processing blocks such as dialog modules, event blocks, and procedures. The statements within these processing blocks can access the global data defined in the global declarations.

In this section, we'll look at both of these program structure parts.

4.1.1 Global Declarations

The global declaration part of an ABAP program uses declarative statements to define memory locations, which are called *data objects* and which store the work data of the program. ABAP statements work only with data available in the program as content for data objects, so it's imperative that data is stored in a data object before it's processed by an ABAP program.

Data objects are local to the program; they can be accessed via ABAP statements in the same program. The global declaration area exists at the top of the program (see [Figure 4.1](#)). We use the term *global* with respect to the program; data objects defined in this area are visible and valid throughout the entire ABAP program and can be accessed from anywhere in the source code using ABAP statements.

ABAP also uses *local declarations*, which can be accessed only within a subset of the program. We'll explore local declarations in [Chapter 7](#) when we discuss modularization techniques.

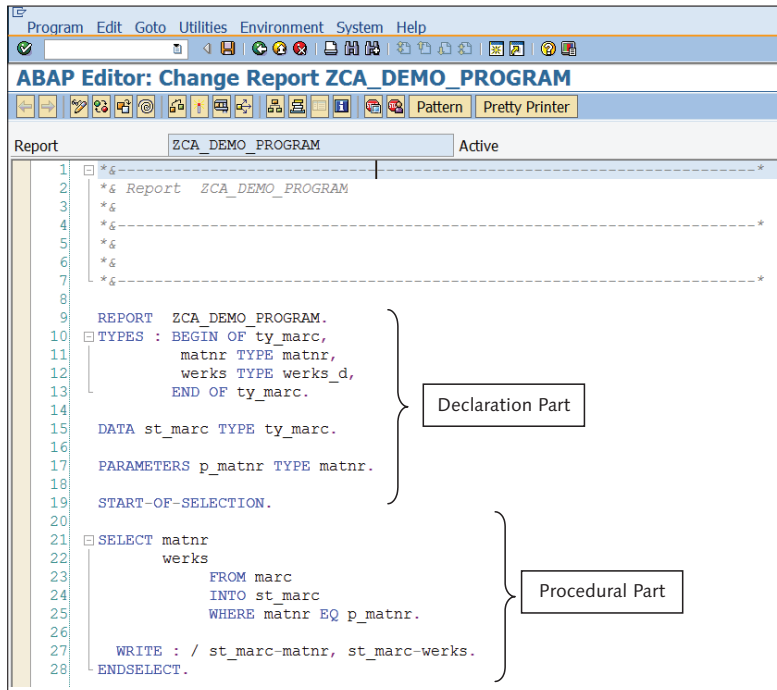


Figure 4.1 Program Structure

4.1.2 Procedural

After the declarative area is the procedural portion of the program. Here, the processing logic of the ABAP program is defined. In this section of the ABAP program, you typically use various ABAP statements to import and process data from an external source.

The procedural area is where the program logic is implemented. It uses various processing blocks that contain ABAP statements to implement business requirements. For example, in a typical report, the procedural area contains an event block to process the selection screen and validate user inputs on the selection screen and another event block to fetch the data from the database based on user input, and then calls a procedure to display the output to the user.

4.2 ABAP Syntax

The source code of an ABAP program is simply a collection of various ABAP statements, interpreted by the runtime environment to perform specific tasks. You use declarative statements to define data objects, modularization statements to define processing blocks, and database statements to work with the data in the database.

In this section, we'll look at the basic syntax rules that every ABAP programmer should know. We'll then look at the use of chained statements and comment lines.

4.2.1 Basic Syntax Rules

There are certain basic syntax rules that need to be followed while writing ABAP statements:

- ▶ An ABAP program is a collection of individual ABAP statements that exist within the program. Each ABAP statement is concluded with a period (".") and the first word of the statement is known as a keyword.
- ▶ An ABAP statement consists of operands, operators, or additions to keywords (see [Figure 4.2](#)). The first word of an ABAP statement is an ABAP keyword, the remaining can be operands, operators, or additions. *Operands* are the data objects, data types, procedures, and so on.
Various operators are available, such as assignment operators that associate the source and target fields of an assignment (e.g., = or ?=), arithmetic operators that assign two or more numeric operands with an arithmetic expression (e.g., +, -, *), relational operators that associate two operands with a logical expression (such as =, <, >), etc. Each ABAP keyword will have its own set of additions.
- ▶ Each word in the statement must be separated by at least one space.
- ▶ An ABAP statement ends with a period, and you can write a new statement on the same line or on a new line. A single ABAP statement can be extended over several lines.
- ▶ ABAP code is not case-sensitive.

In [Figure 4.2](#), the program shown consists of three ABAP statements written across three lines. The first word in each of these statements (REPORT, PARAMETERS,

and `WRITE`) is a keyword. As you can see, each statement begins with a keyword and ends with a period. Also, each ABAP word is separated by a space.

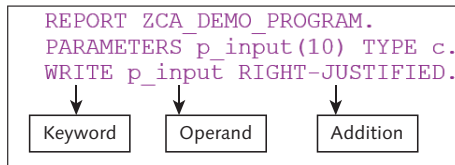


Figure 4.2 ABAP Statement

You can write multiple statements on one line or one statement can extend over multiple lines. Therefore, if you wish, you can rewrite the code in [Figure 4.2](#) as shown:

```
REPORT ZCA_DEMO_PROGRAM. PARAMETERS p_input(10) TYPE c. WRITE p_
input RIGHT-JUSTIFIED.
```

However, to keep the code legible, we recommend restricting your program to one statement per line. In some cases, it's recommended to break a single statement across multiple lines—for example:

```
SELECT * FROM mara INTO TABLE it_mara WHERE matnr EQ p_matnr.
```

The above statement may be written as shown in [Listing 4.1](#) to make it more legible.

```
SELECT * FROM mara
        INTO TABLE it_mara
        WHERE matnr EQ p_matnr.
```

Listing 4.1 Example of Splitting Statement across Multiple Lines

4.2.2 Chained Statements

If more than one statement starts with the same *keyword*, you can use a colon (`:`) as a chain operator and separate each statement with a comma. These are called *chained statements*, and they help you avoid repeating the same keyword on each line.

For example,

```
DATA v_name(20) TYPE c.
DATA v_age TYPE i.
```

Also can be written as

```
DATA : v_name(20) TYPE c,
      v_age TYPE i.
```

End the last statement in the chain with a period. Chained statements are not limited to keywords; you can put any identical first part of a chain of statements before the colon and write the remaining parts of the individual statements separated by comma—for example,

```
v_total = v_total + 1.
v_total = v_total + 2.
v_total = v_total + 3.
v_total = v_total + 4.
```

can be chained as

```
v_total = v_total + : 1, 2, 3, 4.
```

Note

ABAP code is not case-sensitive, so you can use either uppercase or lowercase to write ABAP statements. We recommend writing keywords and their additions in uppercase and using lowercase for other words in the statement to make the code more legible.

4.2.3 Comment Lines

To make your source code easy to understand for other programmers, you can add comments to it (see [Listing 4.2](#)). *Comment lines* are ignored by the system when the program is generated, and they're useful in many ways.

```
DATA f1 TYPE c LENGTH 2 VALUE 'T3'.
DATA f2 TYPE n LENGTH 2.
*This is a comment line
f2 = f1.
WRITE f2. "This is also a comment line
```

Listing 4.2 Comment lines

There are two ways to add comment lines in source code:

- ▶ You can enter an asterisk (*) at the beginning of a line to make the entire line a comment.
- ▶ You can enter a double quotation mark (") midline to make the part of the line after the quotation mark a comment this is called an *in-line comment*).

You can *comment* (i.e., set as a comment) on a block of lines at once (a *multiline comment*) by selecting the lines to be commented on and pressing `[Ctrl] + [<]` on the keyboard. Similarly, to *uncomment* (i.e., set as normal code) a block of lines, you can select the lines and press `[Ctrl] + [>]`.

Alternatively, you can also use the context menu to comment or uncomment code. To comment a line of code or a block lines, select the code, right-click, and select the appropriate option from the `FORMAT` context menu. This helps you avoid the tedious job of adding asterisks manually at the beginning of each line.

Now that you have a better understanding of basic ABAP syntax rules and chaining ABAP statements, in the next section, we'll look at the keywords used in ABAP.

4.3 ABAP Keywords

Because each ABAP statement starts with a keyword, writing ABAP statements is all about choosing the right keyword to perform the required task. Every keyword provides specific functionality and comes with its own set of *additions*, which allow you to extend the keyword's functionality.

For each keyword, SAP maintains extensive documentation, which serves as a guide to understanding the syntax to use with the keyword and the set of additions supported for the keyword.

You can access the keyword documentation by typing "ABAPDOCU" in the command bar to open it just like any other transaction or by placing the cursor on the keyword and pressing `[F1]` while writing your code in ABAP Editor. You can also visit https://help.sap.com/abapdocu_750/en/abenabap.htm, to access an online version of the ABAP keyword documentation (see [Figure 4.3](#)).

Because there are literally hundreds of keywords that can be used in ABAP, the best way to become familiar with the keywords is to explore them in relation to a requirement. We'll be taking this approach throughout the book as we introduce you to these keywords in various examples.

SAP NetWeaver AS ABAP Release 750, ©Copyright 2016 SAP AG. All rights reserved.

ABAP - Keyword Documentation

This documentation describes the syntax and meaning of the keywords of the ABAP language and its object-oriented part ABAP Objects. Alongside this, language frameworks and the associated system classes are also described.

- [ABAP - Overview](#)
An introduction to ABAP and the most important umbrella topics.
- [ABAP Dictionary](#)
A complete description of the most important objects for ABAP from ABAP Dictionary.
- [ABAP - Reference](#)
A complete description of all ABAP keywords in their relevant context.
- [ABAP - Quick Reference](#)
A short overview of all statements, ordered alphabetically.
- [ABAP - Release-Specific Changes](#)
List of all changes and enhancements made to ABAP since Release 3.0
- [ABAP - Programming Guidelines](#)
Rules and hints on using ABAP.
- [ABAP - Security Notes](#)
Overview of all potential security risks in ABAP programs.
- [ABAP - Glossary](#)
Terms in the ABAP environment, and their explanations.
- [ABAP - Index](#)
Alphabetical index of all language elements.
- [ABAP - Keyword Directory](#)
Alphabetical keyword directory for searching by topic.
- [ABAP - Examples](#)
Compilation of executable example programs.

This external version of the ABAP key word documentation only supports text links within the documentation itself and to external Web addresses, such as [SAP Help Portal](#). Other links, such as links to programs in the SAP system, are not active in this version.

Note

The programs and program extracts outlined in the ABAP keyword documentation are syntax examples and are not intended for use in a production system environment. The purpose of the source code examples is to enable a better explanation and visualization of the syntax and semantics of ABAP statements. SAP does not guarantee either the correctness or the completeness of the code. In addition, SAP takes no legal responsibility or liability for possible errors or their consequences, which occur through the use of the example programs.

The database tables of the flight data model used in the example programs can be filled using the program SAPBC_DATA_GENERATOR.

Figure 4.3 ABAP Keyword Documentation

4.4 Introduction to the TYPE Concept

An ABAP program only works with data inside data objects. The first thing you do when developing a program is declare data objects. Inside these data objects, you store the data to be processed in your ABAP program. You use declarative statements to define data objects that store data in the program; these statements are called *data declarations*.

Typically, you'll work with various kinds of data, such as a customer's name, phone number, or amount payable. Each type of data has specific characteristics. The customer's name consists of letters, a phone number consists of digits from 0 to 9, and the amount due to the customer will be a number with decimal values. Identifying the type and length of data that you plan to store in data objects is what the TYPE concept is all about.

In this section, we'll look at data types, domains, and data objects.

4.4.1 Data Types

Data types are templates that define data objects. A data type determines how the contents of a data object are interpreted by ABAP statements. Other than occupying some space to store administrative information, they don't occupy any memory space for work data in the program. Their purpose simply is to supply the technical attributes of a data object.

Data types can be broadly classified as elementary, complex, or reference types. In this chapter, we'll primarily explore the elementary data types and will only briefly cover complex types and reference types. More information on complex data types can be found in [Chapter 5](#), and more details about reference types are provided in [Chapter 8](#).

Elementary Data Types

Elementary data types specify the types of individual fields in an ABAP program. Elementary data types can be classified as predefined elementary types or user-defined elementary types. We'll look at these subtypes next.

Predefined Elementary Data Types

The SAP system comes built in with predefined elementary data types. These data types are predefined in the SAP NetWeaver AS ABAP kernel and are visible in all ABAP programs. You can use these predefined elementary data types to assign a *type* to your program data objects. You can also create your own data types (user-defined elementary data types) by referring to the predefined data types.

[Table 4.1](#) lists the available predefined elementary data types.

Data Type	Definition
i	Four-byte integer
int8	Eight-byte integer
f	Binary floating-point number
p	Packed number
decfloat16	Decimal floating-point number with sixteen decimal places
decfloat34	Decimal floating-point number with thirty-four decimal places

Table 4.1 Predefined Elementary Data Types

Data Type	Definition
c	Text field (alphanumeric characters)
d	Date field (<i>format: YYYYMMDD</i>)
n	Numeric text field (numeric characters 0 to 9)
t	Time field (<i>format: HHMMSS</i>)
x	Hexadecimal field

Table 4.1 Predefined Elementary Data Types (Cont.)

Predefined elementary data types can be classified as numeric or nonnumeric types. There are six predefined numeric elementary data types:

- ▶ 4-byte integer (*i*)
- ▶ 8-byte integer (*int8*)
- ▶ Binary floating-point number (*f*)
- ▶ Packed number (*p*)
- ▶ Decimal floating point number with 16 decimal places (*decfloat16*)
- ▶ Decimal floating point number with 34 decimal places (*decfloat34*)

There are five predefined nonnumeric elementary data types:

- ▶ Text field (*c*)
- ▶ Numeric character string (*n*)
- ▶ Date (*d*)
- ▶ Time (*t*)
- ▶ Hexadecimal (*x*)

The field length for data types *f*, *i*, *int8*, *decfloat16*, *decfloat34*, *d*, and *t* is fixed. In other words, you don't need to specify the length when you use these data types to declare data objects (or user-defined elementary data types) in your program. The field length determines the number of bytes that the data object occupies in memory. In types *c*, *n*, *x*, and *p*, the length is not part of the type definition. Instead, you define it when you declare the data object in your program.

Before we discuss the predefined elementary data types further, let's see how they're used to define a data object in the program.

The keyword used to define a data object is `DATA`. For the syntax, you provide a name for your data object and use the addition `TYPE` to refer it to a data type, from which the data object can derive its technical attributes.

For example, the line of code in [Figure 4.4](#) defines the data object `v_name` of `TYPE c` (character).

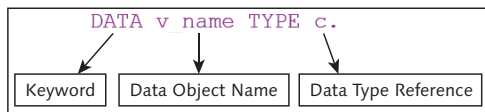


Figure 4.4 Declaring a Data Object

Notice that we did not specify the length in [Figure 4.4](#); therefore, the object will take the data type's initial length by default. Here, `v_name` will be a data object of `TYPE c` (text field) and `LENGTH 1` (default length). It can store only one alphanumeric character.

If you wish to have a different length for your data object, you need to specify the length while declaring the data object, either with parentheses or using the addition `LENGTH`:

```
DATA v_name(10) TYPE c.
DATA v_name TYPE c LENGTH 10.
```

In this example, `v_name` will be a data object of `TYPE c` and `LENGTH 10`. It can now store up to 10 alphanumeric characters. The Valid Field Length column in [Table 4.2](#) lists the maximum length that can be assigned to each data type.

For data types of fixed lengths, you don't need to specify the length, because it's part of the `TYPE` definition—for example:

```
DATA count TYPE i.
DATA date TYPE d.
```

[Table 4.2](#) lists the initial length of each data type, its valid length, and its initial value. The initial length is the default length that the data object occupies in memory if no length specification is provided while defining the data object, and the initial value of a data object is the value it stores when the memory is empty.

For example, as shown in [Table 4.2](#), a TYPE c data object will be filled with spaces initially, whereas a TYPE n data object would be filled with zeros.

Data Type	Initial Field Length (Bytes)	Valid Field Length (Bytes)	Initial Value
Numeric Types			
i	4	4	0
int8	8	8	0
f	8	8	0
p	8	1–16	0
decfloat16	8	8	0
decfloat34	16	16	0
Character Types			
c	1	1–65535	Space
d	8	8	'00000000'
n	1	1–65535	'0 ... 0'
t	6	6	'000000'
Hexadecimal Type			
x	1	1–65535	X'0 ... 0'

Table 4.2 Elementary Data Types: Technical Specifications

Predefined nonnumeric data types can be further classified as follows:

► **Character types**

Data types c, n, d, and t are character types. Data objects of these types are known as *character fields*. Each position in one of these fields can store one code character. For example, a data object of LENGTH 5 can store five characters and a data object of LENGTH 10 can store ten characters. Currently, ABAP only works with single-byte codes, such as American Standard Code for Information Interchange (ASCII) and Extended Binary Coded Decimal Interchange Code (EBCDI). As of release 6.10, SAP NetWeaver AS ABAP supports both Unicode and non-Unicode systems. However, support for non-Unicode systems has been withdrawn from SAP NetWeaver 7.5.

Single-byte code refers to character encodings that use exactly one byte for each character. For example, the letter A will occupy one byte in single-byte encoding.

ASCII is a character-encoding standard. ASCII code represents text in computers and other devices and is a popular choice for many modern character-encoding schemes. In an ASCII file, each alphabetic or numeric character is represented with a seven-bit binary number.

EBCDI is an eight-bit character encoding, which means each alphabetic or numeric character is represented with an eight-bit binary number.

► **Hexadecimal types**

The data type `x` interprets individual bytes in memory. These fields are called *hexadecimal fields*. You can process single bits using hexadecimal fields. Hexadecimal notation is a human-friendly representation of binary-coded values. Each hexadecimal digit represents four binary digits (bits), so a byte (eight bits) can be more easily represented by a two-digit hexadecimal value.

With modern programming languages, we seldom need to work with data at the bits and bytes levels. However, unlike ABAP (which uses single-byte code pages, like ASCII), many external data sources use multibyte encodings. Therefore, `TYPE x` fields are more useful in a Unicode system, in which you can work with the binary data to be processed by an external software application.

For example, you can insert the hexadecimal code 09 between fields to create a tab-delimited file so that spreadsheet software—like Microsoft Excel—knows where each new field starts. `TYPE x` fields are also useful if you wish to create files in various formats from your internal table data.

Predefined Elementary ABAP Types with Variable Length

The data types we've discussed so far either have a fixed length or require a length specification as part of data object declaration. We assume that we know the length of the data we plan to process; for example, a material number is defined in SAP as an alphanumeric field with a maximum of eighteen characters in length. Therefore, if you wish to process a material number in your ABAP program, you can safely create a data object as a `TYPE c` field with `LENGTH 18`. However, there can be situations in which you need a field with dynamic length because you won't know the length of the data until runtime.

For such situations, ABAP provides data types with variable lengths:

► **string**

This is a character type with a variable length. It can contain any number of alphanumeric characters.

► **xstring**

This is a hexadecimal type with a variable length.

When you define a `string` as a data object, only the string header, which holds administrative information, is created statically. The initial length of the `string` data object is 0, and its length changes dynamically at runtime based on the data stored in the data object—for example:

```
DATA path TYPE string.
DATA xpath TYPE xstring.
```

[Table 4.3](#) highlights some of the use cases for the predefined elementary data types.

Data Type	Use Case
Numeric Types	
i	<p>Use to process integers like counters, indexes, time periods, and so on. Valid value range is -2147483648 to +2147483647.</p> <p>If the value range of <code>i</code> is too small for your need, use <code>TYPE p</code> without the <code>DECIMALS</code> addition.</p> <p>Example syntax:</p> <pre>DATA f1 TYPE i. "fixed length</pre>
f	<p>Use to process large values when rounding errors aren't critical. To a great extent, <code>TYPE f</code> is replaced by <code>decfloat</code> (<code>decfloat16</code> and <code>decfloat34</code>) as of SAP NetWeaver AS ABAP 7.1.</p> <p>Use data type <code>f</code> only if performance-critical algorithms are involved and accuracy is not important.</p> <p>Example syntax:</p> <pre>DATA f1 TYPE f. "fixed length</pre>

Table 4.3 Predefined Elementary Data Types and Use Cases

Data Type	Use Case
p	<p>Use type p when fractions are expected with fixed decimals known at design time (distances, amount of money or quantities, etc.).</p> <p>Example syntax:</p> <pre>DATA f1 TYPE p DECIMALS 2. "Takes default length 8 * OR you can manually specify length. DATA f1(4) TYPE p DECIMALS 3.</pre>
decfloat16 and decfloat34	<p>If you need fractions with a variable number of decimal places or a larger value range, use decfloat16 or decfloat34.</p> <p>Example syntax:</p> <pre>DATA f1 TYPE decfloat16. DATA f1 TYPE decfloat34.</pre>
Nonnumeric Types	
c	<p>Use to process alphanumeric values like names, places, or any character strings.</p> <p>Example syntax:</p> <pre>DATA f1 TYPE c. DATA f1(10) TYPE c. DATA f1 TYPE c LENGTH 10.</pre>
n	<p>Use to process numeric values like phone numbers or zip codes.</p> <p>Example syntax:</p> <pre>DATA f1 TYPE n. DATA f1(10) TYPE n. DATA f1 TYPE n LENGTH 10.</pre>
d	<p>Use to process dates; expected format is YYYYMMDD.</p> <p>Example syntax:</p> <pre>DATA f1 TYPE d.</pre>
t	<p>Use to process time; expected format is HHMMSS.</p> <p>Example syntax:</p> <pre>DATA f1 TYPE t.</pre>

Table 4.3 Predefined Elementary Data Types and Use Cases (Cont.)

Data Type	Use Case
x	Use to process the binary value of the data. Useful for working with different code pages. Example syntax: DATA f1 TYPE x. DATA f1(10) TYPE x. DATA f1 TYPE x LENGTH 10.
string	Use when the length of the TYPE c field is known only at runtime. Example syntax: DATA f1 TYPE string.
xstring	Use when the length of the TYPE x field is known only at runtime. Example syntax: DATA f1 TYPE xstring.

Table 4.3 Predefined Elementary Data Types and Use Cases (Cont.)

For arithmetic operations, use numeric fields only. If nonnumeric fields are used in arithmetic operations, the system tries to automatically convert the fields to numeric types before applying the arithmetic operation. This is called *type conversion*, and each data type has specific *conversion rules*. Let's look at this concept in more detail next.

Type Conversions

When you move data between data objects, either the data objects involved in the assignment should be similar (both data objects should be of the same type and length), or the data type of the source field should be convertible to the target field.

If you move data between dissimilar data objects, then the system performs type conversion automatically by converting the data in the source field to the target field using the conversion rules. For the conversion to happen, a conversion rule should exist between the data types involved.

For example, the code in [Listing 4.3](#) assigns a character field to an integer field.

```
DATA f1 TYPE c LENGTH 2 VALUE 23.
DATA f2 TYPE i.
f2 = f1.
```

Listing 4.3 Type Conversion with Valid Content

In [Listing 4.3](#), the data object `f1` is defined as a character field with an initial value of `23`. Since `f1` is of `TYPE c`, the value is interpreted as a character by the system rather than an integer. The listing also defines another data object, `f2`, as an integer type.

When you assign the value of `f1` to the data object `f2`, the system performs the conversion using the applicable conversion rule (from `c` to `i`) before moving the data to `f2`. If the conversion is successful, the data is moved. If the conversion is unsuccessful, the system throws a runtime error. Because the field `f1` has the value `23`, the conversion will be successful, because `23` is a valid integer.

[Listing 4.4](#) assigns an initial value of `T3` for the field `f1`. This is a valid value for a character-type field, but what happens if you try to assign this field to an integer-type field?

```
DATA f1 TYPE c LENGTH 2 VALUE 'T3'.
DATA f2 TYPE i.
f2 = f1.
```

Listing 4.4 Type Conversion with Invalid Content

In [Listing 4.4](#), the field `f1` has the value `T3`; because the system can't convert this value to a number, it'll throw a runtime error by raising the exception `CX_SY_CONVERSION_NO_NUMBER` (see [Figure 4.5](#)) when the statement `f2 = f1` is executed. The runtime error can be avoided by catching the exception, which we will discuss in [Chapter 9](#).

<pre> Error analysis An exception occurred that is explained in detail below. The exception, which is assigned to class 'CX_SY_CONVERSION_NO_NUMBER', was not caught and therefore caused a runtime error. The reason for the exception is: The program attempted to interpret the value "T3" as a number, but since the value contravenes the rules for correct number formats, this was not possible. </pre>
--

Figure 4.5 Runtime Error Raised by Code in Listing 4.4

There are two exceptions: A data object of TYPE t can't be assigned to a data object of TYPE d and vice versa. In addition, assignments between data objects of nearly every different data type are possible.

Conversion Rules

The *conversion rule* for a TYPE c source field and a TYPE i target field is that they must contain a number in commercial or mathematical notation. There are a few exceptions however. The following lists a few of these exceptions:

- ▶ A source field that only has blank characters is interpreted with the number 0.
- ▶ Scientific notations is only allowed if it can be interpreted as a mathematical notation.
- ▶ Decimal places must be rounded to whole numbers.

You can read all the conversion rules and their exceptions by visiting the SAP Help website at http://help.sap.com/abapdocu_750/en/abenconversion_elementary.htm.

Even though the automatic type conversion makes it easy to move data between different types, do not get carried away. Exploiting all the conversion rules to their full extent may give you invalid data. Only assign data objects to each other if the content of the source field is valid for the target field and produces an expected result.

For example, the code in [Listing 4.5](#) will result in invalid data when a TYPE c field containing character string is assigned to a numeric field.

```
DATA f1 TYPE c LENGTH 2 VALUE 'T3'.
DATA f2 TYPE n LENGTH 2.
f2 = f1.
```

Listing 4.5 Example Leading to Data Inconsistency

In [Listing 4.5](#), f2 is of TYPE n and will have the value 03 (T is ignored) instead of raising an exception as in the earlier case, when f2 was declared as TYPE i.

Type conversions are also performed automatically with all the ABAP operations that perform value assignments between data objects (like arithmetic operations).

It's always recommended to use the correct TYPE for the data you plan to process. This not only saves the time of performing type conversions, it also allows your ABAP statements to interpret the data correctly and provide additional functionality.

Say you want to process a date that's represented in an internal format as YYYYMMDD. The following example explains how the `WRITE` statement interprets this data based on the data type. In [Listing 4.6](#), a data object `date` is defined as a `TYPE c` field with `20151130` as the value to represent the date in internal format. When this field is written to the output using the `WRITE` statement, the value is printed as it is in the output. The `WRITE` statement does not interpret this value as a date; instead, it's interpreted as a text value.

```
DATA date TYPE c LENGTH 8 VALUE '20151130'.  
WRITE date.
```

Listing 4.6 Date Stored in TYPE c Field

The output of the code in [Listing 4.6](#) will be `20151130`. If instead you declare the data object `date` as `TYPE d` as shown in [Listing 4.7](#), the output will depend on the date format set by the country-specific system settings or the user's personal settings.

```
DATA date TYPE d VALUE '20151130'.  
WRITE date.
```

Listing 4.7 Date Stored in TYPE d Field

SAP allows you to set your own personal date format. So, say that User 1 has his personal date format set as `MMDDYYYY`, and User 2 has his personal date format set as `DDMMYYYY`. When User 1 executes a program with the code in [Listing 4.7](#), the output would be `11302015`, and when User 2 executes the same program, the output would be `30112015`.

Here, the `WRITE` statement can make use of the system settings/user's personal settings to display the output, because it can interpret the value as a date. In [Listing 4.6](#), however, the value in the data object `v_date` was interpreted as a character string, so the code outputted the value as is.

User-Defined Elementary Data Types

In ABAP, you can define your own elementary data types based on the predefined elementary data types. These are called *user-defined elementary data types* and are useful when you want to define a few related data objects that can be maintained centrally.

User-defined elementary data types can be declared locally in the program using the `TYPE` keyword, or you can define them globally in the system in ABAP Data Dictionary.

Say you want to create a family tree or a pedigree chart. You'd start by defining various data objects to process the names of different family members. The data object declarations would look something like what's shown in [Listing 4.8](#).

```
DATA : Father_Name TYPE c LENGTH 20,
      Mother_Name TYPE c LENGTH 20,
      Wife_Name TYPE c LENGTH 20,
      Son_Name TYPE c LENGHT 20,
      Daughter_Name TYPE c LENGTH 20.
```

Listing 4.8 Data Objects Referring to Predefined Types

Here, it's assumed that the length of a person's name can be a maximum of twenty characters. Later, if you want to extend the length of a person's name to thirty characters, you'll need to edit the definition of all data objects individually.

In [Listing 4.8](#), the source code needs changes in five different places. This may become time-consuming and tedious in larger programs.

To overcome this problem, first you can create a user-defined type and then point all references to a person to this data type, as shown in [Listing 4.9](#).

```
TYPES person TYPE c LENGTH 20.
DATA : Father_Name TYPE person,
      Mother_Name TYPE person,
      Wife_Name TYPE person,
      Son_Name TYPE person,
      Daughter_Name TYPE person.
```

Listing 4.9 Data Objects Referring to User-Defined Types

In [Listing 4.9](#), you first define a user-defined elementary data type `person` using the `TYPE` keyword, and this data type is then used to define other data objects. Here, the data type `person` is based on the elementary `TYPE c`. This way, if you need to extend the length of all data objects referring to a person in your code, you just need to change the definition of your user-defined type rather than edit each individual data object manually.

The basic syntax of the `TYPE` keyword is similar to that of the `DATA` keyword, but with a different set of additions that can be used. Remember that data types don't occupy any memory space on their own and can't be used to store any work data.

You always need to create a data object as an instance of a data type to work with the program data.

Data types and data objects have their own namespaces, which means that a data type and data object with the same name in the same program can exist. However, to avoid confusion, we follow certain naming conventions while defining various data types and data objects. For example, global variables are prefaced with `gv_`, local variables with `lv_`, internal tables with `it_`, and so on. These naming conventions are generally set by a company as part of its programming best practices and coding standards.

Complex Data Types

Complex data types are made up of a combination of other data types and must be created using existing data types. Complex data types allow you to work with interrelated data types under a common name.

For more information on complex data types, see [Chapter 5](#).

Reference Types

Reference types describe reference variables (data references and object references) that provide references to other objects. For more information on reference types, see [Chapter 8](#). For more information on data references see [Chapter 16](#).

Internal and External Format of Data

As you've seen, data can be represented in various formats. Say you have a requirement indicating that the user wants to create a document and capture the due date for each document. This data should be updated in the database so that the user can later run a report to pull out all the documents that are due on a particular date.

While creating the document, users would prefer to input the due date in their own personal format. For example, let's assume there are two users, User 1 and User 2, who will be using this application to create documents. User 1 inputs the due date in MM/DD/YYYY format, and User 2 inputs the date in DD-MM-YYYY format. Assume both users have created four documents in total and that the data

has been updated in the database. The data in the database table would look like [Table 4.4](#).

Document_Number	Due_Date	Created_By
1001	11/05/2015	USER1
1002	11/06/2015	USER1
1003	05-11-2015	USER2
1004	06-11-2015	USER2

Table 4.4 Data Represented in External Format

Now, the report you want to develop takes the due date as an input to pull out the documents that are due on that particular date. This report will have a select statement something like the following to query the database table:

```
SELECT Document_Number FROM db_table WHERE Due_Date EQ inp_due_date.
```

Here, you're comparing the data in the `Due_Date` column of the table with the user-provided date to fetch all the matching document numbers. As you can see, if the user inputs the November 5, 2015, as the due date in MM/DD/YYYY format (11/05/2015), then the `SELECT` statement will fetch only document number 1001 and ignores document 1003, because the latter doesn't match character-for-character with the given input. If the input is given in DD-MM-YYYY format (05-11-2015), then document 1003 is fetched and 1001 is ignored. If you input the date in any other format, none of the records are fetched.

Internal and external data formats can help with this issue. If you maintain the internal format of the date as YYYYMMDD, you can convert the user-provided date to the internal format (20151105) and save it to the database. In this way, the data can be saved in the same format consistently. Similarly, you can convert the date to an external format before displaying it to the user so that the user always sees the date in his own preferred format.

Following this concept, the data in [Table 4.5](#) can be represented as shown.

Document_Number	Due_Date	Created_By
1001	20151105	USER1
1002	20151106	USER1

Table 4.5 Dates Represented in Internal Format

Document_Number	Due_Date	Created_By
1003	20151105	USER2
1004	20151106	USER2

Table 4.5 Dates Represented in Internal Format (Cont.)

Now, when the user inputs a date in your report, it's always converted to the internal format first before querying the database table to fetch the matching records. This allows the user to input the date in any of the valid external formats per his personal choice without worrying about the format the program expects.

Output Length of Data Types

In the previous section, the external format of date included two separators (/ or -). The internal length of a date is eight characters, but to represent the date in an external format requires two extra characters to accommodate the separators. Therefore, the external length should be a minimum of ten characters. This is determined by the output length of the data type.

If you examine the data object of TYPE I in the debugger, you'll see that different lengths have been assigned for LENGTH and OUTPUT LENGTH, as shown in [Figure 4.6](#).

Single Field	f2
Field Contents	0
Type	I
Length	4
Output Length	11
Decimal Places	0

Figure 4.6 Output Length

As shown in [Figure 4.6](#), the data object F2 is of TYPE I with LENGTH 4 bytes. However, the OUTPUT LENGTH is 11 to accommodate the thousands separator for the value.

If you check the same for a data object of TYPE D, you'll see that the LENGTH and the OUTPUT LENGTH are the same—eight bytes—as shown in [Figure 4.7](#).

Single Field	£2
Field Contents	00000000
Type	D
Length	8
Output Length	8
Decimal Places	0

Figure 4.7 Output Length of Type d

If you return to [Listing 4.7](#), you'll see that the system outputted the date without the separators, because there was no space in the data object to accommodate the separators. Therefore, it simply converted the date to external format while ignoring the separators. The output length of each data type is predefined and can't be overridden manually.

If you want more control, you can create your own user-defined elementary data types, as described earlier in this section.

4.4.2 Data Elements

As mentioned earlier, user-defined elementary data types can be created for local reusability within the program or for global reusability across multiple programs. The global user-defined elementary types are called *data elements* and are created in ABAP Data Dictionary.

Data types defined using the `TYPE` keyword are only visible within the same program that they're created in. If you want to create an elementary user-defined type with global visibility across the system, you can do so in ABAP Data Dictionary. As you may recall from [Chapter 3](#), ABAP Data Dictionary is completely integrated into ABAP Workbench. This allows you to create and manage data definitions (metadata) centrally.

Now, let's create a data element in ABAP Data Dictionary. Proceed through the following steps to create a data element `ZCB_PERSON`, which is of `TYPE c` and `LENGTH 20`:

1. Open ABAP Data Dictionary via Transaction SE11 in the command bar or by navigating to the menu path `TOOLS • ABAP WORKBENCH • DEVELOPMENT • ABAP DICTIONARY` in SAP menu.

2. Select the DATA TYPE radio button, provide a name for your data element in the field to the right of the button, and click the CREATE button. Because the data elements are repository objects, they should exist in a customer namespace (i.e., the data element name should start with Z or Y; see [Figure 4.8](#)).

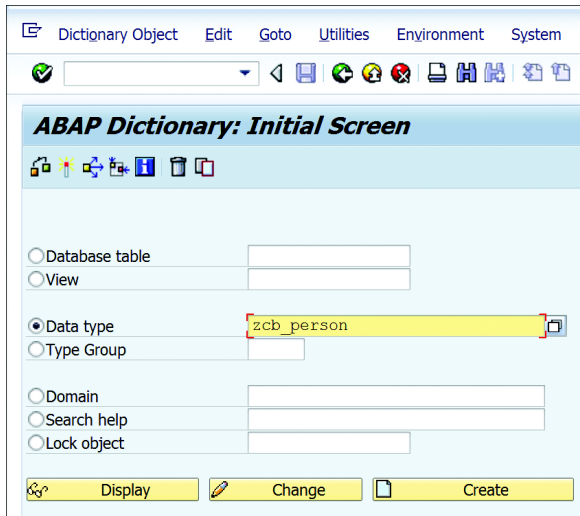


Figure 4.8 ABAP Data Dictionary: Initial Screen

3. The system presents a dialog box (see [Figure 4.9](#)) asking you to select the data type you want to create. DATA ELEMENT represents the user-defined elementary data type, so select the DATA ELEMENT radio button and click CONTINUE (green checkmark). (We'll have an opportunity to explore structures and table types in later chapters.)

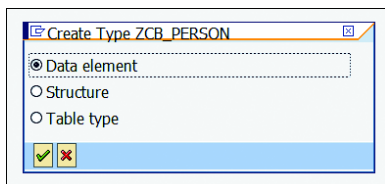


Figure 4.9 Create Type

4. On the next screen (see [Figure 4.10](#)), provide a short description for your data type to help others understand its purpose. This short text is also displayed as a title in the **[F1]** help for all screen fields referring to this data element.

Note that the DATA TYPE tab is selected by default and the radio button ELEMENTARY TYPE preselected. Here, you have two options to maintain the technical attributes for your data element: You can either derive them from a domain or use one of the predefined types.

For now, use the predefined elementary data type; we'll explore the domain concept in the next section.

The screenshot shows the SAP 'Dictionary: Change Data Element' screen. The 'Data Element' tab is selected, and the data element name is 'ZCB_PERSON'. The 'Data Type' tab is active, and the 'Elementary Type' radio button is selected. Under 'Elementary Type', the 'Domain' radio button is selected, and the 'Predefined Type' radio button is unselected. The 'Data Type' field is empty, and the 'Length' field is set to '0'. The 'Decimal Places' field is also set to '0'.

Figure 4.10 Change Data Element Screen

5. Select the PREDEFINED TYPE radio button and enter the predefined type name in the DATA TYPE field. Note that ABAP Data Dictionary has its own predefined elementary data types that correspond to the predefined elementary data types in ABAP programs we discussed earlier.
6. Because we plan to create a character data type, you can enter "CHAR" in the DATA TYPE field or select it by using **F4** help in the field (see [Figure 4.11](#)).
7. Enter the length of the data type in the LENGTH field. For this data element, enter "20". Click the ACTIVATE button (see [Figure 4.12](#)) or press **Ctrl** + **F3** to activate the data element. Save it to a package, and click CONTINUE when an informative message asks you to maintain field labels.

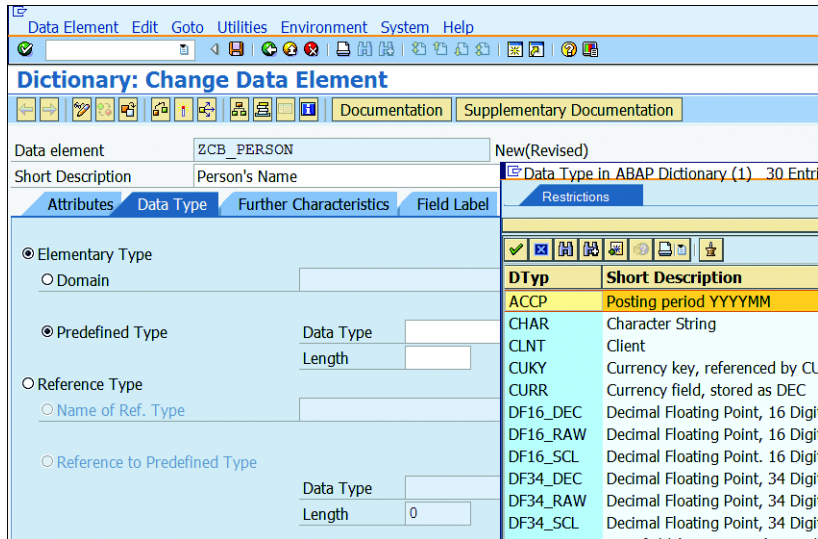


Figure 4.11 Data Type Value List

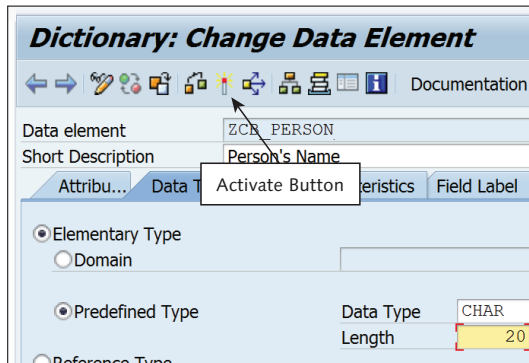


Figure 4.12 Activating a Data Element

8. If the data type supports decimals, you can also maintain the number of decimal places in the DECIMAL PLACES field.

The data type we just created has global visibility and can be used to define data objects or user-defined data types in any program, as shown:

```
DATA name TYPE ZCB_PERSON.
TYPES user TYPE ZCB_PERSON.
```

If you change the definition of the data element, the changes will be automatically reflected for all the objects referring to that data element. This allows you to centrally maintain the definitions of all related objects. Because data elements are global objects, it's recommended not to change them without analyzing the impact first.

There are many things that can be maintained at the data element level, like search helps, field labels, and help documentation. The program fields can derive these properties automatically. We'll explore these concepts in greater detail in [Chapter 10](#).

4.4.3 Domains

A *domain* describes the technical attributes of a field. The primary function of a domain is to define a value range that describes the valid data values for the fields that refer to this domain.

However, if you plan to create multiple data elements that are technically the same, you can attach a domain to derive the technical attributes of a data element. This allows you to manage the technical attributes of multiple data elements centrally, meaning that you can change the technical attributes once for the domain, and the change will be reflected automatically for all the data elements that use the domain. This is an alternative to changing each individual data element to maintain technical attributes when a predefined elementary data type is selected.

A field can't be referred to a domain directly; it picks up the domain reference through a data element if the domain is attached to the data element. In other words, you always attach a domain to a data element. [Figure 4.13](#) depicts this relationship.

In [Figure 4.13](#), Field 1 refers to Data Element 1, whereas Field 2 and Field 3 refer to Data Element 2. Both Data Element 1 and Data Element 2 are using the same domain to derive technical attributes. This implies that Field 2 and Field 3 are both semantically and technically the same, whereas Field 1 is *semantically* different from Field 2 and Field 3 but is *technically* the same as these two fields.

For example, a sales document number and a billing document number can both be technically the same, such as both being ten-digit character fields. However, the sales document and billing document are semantically different (i.e., their purposes are different). For such a case, you can use the same domain for both the

sales document number and billing document number but a separate data element for each.

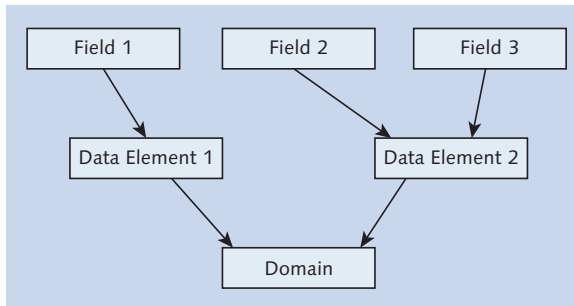


Figure 4.13 Fields, Data Elements, and Domain Relationships

You can also maintain a *conversion routine* at the domain level for automatic conversion to internal and external data formats for fields referring to a domain. We'll discuss this topic more in [Chapter 10](#).

You can take a *top-down* approach or *bottom-up* approach to create a domain; that is, you either can create the domain separately first and attach it to a data element or can create the domain while creating the data element. Let's take a bottom-up approach to create a domain and attach it to the previously created data element ZCB_PERSON.

Because domains and data elements have their own namespaces, you can use the same name for both of them. In the following example, you'll create a domain using the same name as the data element created previously in [Section 4.4.2](#); you can use a different name if it becomes too confusing.

The following steps will walk you through the procedure to create a domain in ABAP Data Dictionary:

1. On the initial ABAP Data Dictionary screen ([Figure 4.14](#)), select the DOMAIN radio button, input the name of the domain, and click the CREATE button. Domains are also repository objects, so they should exist in a customer namespace.
2. Enter a SHORT DESCRIPTION for the domain. This short description is never seen by end users. It's displayed when searching for domains using `[F4]` help and it helps other developers understand your domain's purpose.

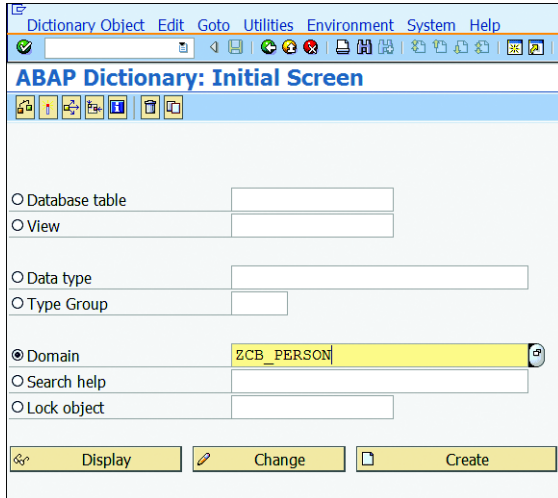


Figure 4.14 ABAP Dictionary: Initial Screen

- Use the help for the DATA TYPE field to select the data type and enter a value for the NO. CHARACTERS field. This value defines the field length (Figure 4.15). Optionally, you can enter the number of DECIMAL PLACES for numeric types.

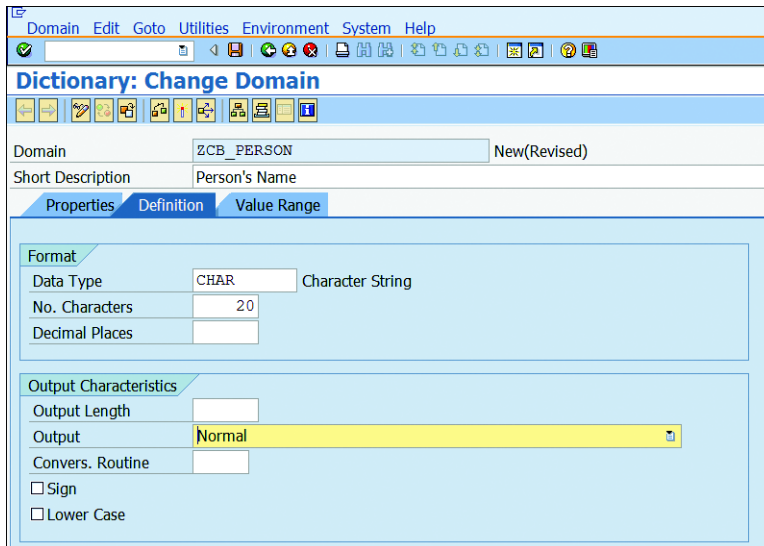


Figure 4.15 Change Domain Screen

4. You can also enter OUTPUT CHARACTERISTICS like output length, conversion routine, or disabling automatic uppercase conversion for screen fields referring to this domain. We'll explore these options further in [Chapter 10](#).
5. Activate the domain. To attach this domain to a data element, open the data element and select the DOMAIN radio button under the ELEMENTARY TYPE radio button, as shown in [Figure 4.16](#). Type the name of the domain and press .
6. Notice that the DATA TYPE and LENGTH are chosen by the system automatically. If the system does not set the DATA TYPE and LENGTH after pressing the key, it means the domain does not exist, and an informative message, NO ACTIVE DOMAIN [DOMAIN_NAME] AVAILABLE, is displayed.

At this point, you can simply double-click the domain name to create it using the top-down approach.

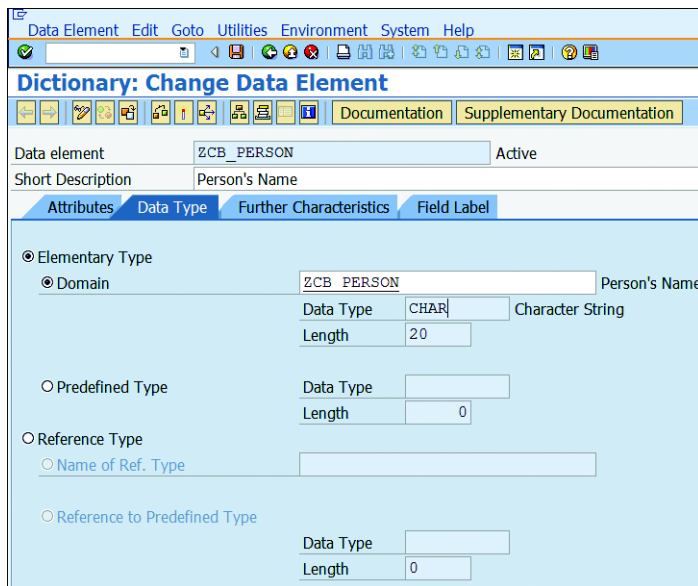


Figure 4.16 Attaching Domain to Data Element

4.4.4 Data Objects

Data objects derive their technical attributes from data types and occupy memory space to store the work data of a program. ABAP statements access this content by addressing the name of the data object. Data objects exist as an instance of data

types. Each ABAP data object has a set of technical attributes, such as data type, field length, and number of decimal places.

Data objects are the physical memory units with which your ABAP statements can work. ABAP statements can address and interpret the contents of data objects. All data objects are declared in the ABAP program and are local to the program, which means they exist in the program memory and can be accessed only from within the same program. There is no concept of defining data objects centrally in the system.

Data objects are not persistent; they exist only while the program is being executed. The life of the data object lasts only as long as the program execution lasts. They are created when the program execution starts and destroyed when the program execution ends.

Before you can process persistent data (such as data from a database or a sequential file), you must read the data into data objects first, which can then be accessed by ABAP statements. Similarly, if you want to retain the contents of a data object beyond the end of the program, you must save it in a persistent form. Technically, anything that can store work data in a program can be called a data object.

Let's explore different kinds of data objects that can be defined in your ABAP programs. In the following subsections, we will look at the different classifications of data objects.

Literals

Literals are not created using any declarative statements, nor do they have names. They exist in the program source code and are called *unnamed data objects*. Like all data objects, they have fixed technical attributes.

You cannot access the memory of a literal to read its content, because it's an unnamed data object. This means that literals are not reusable data objects and their contents can't be changed. Unlike literals, all other data objects are named data objects, which are explicitly declared in the program.

For example, in the following snippet, `Hello World` and `1234` are literals:

```
WRITE 'Hello World'.  
WRITE 1234.
```

There are two types of literals:

► **Numeric literals**

Numeric literals are a sequence of digits (0 to 9) that can have a prefixed sign. They do not support decimal separators or notation with a mantissa and exponent.

Examples of numeric literals include the following:

- +415
- -345
- 400

The following excerpt shows numeric literals used in ABAP statements:

```
DATA f1 TYPE I VALUE -4563.
WRITE 1234.
F1 = 1234.
```

► **Character literals**

Character literals are alphanumeric characters in the source code of the program enclosed in single quotation marks (') or back quotes (`).

For example:

```
'This is a text field literal'.
'1901 CA'.
`This is a string literal`.
`1901 CA`.
```

Character literals maintained in single quotes have the predefined type `c` and length matching the number of characters. These are called *text field literals*. The ones maintained with back quotes have the predefined type `string` and are called *string literals*. If you use character literals where a numeric value is expected, they are converted into a numeric value. Examples of character literals that can be converted to numeric types include the following:

- '12345'.
- '-12345'.
- '0.345'.
- '123E5'.
- '+23E+12'.

Variables

Variables are data objects for which content can be changed via ABAP statements. Variables are named data objects and are declared in the declaration area of the program. Different keywords, like `DATA` and `PARAMETERS`, declare different types of variables.

For now, let's explore these two keywords, which define variables, and defer the discussion of other keywords to later chapters.

DATA

The `DATA` keyword defines a variable in every context. You can use this keyword in the global declaration area of the ABAP program to declare a global variable that has visibility throughout the program, or you can use it inside a procedure to declare a local variable with visibility only within the procedure—for example:

```
DATA field1 TYPE i
```

In the preceding statement, a variable with the name `field1` is defined as an integer field, using the `DATA` keyword.

Inline Declarations

SAP NetWeaver 7.4 introduced *inline declaration*, which no longer restricts you to define your local variables separately at the beginning of the procedure. You can define them inline as embedded in the given context, helping you to make your code thinner. We'll explore this concept in later chapters.

PARAMETERS

The `PARAMETERS` keyword plays a dual role. It defines a variable within the program context and also generates a screen field (selection screen). This keyword is used to create a selection screen for report programs. In these report programs, we present a selection screen for the user to input the selection criteria for report processing—for example:

```
PARAMETERS p_input TYPE c LENGTH 10.
```

In the preceding statement, a ten-character input field with the name `p_input` is defined on the selection screen using the `PARAMETERS` keyword. This field also exists as a variable in the program and is linked to the screen field. The input made on the selection screen for this input field will be stored in the program as the content of the `p_input` variable.

Constants

Constants are named data objects that are declared using a keyword and whose content cannot be changed by ABAP statements. The keyword used to declare a constant is `CONSTANT`. It is recommended to use constants in lieu of literals wherever possible. Unlike literals, constants can be reused and maintained centrally. The syntax of constants statement mostly matches the data statement. However, with constants statements, it is mandatory to use the addition `VALUE` to assign an initial value. This value cannot be changed at runtime. For example:

```
CONSTANTS c_item TYPE c LENGTH 4 VALUE 'ITEM'.
```

Text Symbols

A *text symbol* is a named data object in an ABAP program that is not declared in the program itself. Instead, it's defined as a part of the text elements of the program.

A text symbol behaves like a constant and has the data type `c` with the length defined in the text element. We'll explore text symbols further in [Chapter 6](#).

An example of a text symbol is `WRITE text-001`. In this statement, a text symbol of the program with the number `001` is written to the output using the `WRITE` statement. This text symbol is defined separately in ABAP Editor via the menu path `GOTO • TEXT ELEMENTS • TEXT SYMBOLS`.

Text symbols are accessed using the syntax `text-nnn`, where *nnn* is the text symbol number.

4.5 ABAP Statements

As we discussed earlier, the source code of an ABAP program is made up of various ABAP statements. Unlike other programming languages like C/C++ or Java, which contain a limited set of language-specific statements and provide most functionality via libraries, ABAP contains an extensive set of built-in statements. We'll explore many ABAP statements as we progress in this book.

The best way to learn about the various ABAP statements available is to put them in perspective with the requirements at hand. It is beyond the scope of this book to cover all the available ABAP statements, so we'll give a brief introduction to

some popular statements here, and we'll explore these in greater detail in the upcoming chapters:

► **Declarative statements**

Declarative statements define data types or declare data objects that are used by the other statements in a program.

Examples include `TYPE`, `DATA`, `CONSTANTS`, `PARAMETERS`, `SELECT-OPTIONS`, and `TABLES`.

► **Modularization statements**

Modularization statements define the processing blocks in an ABAP program. *Processing blocks* allow you to organize your code into modules. All ABAP programs are made up of processing blocks, and different processing blocks allow you to modularize your code differently. We will explore this topic further in [Chapter 7](#).

Examples include the `LOAD-OF-A-PROGRAM`, `INITIALIZATION`, `AT SELECTION SCREEN`, `START-OF-SELECTION`, `END-OF-SELECTION`, `AT USER-COMMAND`, `AT LINE-SELECTION`, `GET`, `AT USER COMMAND`, `AT LINE SELECTION`, `FORM-ENDFORM`, `FUNCTION-ENDFUNCTION`, `MODULE-ENDMODULE`, and `METHOD-ENDMETHOD`.

► **Control statements**

Control statements control the flow of the program within a processing block. Examples include `IF-ELSEIF-ELSE-ENDIF`, `CASE-WHEN-ENDCASE`, `CHECK`, `EXIT`, and `RETURN`.

► **Call statements**

Call statements are used to call processing blocks or other programs and transactions.

Examples include `PERFORM`, `CALL METHOD`, `CALL TRANSACTION`, `CALL SCREEN`, `SUBMIT`, `LEAVE TO TRANSACTION`, and `CALL FUNCTION`.

► **Operational statements**

Operational statements allow you to modify or retrieve the contents of data objects.

Examples include `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, `SEARCH`, `REPLACE`, `CONCATENATE`, `CONDENSE`, `READ TABLE`, `LOOP AT`, `INSERT`, `DELETE`, `MODIFY`, `SORT`, `DELETE ADJACENT DUPLICATES`, `APPEND`, `CLEAR`, `REFRESH`, and `FREE`.

► **Database access statements (Open SQL)**

Database access statements allow you to work with the data in the database. Examples include `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `MODIFY`.

In this chapter thus far, we've explored the general program structure of an ABAP program, learned basic rules for using ABAP syntax, touched on the use of ABAP keywords and ABAP keyword documentation, and introduced the `TYPE` concept for data types, data elements, and data objects.

In the next section, you'll start creating your first ABAP program using what you've learned so far.

4.6 Creating Your First ABAP Program

Before we explore ABAP basics further, you'll have the chance to create your first program using ABAP Editor.

You'll need developer access to the SAP system with relevant development authorizations and a developer key assigned to your user ID. Contact your system administrator if the system complains about missing authorizations or prompts you for a developer key when creating the program.

To begin, proceed with the following steps:

1. Open Transaction SE38 or follow the menu path `TOOLS • ABAP WORKBENCH • DEVELOPMENT • ABAP EDITOR`.
2. On the ABAP Editor initial screen, enter the program name, select the `SOURCE CODE` radio button and click the `CREATE` button. Because this program will be a repository object, it should be in a customer namespace starting with Z or Y, as shown in [Figure 4.17](#).
3. You'll see the `PROGRAM ATTRIBUTES` window ([Figure 4.18](#)) to maintain the attributes for your program. *Program attributes* allow you to set the runtime environment of the program.

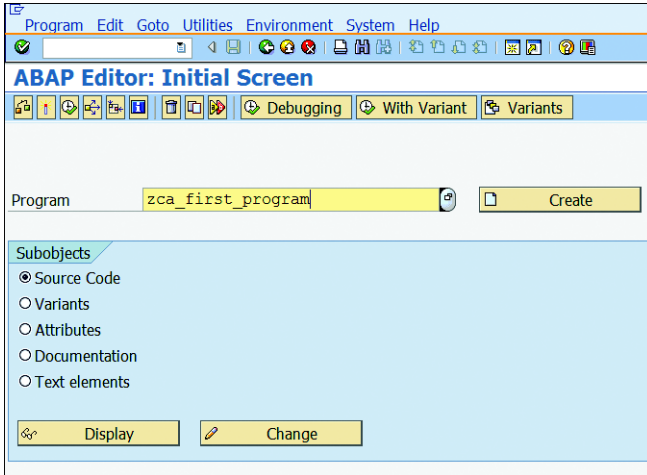


Figure 4.17 ABAP Editor: Initial Screen

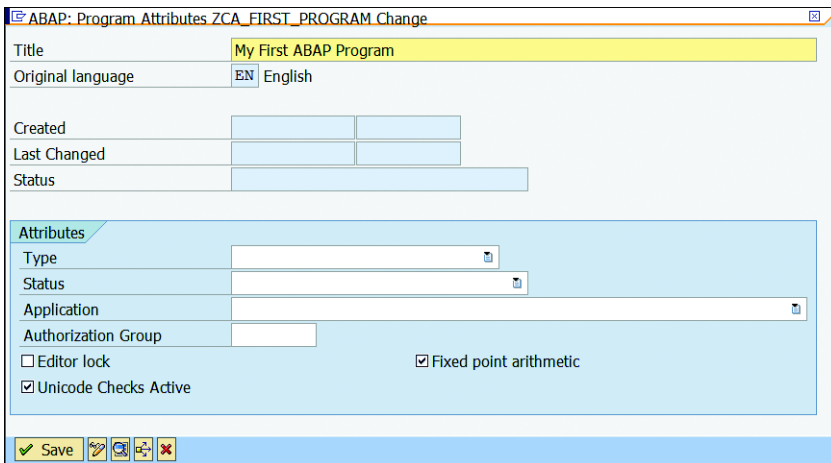


Figure 4.18 Program Attributes Screen

Here, you can maintain a title for your program and other attributes. The TITLE and TYPE are mandatory fields; the others are optional.

Table 4.6 provides an explanation of all the attributes and their significance.

Attribute	Explanation
TYPE	<p>Allows you to select the type of program you wish to create. This is the most important attribute and specifies how the program is executed. It's a mandatory field.</p> <p>From within ABAP Editor, you can only choose from the following program types:</p> <ul style="list-style-type: none"> ▶ EXECUTABLE PROGRAM ▶ MODULE POOL ▶ SUBROUTINE POOL ▶ INCLUDE PROGRAM <p>All other program types are not created directly from within ABAP Editor but are created with the help of special tools such as Function Builder for function groups or Class Builder for class pools.</p>
STATUS	<p>Allows you to set the status of the program development—for example, production program or test program.</p>
APPLICATION	<p>Allows you to set the program application area so that the system can allocate the program to the correct business area. For example, SAP ERP Financial Accounting.</p>
AUTHORIZATION GROUP	<p>In this field, you can enter the name of a program group. This allows you to group different programs together for authorization checks.</p>
LOGICAL DATABASE	<p>Visible only when the program type is selected as an executable program. This attribute determines the logical database used by the executable program. Used only when creating reports using a logical database.</p> <p>Logical databases are special ABAP programs created using Transaction SLDB that retrieve data and make it available to application programs.</p>
SELECTION SCREEN	<p>Visible only when the program type is selected as an executable program. Allows you to specify the selection screen of the logical database that should be used.</p>
EDITOR LOCK	<p>If you set this attribute, other users can't change, rename, or delete your program. Only you will be able to change the program, its attributes, text elements, and documentation or release the lock.</p>

Table 4.6 Program Attributes

Attribute	Explanation
FIXED POINT ARITHMETIC	If this attribute is set for a program, the system rounds type p fields according to the number of decimal places, or pads them with zeros. The decimal sign in this case is always the period (.), regardless of the user's personal settings. SAP recommends that this attribute is always set.
UNICODE CHECKS ACTIVE	This attribute allows you to set if the syntax checker should check for non-Unicode-compatible code and display a warning message. As of SAP NetWeaver 7.5, the system does not support non-Unicode systems, so this option is always selected by default.
START USING VARIANT	Applicable only for executable programs. If you set this attribute, other users can only start your program using a variant. You must then create at least one variant before the report can be started.

Table 4.6 Program Attributes (Cont.)

4. Under the ATTRIBUTES section, provide the following values (see [Figure 4.19](#)):

- ▶ TYPE: EXECUTABLE PROGRAM
- ▶ STATUS: TEST PROGRAM
- ▶ APPLICATION: UNKNOWN APPLICATION
- ▶ Also check the UNICODE CHECKS ACTIVE and FIXED POINT ARITHMETIC checkboxes.

Click the SAVE button when finished.

The screenshot shows the 'Attributes' dialog box in SAP. The 'Type' field is set to 'Executable program', 'Status' to 'Test Program', and 'Application' to 'Unknown application'. The 'Unicode Checks Active' and 'Fixed point arithmetic' checkboxes are checked. The 'Save' button is highlighted in yellow at the bottom left.

Figure 4.19 Maintaining Program Attributes

5. On the CREATE OBJECT DIRECTORY ENTRY screen (see [Figure 4.20](#)), you can assign the program to a package. The package is important for transports between systems. All of the ABAP Workbench objects assigned to one package are combined into one transport request.

When working in a team, you may have to assign your program to an existing package, or you may be free to create a new package. All programs assigned to the package \$TMP are private objects (local object) and can't be transported into other systems. For this example, create the program as a local object.

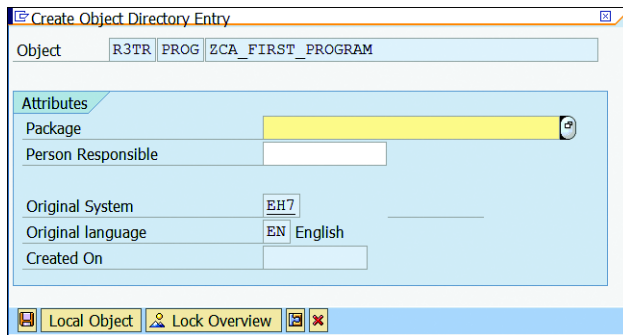


Figure 4.20 Assigning Package

6. Either enter "\$TMP" as the PACKAGE and click the SAVE button or simply click the LOCAL OBJECT button (see [Figure 4.20](#)).
7. This takes you to the ABAP EDITOR: CHANGE REPORT screen where you can write your ABAP code. By default, the source code includes the introductory statement to introduce the program type. For example, executable programs are introduced with the REPORT statement, whereas module pool programs are introduced with the PROGRAM statement.
8. Write the following code in the program and activate the program:


```
PARAMETERS p_input TYPE c LENGTH 20.

WRITE : 'The input was:', p_input.
```
9. In the code, you're using two statements—PARAMETERS and WRITE. The PARAMETERS statement generates a selection screen (see [Figure 4.21](#)) with one input field. The input field will be of TYPE c with LENGTH 20, so you can input up to twenty alphanumeric characters.

Here, the `PARAMETERS` statement is performing a couple of tasks:

- ▶ Declaring a variable called `p_input`.
- ▶ Generating a screen field (screen element) on the selection screen with the same name as the variable `p_input`. The screen field `p_input` is automatically linked to the variable sharing the same name. Therefore, the data transfer between the screen and the program is handled automatically. In other words, if you enter some data in the `p_input` screen field, it will be automatically transferred to the program and stored in the `p_input` variable. This data in the `p_input` variable can then be accessed by other statements, like you're doing with the `WRITE` statement to print it in the output.

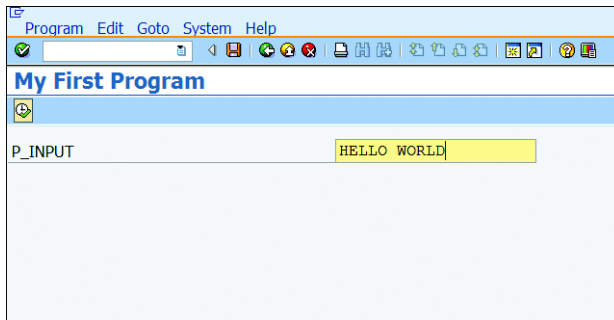


Figure 4.21 Selection screen

10. Enter a value in the input field and click the `EXECUTE` button, or press `F8`. This should show you the output or `LIST` screen, as shown in [Figure 4.22](#).

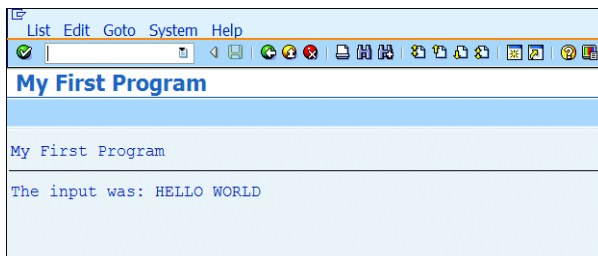


Figure 4.22 List Screen

11. The list screen (output screen) is generated by the `WRITE` statement in the code. With the `WRITE` statement, you're printing the contents of two data objects

here. One is a text literal, 'The input was:', and the other is the variable `p_input` created by the `PARAMETERS` keyword.

Whatever input was entered on the selection screen was transferred to the program and stored in the variable `p_input`, which was then processed by the `WRITE` statement to display in the output.

We'll have many opportunities to work with different types of screens as we progress through the book.

4.7 Summary

In this chapter, you learned the basics of the ABAP programming language elements. You saw that ABAP programs can only work with data of the same program. Because we typically process various kinds of data in the real world, we explained the different elementary types that are supported by the system to process various kinds of data. Because the basic task of an ABAP program is to process data, this understanding is crucial to process the data consistently.

We also described the syntax to write a couple of statements. We'll explore more statements as we progress through the book. In the final section, you created your first ABAP program.

By now, you should have a good understanding of data types and data objects. However, if you find yourself a little lost, don't worry; the concepts should make more sense as we work through creating more programs in the next chapter. In the next chapter, we'll discuss complex data types and internal tables.

This chapter explores complex data types and lays the foundations for working with data stored in typical database tables.

5 Structures and Internal Tables

In the previous chapter, we discussed how to declare and work with individual fields. However, most of the time, we'll be working with data from a group of related fields. For example, when you store a customer's data, you won't store all of it in one field; you'll create multiple fields to store all the different information about the customer: one field to store the customer's name, one for the customer's address, one for phone number, and so on.

In this chapter, we'll discuss the data types and data objects that let us work with data in the form of an array (a series of objects). [Figure 5.1](#) shows some sample data in a spreadsheet. Here, each field is represented by a column, and each row stores information about a customer.

	A	B	C	D
1	Name	Address	Phone Number	Email ID
2	James	ABC COMPANY Chicago	111-111-111	abc@company.com
3	Ria	XYZ Corp Bangalore	222-222-222	abc@corp.com
4				

Figure 5.1 Data Represented in Rows and Columns

The data in a database table looks similar to what's shown in [Figure 5.1](#). When you want to process this data from a database table in an ABAP program, you need data objects that can store this data in a similar fashion, like a set of fields grouped together to represent a row.

These data types and data objects with individual fields are grouped together into *structure types* and *structures*, respectively. We use the term *structure type* to refer to a data type and the term *structure* to refer to a data object. A structure consists of one or more fields. The fields in a structure are called *components*. Each field in a structure is defined using a *type* or *like* reference, similar to an individual field (variable), which we discussed in [Chapter 4, Section 4.4.2](#).

In [Section 5.1](#), we'll look at different ways to define and work with structures. A structure data object consists of only one row, meaning that it can only store one row of data. When you store a new record in the structure, it replaces the existing record. To process multiple rows of data, we need to use internal tables. We'll discuss internal tables in [Section 5.2](#).

Structures and internal tables are mostly used when working with data from a database table; such data is accessed via Open SQL statements. You'll learn more about relational databases and look at different Open SQL statements in [Section 5.3](#). We'll then look at how to process data from database tables using structures and internal tables in [Section 5.4](#).

It's common, especially for a beginner, for programs to have logical issues that are difficult to detect statically during the syntax check. We'll discuss *debugging* in [Section 5.5](#) to help you become familiar with the debugging tool in order to fix any logical issues with your programs. Debugging also helps you understand the program execution better, and we encourage you to debug all the programs you create during the course of this book.

We'll conclude this chapter with an assignment in [Section 5.6](#), which will reinforce the concepts discussed in this chapter.

5.1 Defining Structures

Structures are either local or global. There are several definitions for both local and global, which we will cover later. Based on the context, the term *local* can mean a couple of things; if we're talking about a program, local may refer to an object (data object or a data type) that is native to a particular procedure within the program. These objects will be visible within the procedure where they're defined. In this context, a *global* object is the one that has visibility throughout the program; it can be accessed from any procedure or processing block in the program. *Visibility* indicates whether a data object or a data type can be accessed from a particular location in the program code.

For example, as shown in [Listing 5.1](#), you can access the data type `ty_sflight` and the data object `ls_sflight`, but only within the subroutine `data`. Trying to access them from another procedure (the subroutine `output` in the example) of the same program will result in a syntax error, because these objects aren't visible

outside their parent procedure. However, if you declare both `ty_sflight` and `ls_sflight` in the global declaration area of the program, they can be accessed from any processing block of the program.

```
FORM data.
TYPES : BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
      END OF ty_sflight.
DATA ls_sflight TYPE ty_sflight.
ENDFORM.

FORM output.
*Both of the lines below will result in a syntax error.
DATA lv_sflight TYPE ty_sflight.
SELECT SINGLE carrid connid fldate FROM sflight
          INTO ls_sflight
          WHERE carrid EQ 'AA'.

ENDFORM.
```

Listing 5.1 Using Local Declaration within Subroutine

If we're talking about an SAP system, then local and global will have different meanings. In such a case, *global* means that the object is visible across the system and can be accessed from any ABAP program. Conversely, *local* means that the object is visible only within a particular program and can't be accessed from other programs.

There are three types of ABAP structures, as depicted in [Figure 5.2](#):

► **Flat structures**

Components are made of individual fields that refer to elementary types.

► **Nested structures**

If at least one component of a structure refers to another structure, it's a nested structure.

► **Deep structures**

References at least one table type (a type that defines an internal table).

In this section, you'll learn how structures are useful in programs. We'll look at how structure types can be defined both locally in the program and globally in ABAP Data Dictionary. You'll also learn how data can be stored and accessed with structures. We'll conclude this section by providing some use cases for structures.

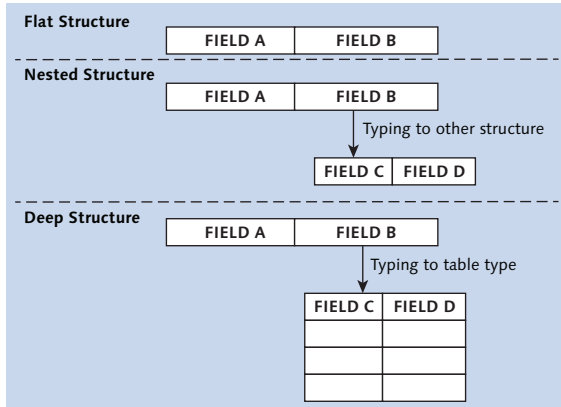


Figure 5.2 Different Structure Types

5.1.1 When to Define Structures

We use structures to process sequential data, such as the data stored in the rows and columns of a database table or a sequential file. Data objects allow you to import data from an external source (such as a database table or a sequential file) into your ABAP program, which has to happen before that same data can be accessed by ABAP statements (see [Figure 5.3](#)).

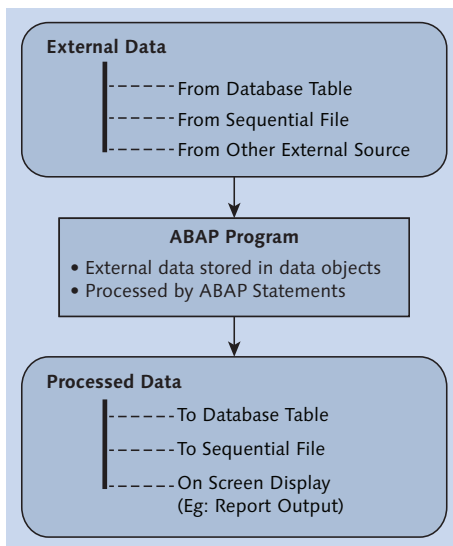


Figure 5.3 Data Processing within ABAP Programs

Therefore, whether you need a single variable or a structure with multiple fields depends on the data that you're importing to process in your ABAP program. If you're working with more than one field in a row or a complete row of data, define a structure in your program. If you're working with an individual field or just one field from a row, define a single field in your program.

5.1.2 Local Structures

Let's take the example of table SFLIGHT to understand the different ways to define structures in your ABAP program. Once you understand how to define a structure, you should be able to put them to use to process data from the database table.

Figure 5.4 shows the table SFLIGHT fields.

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_MANDT	CLNT	3	0	Client
CARRID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_CARR_ID	CHAR	3	0	Airline Code
CONNID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_CONN_ID	NUMC	4	0	Flight Connection Number
FLDATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S_DATE	DATS	8	0	Flight date
PRICE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_PRICE	CURR	15	2	Airfare
CURRENCY	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_CURRCODE	CUKY	5	0	Local currency of airline
PLANETYPE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_PLANETYE	CHAR	10	0	Aircraft Type
SEATSMAX	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_SEATSMAX	INT4	10	0	Maximum capacity in economy class
SEATSOCC	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_SEATSOCC	INT4	10	0	Occupied seats in economy class
PAYMENTSUM	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_SUM	CURR	17	2	Total of current bookings
SEATSMAX_B	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_SMAX_B	INT4	10	0	Maximum capacity in business class
SEATSOCC_B	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_SOCC_B	INT4	10	0	Occupied seats in business class
SEATSMAX_F	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_SMAX_F	INT4	10	0	Maximum capacity in first class
SEATSOCC_F	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S_SOCC_F	INT4	10	0	Occupied seats in first class

Figure 5.4 Table SFLIGHT Fields

Figure 5.5 shows the table data in the data browser tool. To open the table, go to Transaction SE11, select the DATABASE TABLE radio button, enter the table name, and click DISPLAY.

Data Browser: Table SFLIGHT Select Entries 403

Table: SFLIGHT
 Displayed Fields: 14 of 14 Fixed Columns: 4 List Width 0250

MANDT	CARRID	CONNID	FLDATE	PRICE	CURRENCY	PLANETYPE	SEATSMAX	SEATS
800	AA	0017	11.02.2009	422,94	USD	747-400	385	
800	AA	0017	11.03.2009	422,94	USD	747-400	385	
800	AA	0017	08.04.2009	422,94	USD	747-400	385	
800	AA	0017	06.05.2009	422,94	USD	747-400	385	
800	AA	0017	03.06.2009	422,94	USD	747-400	385	
800	AA	0017	01.07.2009	422,94	USD	747-400	385	
800	AA	0017	29.07.2009	422,94	USD	747-400	385	
800	AA	0017	26.08.2009	422,94	USD	747-400	385	
800	AA	0017	23.09.2009	422,94	USD	747-400	385	
800	AA	0017	21.10.2009	422,94	USD	747-400	385	

Figure 5.5 Table Fields Represented as Columns in Data Browser View

Listing 5.2 defines a structure data type using the TYPES keyword. This structure has three fields from table SFLIGHT. Defining a structure with specific fields from the database table is useful when you want to process certain field's data from the table. All the fields of the structure should have the same technical attributes as the corresponding fields of the database table.

To ensure this, refer the components of the structure in the program to the same data elements as their corresponding fields in the database table—for example, CARRID TYPE S_CARR_ID.

Here, S_CARR_ID is the data element to which the CARRID field of table SFLIGHT is referring to. You may also refer the components of the structure to the table field by using the table reference—for example, CARRID TYPE SFLIGHT-CARRID. Referring the fields to data elements or table fields also ensures that any changes in the definition for the data element or table field will automatically be reflected in the program.

In [Listing 5.2](#), we've defined structure type `TY_SFLIGHT` using the `TYPES` keyword with `CARRID`, `CONNID`, and `FLDATE` as components. We've referred the components of the structure type to respective data elements. We've also defined structure `ST_FLIGHT` using the `DATA` keyword, referring to the structure type `TY_SFLIGHT`. The example showing the declaration of structure type `TY_SFLIGHT1` uses elementary types to type the components of the structure type.

*Notice the chain operator colon(:) after the `TYPES` keyword,
 *without which you will have to use the keyword at the beginning
 *of each line and end each line with a period.

```
TYPES : BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
      END OF ty_sflight.
```

*Since the `TYPES` keyword created a data type, you can define a
 *data object referring to this structure as below.
`DATA st_sflight TYPE ty_sflight.`

*The same structure type can also be declared while referring the
 *fields to elementary types. Note that the type and length of
 *each field matches the fields in the table. Matching the type
 *and length is very important when using elementary types.

```
TYPES : BEGIN OF ty_sflight1,
        CARRID TYPE C LENGTH 3,
        CONNID TYPE N LENGTH 4,
        FLDATE TYPE D,
      END OF ty_sflight1.
```

Listing 5.2 Defining Structure Type with Set of Fields

The following shows how to define a structure type to match the ABAP Data Dictionary table:

```
TYPES : ty_sflight TYPE sflight.
```

To define a structure type with all the fields in the table, you can simply refer the structure type to the table. In this example, `ty_sflight` will have all the 14 fields of the ABAP Data Dictionary table `SFLIGHT`.

[Listing 5.3](#) shows an example of defining a nested structure.

```
* Defining nested structure type
TYPES : ty_sflight TYPE sflight.
TYPES : BEGIN OF ty_trip,
        name TYPE name1,
```

```

        flight TYPE ty_sflight,
    END OF ty_trip.

```

Listing 5.3 Nested Structure Type

In [Listing 5.3](#), we've defined structure type `TY_SFLIGHT` referring to the transparent table `SFLIGHT`. We've also defined a nested structure type `TY_TRIP` that consists of the component `NAME` and the structure type `FLIGHT`.

[Listing 5.4](#) shows an example of defining the structure as a data object using the `DATA` keyword. The syntax is similar to the `TYPES` keyword.

```

*To define a structure as a data object, simply use the
*DATA keyword instead of the TYPES keyword.
DATA : BEGIN OF st_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
    END OF st_sflight.
*You can also create a structure by referring to the
*database table to match the table row type.
DATA st_sflight1 TYPE sflight.

```

Listing 5.4 Defining Data Object as Structure

In [Listing 5.4](#), structure `ST_SFLIGHT` consists of three components, `CARRID`, `CONNID`, and `FLDATE`, whereas structure `ST_SFLIGHT1` contains all the fields of the ABAP Data Dictionary table `SFLIGHT`.

5.1.3 Global Structures

Structures types can be defined globally to have visibility across programs. Global structure types are also used to define the parameter interface for function modules or methods and also can be used to define screen elements for general screens. You'll learn more about function modules and methods in [Chapter 7](#) and more about screen programming in [Chapter 12](#).

Global structures types are created in ABAP Data Dictionary (Transaction SE11). Follow these steps to create a global structure:

1. Enter Transaction SE11.
2. Select the `DATA TYPE` radio button on the ABAP Data Dictionary initial screen. Enter the name of the structure. The name should be in a customer namespace, starting with Y or Z. Click the `CREATE` button ([Figure 5.6](#)).

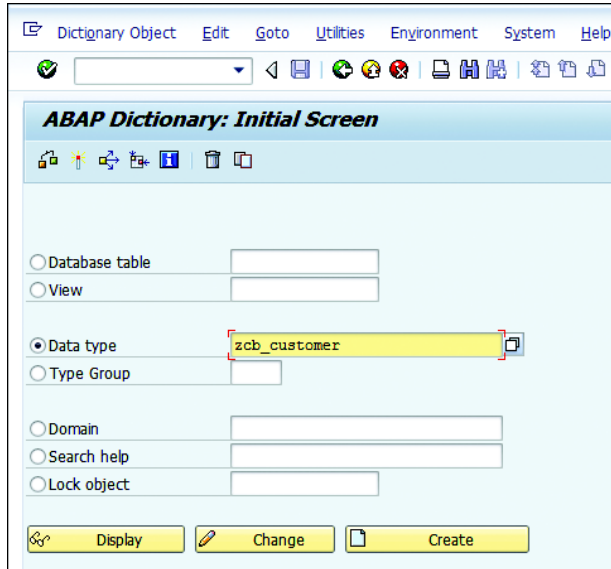


Figure 5.6 ABAP Dictionary Initial Screen

3. In the modal dialog box, select the **STRUCTURE** radio button and click the green checkmark (Figure 5.7).

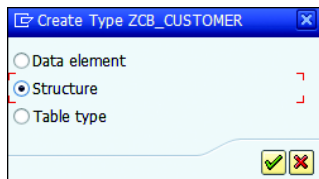


Figure 5.7 Modal Dialog Box to Select Data Type Category

4. On the next screen, enter a **SHORT DESCRIPTION** for your structure. Maintain the fields as shown in Figure 5.8. Enter the field name under the **COMPONENT** column. Select **TYPES** from the dropdown list under the **TYPING METHOD** column. Enter the data element name under the **COMPONENT TYPE** column and press **Enter**.

For **COMPONENT TYPE**, you can either use an existing data element or create your own data element, as discussed in Chapter 4. If you wish, you can use a predefined elementary type by clicking the **PREDEFINED TYPE** button.

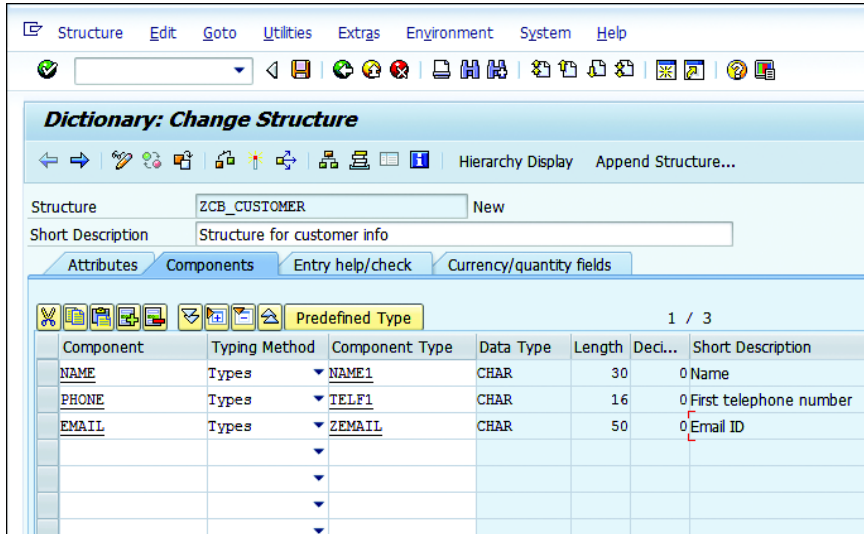


Figure 5.8 Maintaining Structure Type Components

5. Click ACTIVATE to enter the package name and save it. You can save it as a local object (\$TMP package) or save it to a transportable package.
6. Upon activation, the system may produce a warning message about a missing enhancement category; however, this warning will not prevent you from activating the structure.

SAP-delivered tables and structures in ABAP Data Dictionary can be enhanced by adding extra fields through *customizing includes* (CI Includes) or *append structures*. We will learn more about CI Includes and append structures in [Chapter 20](#) when we discuss modifications and enhancements.

The enhancement category setting specifies how a particular table or structure can be enhanced. Using this setting, you can restrict whether the structure can be enhanced by allowing only character type fields, both character and numeric fields, or deep structures.

You can maintain the enhancement category via the menu path EXTRAS • ENHANCEMENT CATEGORY ([Figure 5.9](#)). You can refer to the online documentation by using the INFORMATION button.

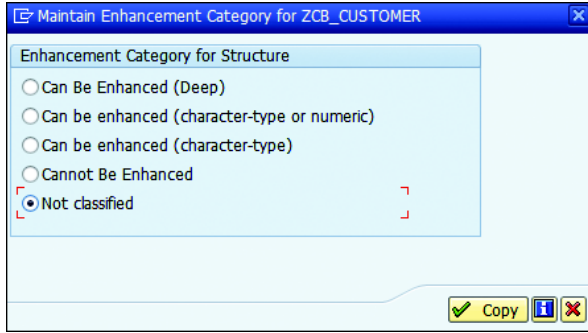


Figure 5.9 Enhancement Category

If your structure has a currency (type `CURR`) or quantity (type `QUAN`) fields, you should maintain reference fields in the `CURRENCY/QUANTITY FIELDS` tab (as shown in [Figure 5.10](#)). A currency field should refer to a currency key field (type `CUKY`) and a quantity field should refer to a `UNIT OF MEASURE` field (type `UNIT`). The reference fields can exist in the same table or another table.

Maintaining the reference fields for currency and quantity fields is mandatory, and the system uses this reference to insert the relevant currency or unit when the field is outputted. If the reference field exists in the same table, then the system can use the reference to automatically apply the correct formatting to the output. While providing the reference fields, enter the reference table name and reference field name in the respective columns.

Field	Data element	Data T...	Reference table	Ref. field	Short Description
VOLUM	VOLUM_15	QUAN	VBRP	VOLEH	Volume
VOLEH	VOLEH	UNIT			Volume unit
GSBER	GSBER	CHAR			Business Area
PRSDT	PRSDT	DATS			Date for pricing and exchange rate
FBUDA	FBUDA	DATS			Date on which services rendered
KURSK	KURSK	DEC			Exchange Rate for Price Determination
NETWR	NETWR_FP	CURR	VBRK	WAERK	Net value of the billing item in document currency
VBELV	VBELV	CHAR			Originating document
POSNV	POSNV	NUMC			Originating item

Figure 5.10 Reference Fields Maintained for Currency and Quantity Fields in Table VBRP

[Figure 5.10](#) shows the reference fields as maintained in billing document item table VBRP. The `VOLUM` field of table VBRP is of type `QUAN` and is referenced to the

field `VOLEH` of the same table `VBRP`, which is of type `UNIT`. Similarly, the `NETWR` field of table `VBRP` is of type `CURR` and is referenced to the `WAERK` field of table `VBRK`, which is of type `CUKY`.

If the reference fields aren't maintained for currency and quantity fields of your table/structure, you won't be able to activate the table or structure in ABAP Data Dictionary.

5.1.4 Working with Structures

Now that you know how to define both local and global structures, let's look at how to work with them in ABAP programs.

There are four possible ways to define a structure data object in a program:

1. Refer to a local structure type defined in the program using the `TYPES` keyword (see [Listing 5.5](#)).
2. Refer to a global structure type.
3. Refer to an ABAP Data Dictionary table.
4. Directly declare the data object using the `BEGIN OF...END OF` syntax for the `DATA` keyword.

Structures behave similarly to single fields, except for how you access the components of a structure. To access the field of a structure, you need to give the structure name field reference, as shown in the example in [Listing 5.5](#). Note that structures can store only one row at a time. If you store a new record, it will replace the existing record.

The example code in [Listing 5.5](#) defines a flat structure `TY_SFLIGHT` and a nested structure `TY_TRIP`. Corresponding data objects `ST_SFLIGHT` and `ST_TRIP` are defined that refer to these structures respectively. The comment lines in the code explain how the components of the structures are accessed using ABAP statements.

```
TYPES : BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
        END OF ty_sflight.
TYPES : BEGIN OF ty_trip,
        name TYPE name1,
```

```

        flight type ty_sflight,
    END OF ty_trip.

DATA : st_sflight TYPE ty_sflight,
      st_trip TYPE ty_trip.

*Addressing to the field of a structure type
DATA v_connid TYPE ty_sflight-connid.

*Accessing field of a structure data object
st_sflight-carrid = 'AA'.
st_sflight-connid = '04'.
WRITE st_sflight-connid.

*Accessing field of a nested structure
st_trip-flight-carrid = 'AB'.

```

Listing 5.5 Accessing Fields of Structure

5.1.5 Use Cases

Structures should be used in the following cases:

- ▶ Use structures to process multiple fields of data. For example, when you want to process a record, you can define a structure with the fields of the record as components and process the record as a single entity.
- ▶ Group related fields into a structure if you need to perform certain actions on the fields together. It's easy to clear the data in all the fields of a structure at once instead of individually clearing the data for each field.

For example, if you have a structure `ST_SFLIGHT` with all the components of the ABAP Data Dictionary table `SFLIGHT`, you can clear a record easily by using the statement `CLEAR ST_SFLIGHT` rather than using the `CLEAR` statement for each individual field.

- ▶ Even though it's possible to select data from a database table row into individual fields, it's always recommended to use structures when processing data from database tables.
- ▶ Structure types also help you keep the parameter interface (signature) of a procedure (such as function module or method) simple by grouping relevant fields into one structure, as opposed to maintaining all fields individually in the interface.

5.2 Internal Tables

Internal tables are dynamic data objects that can store multiple rows of data with a common row type. In fact, internal tables are the only data objects that can store multiple rows of data at a time. We use internal tables when we need to work with mass data. Internal tables can be compared to two-dimensional arrays.

Working with internal tables is a bit different from other data objects. In this section, we'll look at how to define, store, and access data with internal tables. We'll discuss how to define internal types and the different types of internal tables available. We'll then provide steps for working with internal tables before discussing the use of control break statements.

5.2.1 Defining Internal Tables

[Listing 5.6](#) shows the basic syntax for declaring an internal table. Here, the data object `IT_SFLIGHT` is declared as an internal table with the line type as `TY_SFLIGHT`. The *line type* of a table specifies the fields that make up a row. The `TABLE OF` addition to the type specification defines an internal table.

```
TYPES : BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
        END OF ty_sflight.
DATA : it_sflight TYPE TABLE OF ty_sflight.
```

Listing 5.6 Declaring Internal Table

Because an internal table consists of multiple rows, you can't work with the data of an individual row directly. To process the data from an internal table, you need a structure data object, called the work area. The *work area* of an internal table is a structure that can store one record at a time and has the same *Line type* as the internal table. This allows you to read an individual row from an internal table into the work area and process it.

[Listing 5.7](#) shows the work area `wa_sflight` declaration, referring to the same *Line type* as the internal table `IT_SFLIGHT`.

```
TYPES : BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
```

```

        FLDATE TYPE S_DATE,
    END OF ty_sflight.
DATA : it_sflight TYPE TABLE OF ty_sflight,
      wa_sflight TYPE ty_sflight.

```

Listing 5.7 Defining Internal Table Work Area

Figure 5.11 depicts the usage of a work area. Because the work area can store only one record, you can read a single row from the internal table or loop through all rows of the table one row at a time.

To read a single record from the internal table, use the `READ` statement, and to process more than one record, use the `LOOP` statement. The `READ` and `LOOP` statements copy the data from the internal table to the work area.

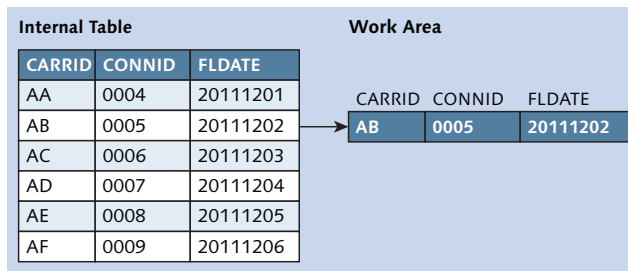


Figure 5.11 Internal Table and Work Area

There are a couple of ways to access a single record from an internal table via the `READ` statement. You can use an index (index access) or you can read the table by comparing the value of a field (or a set of fields) of the row (key access). Please note that not all internal tables support index access, which we will discuss in the next section.

Listing 5.8 shows example code to read an internal table using index access.

```

TYPES : BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
    END OF ty_sflight.
DATA : it_sflight TYPE TABLE OF ty_sflight,
      wa_sflight TYPE ty_sflight.
SELECT carrid connid fldate FROM sflight INTO TABLE it_sflight.

```

*Reading a single record from the internal table into work area

```
*using index access
READ TABLE it_sflight INTO wa_sflight INDEX 2.
```

Listing 5.8 Using Index Access

Listing 5.8 defines a structure type `TY_SFLIGHT` and an internal table `IT_SFLIGHT` using the `TABLE OF` addition to the `DATA` keyword. The line type of the internal table `IT_SFLIGHT` is defined as the structure type `TY_SFLIGHT`. We used a `SELECT` statement to fetch the data from database table `SFLIGHT` into the internal table `IT_SFLIGHT`.

Because the internal table can store multiple records, the system will transfer all the records from the database table `SFLIGHT` into the internal table `IT_SFLIGHT`. A `SELECT` statement should always be restricted using a `WHERE` clause, but we ignored that rule in this example for simplicity. You'll learn more about Open SQL statements in [Section 5.3](#).

In Listing 5.8, we're reading a single record from the internal table using the `READ` statement and specifying the index using the `INDEX` addition. In this example, the second record of the internal table will be copied to the work area.

To read an internal table record using *key access*, use the addition `WITH KEY` and list the components of the table to be matched with a value using the `=` operator as shown in the example ahead. You can list multiple components separated by a space:

```
READ TABLE it_sflight INTO wa_sflight WITH KEY connid = 'AA'.
```

To process all the records of an internal table one row at a time, use the `LOOP` statement:

```
LOOP AT it_sflight INTO wa_sflight.
ENDLOOP.
```

We can restrict the loop to process specific records using the index:

```
LOOP AT it_sflight INTO wa_sflight FROM 1 to 5.
ENDLOOP.
```

We can also restrict the loop using a key:

```
LOOP AT it_sflight INTO wa_sflight WHERE carrid EQ 'AA'.
ENDLOOP.
```

5.2.2 Types of Internal Tables

An internal table is defined by its row type, table category, and table key. We will walk through each of these elements in the following subsections.

Row Type

A *row type* defines the row of an internal table (the fields that make up the row of a table). Any type can be used as a row type, such as an elementary type, structured type, table type, or reference type. For example, an internal table can consist of fields (known as elementary types), a structure, a table type, or even a reference object (such as an ABAP class).

Table Category

There are three types of internal tables that can be defined in ABAP programs: standard, sorted, and hashed. The *table category* defines the type of internal table based on how individual rows can be accessed.

In the following subsections, we'll take a close look at the different types of table categories that can be defined.

Standard Tables

Standard tables are managed using a row index. For every record that you add to an internal table, the system creates a linear index. New rows can be either appended to the end of the table using the `APPEND` keyword (e.g., `APPEND wa_sflight TO it_sflight`) or inserted at specific positions using the `INSERT` keyword (e.g., `INSERT wa_sflight INTO it_sflight INDEX 3`). Individual rows are accessed using the table key or the row index; standard table support both key access and index access.

You are not required to explicitly specify the primary table key during table declaration. The primary key can be empty for standard tables. Standard tables only support non-unique primary keys, which means that multiple rows in the same table can have the same key value.

When using the `TABLE KEY` addition with the `READ` statement, the system performs a linear search to identify the matching row.

The syntax for declaring a standard table is as follows:

```
DATA internaltable_name TYPE TABLE OF [or STANDARD TABLE OF] structure_
reference
```

The `TYPE TABLE OF` and `TYPE STANDARD TABLE OF` additions both define standard tables. `internaltable_name` is the name of the internal table, and `structure_reference` is a reference to the structure type.

In the statement

```
DATA it_sflight TYPE TABLE OF ty_sflight.
```

the internal table `IT_SFLIGHT` is defined as a standard table with the line type `TY_SFLIGHT`. Here, `TY_SFLIGHT` is a structure type. The `TABLE OF` addition is a short form to define a standard table. We can also use the long form `STANDARD TABLE OF` to define the standard table, as shown:

```
DATA it_sflight TYPE STANDARD TABLE OF ty_sflight.
```

Sorted Tables

Sorted tables are similar to standard tables, except the records are always sorted by the key in ascending order. New records can be inserted or appended only if they are sorted. Like standard tables, sorted tables are also managed via a row index, so you can read the individual rows by index or key.

Sorted tables support both unique and non-unique primary keys, and defining the primary key is mandatory when declaring the internal table. If the table has a unique key, the key can't be repeated. A table with a non-unique key can contain multiple records for the key. In other words, for a unique key, you can't have multiple records with the same value for the key fields.

For example, if the key of the table is defined as `CARRID` and `CONNID` and the table contains a record in which `CARRID` has the value 'AA' and `CONNID` has the value '0017', then you can't have a second record in the table with the same values for `CARRID` and `CONNID`. The individual fields that form the key can repeat themselves, but the combination of the key fields should be unique. Therefore, we can have another record with `CARRID` as 'AA' and a different record with `CONNID` as '0017', but we can't have another single record that contains `CARRID` as 'AA' and `CONNID` as '0017'.

When you define a table with a non-unique key, you can have multiple records with the same values for the key and no restrictions apply. You should choose the key carefully based on the data being processed to avoid data inconsistency.

When using the `TABLE KEY` addition with the `READ` statement, the system performs a binary search to identify the matching row.

The syntax for declaring a sorted table is as follows:

```
DATA internaltable_name TYPE SORTED TABLE OF structure_
reference WITH UNIQUE [or NON-UNIQUE] KEY key_fields.
```

To define a sorted table, use the `SORTED TABLE OF` addition. It's mandatory to define the key of the table during a declaration. The key is defined using the `WITH UNIQUE KEY` or `WITH NON-UNIQUE KEY` addition to define a unique or non-unique key, respectively. The key of the sorted table can consist of a single field or multiple fields. If the key contains multiple fields, each field is listed after the `WITH UNIQUE` (or `NON-UNIQUE`) `KEY` addition, separated by a space.

For example, the following statement defines a sorted table `IT_SFLIGHT` with line type `TY_SFLIGHT` consisting of unique key `CARRID` and `CONNID` (assuming `CARRID` and `CONNID` are components of structure type `TY_SFLIGHT`):

```
DATA it_sflight TYPE SORTED TABLE OF ty_
sflight WITH UNIQUE KEY carrid connid.
```

In addition, the following statement defines a sorted table `IT_SFLIGHT` with non-unique key `CARRID` and `CONNID`:

```
DATA it_sflight TYPE SORTED TABLE OF ty_sflight WITH NON-
UNIQUE KEY carrid connid.
```

Hashed Tables

Unlike standard and sorted tables, *hashed tables* do not have a row index. They are managed using a *hash algorithm*. Because there is no row index, you can't perform any index operations and can only read the individual rows by the key. Hashed tables support only unique keys, and defining the key is mandatory while declaring the internal table.

When using the `TABLE KEY` addition with the `READ` statement, the system doesn't compare the contents of the key fields manually to identify the matching row. Instead, the system uses hash functions to identify the matching row.

The syntax for declaring hashed table is as follows:

```
DATA internaltable_name TYPE HASHED TABLE OF structure_
reference WITH UNIQUE KEY key_fields.
```

Use the HASHED TABLE OF addition to define a hashed table and list the table key using the WITH UNIQUE KEY addition. The following statement defines an internal table IT_SFLIGHT as a hashed table with unique key CARRID and CONNID:

```
DATA it_sflight TYPE HASHED TABLE OF ty_
sflight WITH UNIQUE KEY carrid connid.
```

[Table 5.1](#) lists some of the key properties of the table categories.

Standard Table	Sorted Table	Hashed Table
<ul style="list-style-type: none"> ▶ Contains row index. ▶ Accessed using key and index. ▶ Only non-unique primary key. ▶ Linear search. ▶ Lookup time is directly proportional to the number of entries in the table. ▶ Primary key can be empty. ▶ Rows are not sorted implicitly but can be sorted explicitly. 	<ul style="list-style-type: none"> ▶ Contains row index. ▶ Accessed using key and index. ▶ Both unique and non-unique primary key. ▶ Binary search. ▶ Lookup time is directly proportional to the number of entries in the table but faster than linear search. ▶ Key is mandatory. ▶ Rows are always sorted by key in ascending order by default and no manual sort operations are allowed. 	<ul style="list-style-type: none"> ▶ No row index. ▶ No INDEX operations allowed; only key access. ▶ Only unique primary key. ▶ Hash algorithm. ▶ Lookup time is always constant irrespective of the number of entries. ▶ Key is mandatory. ▶ Rows are not sorted implicitly but can be sorted explicitly.

Table 5.1 Key Properties of Internal Tables

Any Tables and Index Tables

In addition to the standard table, sorted table, and hashed table categories, there are two other generic table categories—namely, any table and index table. The generic table categories are used when typing formal parameters of a procedure and field symbols.

The *any table* category covers all table categories; the *index table* category only covers those table categories in which index operations are possible (standard tables and sorted tables). The objects referred to generic table types don't contain *any fields* statically during design time; instead, any fields defined during runtime are based on the definition of the parent object.

The following example shows a field symbol `<fs_table>` defined as a generic type:

```
FIELD-SYMBOLS <fs_table> TYPE ANY TABLE.
```

Because the field symbol `<fs_table>` is defined as the generic type `ANY TABLE`, you can assign any internal table (be it standard, sorted, or hashed) to the field symbol at runtime. The field symbol will then point to the assigned table.

Field symbols are an important concept in dynamic programming; we will explore them further in [Chapter 16](#).

5.2.3 Table Keys

Internal tables have a primary table key and can also support secondary table keys. A *table key* can consist of a list of structure components with a certain sequence or can use `table_line` as a pseudo-component to specify the entire table row as a key, as shown in the following example:

```
DATA it_sflight TYPE HASHED TABLE OF ty_sflight WITH UNIQUE key TABLE_
LINE.
```

You can also list the individual components of the table as the table key:

```
DATA it_sflight TYPE HASHED TABLE OF ty_
sflight WITH UNIQUE key CARRID CONNID.
```

You can also specify the standard key's *default key*. This key contains all components that are not numeric and are not themselves table types, as shown:

```
DATA it_sflight TYPE HASHED TABLE OF ty_
sflight WITH UNIQUE DEFAULT KEY.
```

Support for secondary table keys was introduced with SAP NetWeaver 7.0 EHP 2. Previously, each internal table had just one table key (the primary key). When reading an internal table, you can use either a table key or a free key; that is, you can either read the table using the complete key defined for the table (table key) or read the table using any of the fields of the table (free key).

When a table key is used to read the table, the system performs a linear search for standard tables, a binary search for sorted tables, and uses hash functions for hashed tables. However, when using a free key, the system always performs a linear search irrespective of the type of table, thus degrading the performance.

Because using a table key is very restrictive, many programmers use a free key and compromise on the performance. To overcome this problem, SAP introduced secondary keys. You can define secondary keys with different names and use them while processing the internal table. This provides the flexibility of using different keys while ensuring efficiency.

Along with the primary key, an internal table can have up to fifteen secondary table keys with different names. With the introduction of secondary keys, what was previously the *table key* has now become the *primary table key* and a predefined name for it, `primary_key`, has been introduced. Secondary table keys can be hash keys or sorted keys. *Hash keys* must be unique, whereas *sorted keys* can be unique or non-unique. The system creates a secondary table index for each sorted secondary key of an internal table.

[Listing 5.9](#) shows the syntax to define a standard table with a non-unique sorted secondary key and unique hashed key.

```
DATA it_sfflight TYPE STANDARD TABLE OF ty_sfflight
    WITH NON-UNIQUE DEFAULT KEY
    WITH NON-UNIQUE sorted KEY by_airline COMPONENTS carrid
    WITH UNIQUE HASHED KEY by_flight COMPONENTS carrid connid.
```

Listing 5.9 Secondary Keys with Standard Table

Note

Note that defining the primary and secondary keys is completely optional for standard tables.

In [Listing 5.9](#), we defined the internal table `IT_SFLIGHT` as a standard table with a non-unique default primary key that contains two secondary keys, `BY_AIRLINE` and `BY_FLIGHT`. The secondary key `BY_AIRLINE` is defined as a non-unique sorted key with `CARRID` as the key component. The secondary key `BY_FLIGHT` is defined as a unique hash key with `CARRID` and `CONNID` as the key components.

The syntax to use a secondary key in a `READ` statement is shown in the following example; here, `key_name` signifies the name of the secondary key:

```
WITH KEY[or WITH TABLE KEY] key_name COMPONENTS comp1...
```

To read an internal table using the secondary key, specify the name of the secondary key in the `READ` statement, as shown in [Listing 5.10](#).

```
*Using the secondary key by_airline to read the internal table.
READ TABLE it_sflight INTO wa_sflight WITH TABLE KEY by_airline
COMPONENTS carrid = 'AA'.
```

Listing 5.10 Reading Internal Table Using Secondary Key

In [Listing 5.10](#), assuming the internal table `IT_SFLIGHT` is defined with a secondary key `BY_AIRLINE`, we're using the `READ` statement and specifying the name of the secondary key to read the record of the table using the secondary key.

In [Listing 5.11](#), we've defined an internal table `IT_SFLIGHT` as a sorted table using the pseudo-component `table_line` as the unique primary key. Two secondary keys, `BY_AIRLINE` and `BY_FLIGHT`, are also defined for the table. The secondary key `BY_AIRLINE` is defined as a non-unique sorted key, and the secondary key `BY_FLIGHT` is defined as a unique hashed key.

```
DATA it_sflight TYPE SORTED TABLE OF ty_sflight
    WITH UNIQUE KEY table_line
    WITH NON-UNIQUE SORTED KEY by_airline COMPONENTS carrid
    WITH UNIQUE HASHED KEY by_flight COMPONENTS carrid connid.
```

Listing 5.11 Secondary Keys for Sorted Table

[Listing 5.12](#) shows an example to define secondary keys for hashed table.

```
DATA it_sflight TYPE HASHED TABLE OF ty_sflight
    WITH UNIQUE KEY table_line
    WITH NON-UNIQUE SORTED KEY by_airline COMPONENTS carrid
    WITH UNIQUE HASHED KEY by_flight COMPONENTS carrid connid.
```

Listing 5.12 Secondary Keys for Hashed Table

In [Listing 5.12](#), we defined an internal table `IT_SFLIGHT` as a hashed table using the pseudo-component `table_line` as the unique primary key. Two secondary keys, `BY_AIRLINE` and `BY_FLIGHT`, are defined for the table. The secondary key `BY_AIRLINE` is defined as a non-unique sorted key, and the secondary key `BY_FLIGHT` is defined as a unique hashed key.

Before secondary keys were available, the only way to have the flexibility to read a table by any key while ensuring performance was to sort a standard table by the key and read it using the `BINARY` search addition.

Listing 5.13 makes the standard table behave like a sorted table with `CONNID` as the non-unique primary key.

```
SORT it_sflight BY connid.
READ TABLE it_sflight INTO wa_sflight WITH KEY connid BINARY SEARCH.
```

Listing 5.13 Performing Binary Search on Standard Table

Secondary keys provide a lot more flexibility. You can use different keys and improve performance while accessing individual rows of a table. However, the ABAP runtime system needs to administer each additional secondary key, so you should use them carefully.

You should define a secondary key for an internal table only when the required key or table index is used in processing statements.

Be mindful of the following considerations when using secondary keys:

- ▶ Use secondary keys for large internal tables with infrequently changed content. This ensures that the costs for administrating the secondary keys are only incurred when setting up the internal table initially with the data.
- ▶ Always avoid secondary hash keys with too many components. If the hash key has too many components, the system has to deal with a high system load for additional hash management. Use sorted keys for secondary keys with a large number of components.
- ▶ If you have secondary keys for an index table, the primary index of the table is updated immediately when you insert or delete rows using a secondary key. Because the standard table uses linear search, deleting a row from a standard table using a secondary key can't be optimized.
- ▶ You can use secondary keys for small internal tables in some circumstances if you want to ensure unique table entries in relation to particular components. Primary keys don't allow this, especially in the case of standard tables.
- ▶ For pure read access, non-unique secondary keys are as quick as unique keys. Use non-unique secondary keys if unique table entries are not required. Non-unique secondary keys perform better while updating a table and use a lazy update. With a *lazy update*, secondary keys aren't updated immediately after a record is inserted or modified in the table; they're updated when the secondary key is used explicitly the next time after the records were updated or modified in the table.

- ▶ You should not use secondary table keys in the following situations:
 - ▶ For small internal tables (less than fifty rows). The performance gain for read accesses is much less compared to the increased storage and administration load.
 - ▶ With tables that are updated or modified frequently. Each time the table is modified, the secondary keys need to be updated. The loads incurred by updating the secondary keys outweighs the performance benefit for read accesses. Also, with delayed updates and lazy updates in particular, the system may perform the update during a read access, affecting the performance of the `READ` statement. This defeats the purpose of using secondary keys, because the main purpose of secondary keys is to optimize read access.
 - ▶ Avoid using secondary hash keys unless you want to ensure the uniqueness of table entries.

5.2.4 Working with Internal Tables

Now that you know how to define internal tables, let's discuss how to perform various actions with them. As discussed earlier, we need to use a work area to add a record to internal table or to read a record from an internal table. In the following subsections, we'll discuss the different ways we can work with internal tables.

APPEND and INSERT Statements

To add a record to an internal table, use the `APPEND` or `INSERT` statement. An `APPEND` statement always adds the record at the end of the table and is used with standard tables. An `INSERT` statement behaves differently based on the table type. For example, an `INSERT` statement behaves like an `APPEND` statement for standard tables in that it adds a record at the end of the table if no explicit index is specified.

However, with sorted tables, the `INSERT` statement automatically selects the position based on the sort sequence of the table. You need to use caution when using an `APPEND` statement with sorted tables; the system may produce a runtime error if the new row does not match the sort sequence of the table or if the table is defined with a unique key and the new row creates a duplicate entry. For this reason, we recommend always using the `INSERT` statement with sorted tables. You

can't use `APPEND` statements with hashed tables, because they only support `INSERT` statements.

[Listing 5.14](#) shows an example of using the `APPEND` statement to add records to a standard table. To add a record to an internal table, first fill the work area with the required data and then append it to the table using the `APPEND` statement.

```
*Adding a record to internal table
TYPES : BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
      END OF ty_sflight.
DATA : it_sflight TYPE STANDARD TABLE OF ty_sflight,
      wa_sflight TYPE ty_sflight.
```

*You can't refer to the table row directly to add a record.

*Fill the work area with the values and add it to the table.

```
wa_sflight-carrid = 'AA'.
```

```
wa_sflight-connid = '0004'.
```

```
wa_sflight-fldate = '20140101'.
```

```
APPEND wa_sflight TO it_sflight. "Adds the first row to the table
```

```
wa_sflight-carrid = 'AB'.
```

```
wa_sflight-connid = '0005'.
```

```
wa_sflight-fldate = '20140102'.
```

```
APPEND wa_sflight TO it_sflight. "Adds second row to the table
```

Listing 5.14 Using `APPEND` Statement to Add Records to Internal Table

[Listing 5.15](#) shows an example using the `INSERT` statement. Notice the difference in the syntax compared to the `APPEND` statement. The listing shows the usage of the `INSERT` statement with and without an explicit index. When an index is supplied with the `INSERT` statement, the record is added explicitly in that row.

If the index supplied is greater than the line of the table, no record is inserted and the system field `sy-subrc` is set to 4. Refer to the box ahead for an explanation of system fields.

*The following code inserts the record as last row

```
wa_sflight-carrid = 'AC'.
```

```
wa_sflight-connid = '0006'.
```

```
wa_sflight-fldate = '20140103'.
```

```
INSERT wa_sflight INTO TABLE it_sflight.
```

*The follow code inserts the record at a specific position based on

*the index.

```
wa_sflight-carrid = 'AD'.
```



```

wa_sflight-connid = '0007'.
wa_sflight-fldate = '20140104'.
INSERT wa_sflight INTO it_sflight INDEX 1

```

Listing 5.15 Adding Records Using INSERT Statement

The syntax to use the `INSERT` statement is the same for all table types. However, index operations can't be performed on hash keys.

System Fields

System fields are always available in ABAP programs and they are filled by the runtime system based on the context. You query the system fields in the program to identify the status. For example, you can query the system field `subrc` to learn whether an `INSERT` operation was successful or whether a `SELECT` statement could find a matching record in the database.

System fields are available as variables in the program. However, they should be treated like constants and should never be manipulated. All system fields are available in the ABAP Data Dictionary structure `SYST` and can be accessed in the program using the pre-defined structure `SY`. For example, to access the system field `subrc` in an ABAP program, use the structure reference `sy` followed by the field name separated by a hyphen, as in `sy-subrc`.

To refer your program data types and data objects to system fields, use the `SYST` structure reference. For example, `DATA lv_index TYPE syst-index`.

Often, you'll perform an `INITIAL` check on system fields to see if a statement was successful, as shown in [Listing 5.16](#).

```

READ TABLE it_sflight INTO wa_sflight WITH KEY CARRID = 'AA'.
IF sy-subrc IS INITIAL.
*****
ELSE.
*****
ENDIF.

```

Listing 5.16 System Fields

This example checks if the `READ` statement could successfully find a matching record and accordingly build the program logic. In this case, `sy-subrc` will have its initial value if the `READ` statement was successful; otherwise, it will have a value other than its initial value.

Each keyword updates different system fields with different values based on the context. The keyword documentation helps you understand the system field values set by different keywords. To open the keyword documentation, simply press `F1` in the ABAP Editor.

For more information on various system fields, visit SAP Help at https://help.sap.com/saphelp_nw70/helpdata/en/7b/fb96c8882811d295a90000e8353423/content.htm.

A system field check should immediately follow the statement for which the status is queried, because other statements also may update the system fields.

COLLECT Statement

There is one more statement available to add records to an internal table: the `COLLECT` statement. It behaves a bit differently from the `APPEND` and `INSERT` statements. When you try to add a row with a `COLLECT` statement, the system checks if the nonnumeric fields of the row exist in the table.

If the nonnumeric fields match, it will not create a new row but simply adds the numeric fields of the row. If the nonnumeric fields don't match, then it creates a new row.

This statement is useful when you need to consolidate data, such as in a super-market billing application in which the currency and quantity fields are updated when an existing item in the bill is scanned multiple times rather than creating a new entry each time.

Listing 5.17 shows an example of using a `COLLECT` statement. In this example, a new entry is added in the table for each new item. However, if the item already exists in the table, then only the quantity and amount fields are updated.

```
TYPES : BEGIN OF ty_bill,
        item TYPE c LENGTH 20, "Item name
        qty  TYPE i,           " Quantity
        amt  TYPE p DECIMALS 2,"Amount
      END OF ty_bill.
DATA : it_bill TYPE STANDARD TABLE OF ty_bill,
      wa_bill TYPE ty_bill.

wa_bill-item = 'soap'.
wa_bill-qty = 1.
wa_bill-amt = 10.
COLLECT wa_bill INTO it_bill. " Creates a new entry

wa_bill-item = 'brush'.
wa_bill-qty = 1.
wa_bill-amt = 20.
COLLECT wa_bill INTO it_bill. " Creates a new entry
```

```

wa_bill-item = 'soap'.
wa_bill-qty = 1.
wa_bill-amt = 10.
COLLECT wa_bill INTO it_bill. "Adds the qty and amt
*fields of the first record instead of creating a new record.

```

Listing 5.17 COLLECT Statement

Reading Internal Table Data

You can't access the records of the internal table directly by addressing the name of the internal table; you use the `READ` and `LOOP` statements to process the internal table records. To read a single record from the internal table you use the `READ` statement, and to read more than one record, you use the `LOOP` statement.

To read the internal table, you can use a `TABLE KEY` or `FREE KEY`. The `TABLE KEY` compares all the key fields and automatically performs a linear search for standard tables, a binary search for sorted tables, and uses the hash algorithm for hashed tables.

If you use `FREE KEY` instead of `TABLE KEY`, you can compare any field of the table and the system will always perform a linear search irrespective of the type of table. A `WITH KEY` portion of the `READ` statement allows you to use a free key, while a `WITH TABLE KEY` portion expects you to provide the table key.

When using the table key, you should specify the fields in the same order as the key. [Listing 5.18](#) shows the syntax to use the `READ` statement.

```

*Read statement using free key.
READ TABLE it_sflight INTO wa_sflight WITH KEY connid = '0004'.

*READ statement using table key. Assuming the key is defined as
*carrid and connid
READ TABLE it_sflight INTO wa_sflight WITH TABLE KEY carrid =
'AA' connid = '0004'.

*READ statement using the index. Note that index operations can
*be performed only on standard and sorted tables.
READ TABLE it_sflight INTO wa_sflight INDEX 3.

*Assuming a secondary key is defined as "by_airline"
READ TABLE it_sflight INTO wa_sflight INDEX 3 USING KEY by_airline.

```

Listing 5.18 Syntax to Use `READ` Statement

Listing 5.18 demonstrates using the `READ` statement with a free key, table key, and index. When reading an internal table record using an index, the system uses the index of the primary key by default.

If you want to use the index of the secondary key, you can explicitly specify the secondary key name via the `USING KEY` addition. If you specify a primary key after the `USING KEY` addition, then the behavior is similar to using the `READ` statement without the `USING KEY` addition.

The `LOOP` statement can be used to process multiple rows. The system loops between `LOOP` and `ENDLOOP`, reading one row at a time to the work area.

Listing 5.19 shows the syntax to use the `LOOP` statement.

```
*Looping all the records of the internal table
LOOP AT it_sflight INTO wa_sflight.
*Statements to process the row can go here
ENDLOOP.
*Looping specific records of the internal table using key
LOOP AT it_sflight INTO wa_sflight WHERE connid EQ '0004'.

ENDLOOP.
*Looping specific records using secondary key. Assuming a
*secondary key "by_airline" is defined with carrid and connid as
*components.
LOOP AT it_sflight INTO wa_sflight
      USING KEY by_airline
      WHERE carrid EQ 'AA'
      AND connid EQ '0004'.

ENDLOOP.
*Looping specific records using index.
LOOP AT it_sflight INTO wa_sflight FROM 3 TO 5.

ENDLOOP.

*Looping specific records using secondary key index. Note that
*the secondary key should be a sorted key and not hashed key
LOOP AT it_sflight INTO wa_sflight USING KEY by_airline
      FROM 3 TO 5.

ENDLOOP.
```

Listing 5.19 Using `LOOP` Statement

Listing 5.19 demonstrates different scenarios to use the `LOOP` statement to process the internal table records. You can loop all the records of the internal table or

restrict the loop to certain records. The loop can be restricted either by key or index; you can restrict the loop with a key via the `WHERE` clause.

Modifying Records of an Internal Table

In certain scenarios, you may want to modify the records of the internal table. For example, if you're fetching data about a sales order from a database table to an internal table and displaying it on the screen for the user to edit the sales order, then once the user makes any changes to the records on the screen, you may want to modify the corresponding record in the internal table and update it back to the database table.

The `MODIFY` keyword can be used to modify a particular row(s) of an internal table. You need to be careful while using the `MODIFY` statement; it will result in a runtime error if the key doesn't match. You can only modify the non-key fields and can't change an active key.

Listing 5.20 shows an example of the `MODIFY` statement using both an index and a key.

```
*To modify using index
MODIFY it_sflight FROM wa_sflight INDEX 3.
```

```
*To modify using key
MODIFY TABLE it_sflight FROM wa_sflight.
```

Listing 5.20 Performing Modification Operations

When the index is provided, the corresponding row is modified by copying the contents of the work area to the internal table. When using the key, the system scans for the corresponding record in the table and modifies it by copying the contents of the work area to the internal table.

If you want to restrict the fields of the row that are modified, you can use the `TRANSPORTING` addition, as shown ahead. The `TRANSPORTING` addition transports only the specified fields from the work area to the internal table:

```
MODIFY TABLE it_sflight FROM wa_sflight TRANSPORTING carrid.
```

To delete a row from an internal table, you can use the `DELETE` keyword:

```
DELETE it_sflight INDEX 3.
DELETE it_sflight WHERE connid EQ 'AA'.
CLEAR it_sflight. "To delete all rows
```

5.2.5 Control Break Statements

While processing data in a loop, special control-level processing can be performed via control break statements. The statement block `LOOP` can contain control structures that trigger the control break statements when the control structure is changed.

For example, a control break statement is triggered when the first record is processed in a loop, when a field value in the control structure changes, or when the last record of a loop is processed. Control break statements are introduced with `AT` and closed with `ENDAT`.

There are four control break statements available:

- ▶ **AT FIRST**
Triggered in the first loop pass.
- ▶ **AT NEW comp**
Triggered when the value of any field up to `comp` in the row changes, compared to the previous record.
- ▶ **AT END OF comp**
Triggered when the value of any field up to `comp` in the row changes, compared to the next record.
- ▶ **AT LAST**
Triggered in the last loop pass.

Assuming internal table `IT_SFLIGHT` has three components (`CARRID`, `CONNID`, and `FLDATE`—in that order), [Listing 5.21](#) shows the syntax of using various control break statements and why they trigger.

```

LOOP AT it_sflight INTO wa_sflight.
AT FIRST.
*The code here will be executed only in first loop pass.
ENDAT.
AT NEW carrid.
*The code here will be executed if the value of the field carrid
*is different from previous loop pass.
ENDAT.
AT NEW connid.
*The code here will be executed if the value of carrid or connid
*is different from previous loop pass. It not only checks the
*field mentioned in the control structure but checks all
*the fields to the left-hand side in the row.
ENDAT.

```

```

AT END OF connid.
*The code here will be executed if the value of either carrid or
*connid will be different in the next loop pass.
ENDAT.
AT LAST.
*The code here will be executed only in last loop pass.
ENDAT.
ENDLOOP

```

Listing 5.21 Control Break Statements

Control break statements can be maintained in any order in the loop, because they're always triggered on the control break and not by the order in which they are maintained in the code.

The code sample in [Listing 5.22](#) shows an example of some control break statements. The code in this example prints the header and item details of a billing document, highlighting how control break statements can be used to control the report output.

```

*Billing document header structure
TYPES: BEGIN OF TY_VBRK,
        VBELN TYPE VBELN_VF, "Document Number
        FKDAT TYPE FKDAT, "Bill Date
        NETWR TYPE NETWR, "Net Value
        KUNRG TYPE KUNRG, "Payer
END OF TY_VBRK.

*Billing document item structure
TYPES: BEGIN OF TY_VBRP,
        VBELN TYPE VBELN_VF, "Document Number
        POSNR TYPE POSNR_VF, "Item Number
        ARKTX TYPE ARKTX, "Description
        FKIMG TYPE FKIMG, "Qty
        VRKME TYPE VRKME, "Sales Unit
        NETWR TYPE NETWR_FP, "Net Value
        MATNR TYPE MATNR, "Material Num
        MWSBP TYPE MWSBP, "Tax amt
END OF TY_VBRP.

DATA: it_vbrp TYPE TABLE OF ty_vbrp,
      it_vbrk TYPE TABLE OF ty_vbrk,
      wa_vbrk TYPE ty_vbrk,
      wa_vbrp TYPE ty_vbrp.

*Selection screen declaration for user selection criteria
SELECT-OPTIONS: s_vbeln FOR wa_vbrp-vbeln.

```

```

START-OF-SELECTION.
*Select Header Data
SELECT VBELN FKDAT NETWR KUNRG FROM VBRK
                                INTO TABLE it_vbrk
                                WHERE VBELN in s_matnr.

"Select Item Data
SELECT VBELN POSNR ARKTX FKIMG VRKME NETWR MATNR MWSBP
      FROM vbrp INTO TABLE it_vbrp WHERE vbeln in s_vbeln.

END-OF-SELECTION
*The SORT statement ensures the control break is consistent if
*the table is not a sorted table.
SORT it_vbrp BY vbeln.
LOOP AT it_vbrp INTO wa_vbrp.
  AT FIRST. "Print Column Headings
    WRITE AT: /05 'ITEM',
              15 'DESCRIPTION',
              60 'BILLED QUANTITY',
              80 'UNITS',
              105 'NET VALUE',
              130 'MATERIAL NUMBER',
              150 'TAX AMOUNT'.

    WRITE SY-ULINE.
  ENDAT.

  AT NEW VBELN. " To print header data once per new document
    READ TABLE it_vbrk into wa_vbrk WITH KEY
                                VBELN = wa_vbrp-vbeln.

    WRITE AT : /5 'Billing Document' LEFT-JUSTIFIED.
    WRITE AT : 30 wa_vbrk-vbeln LEFT-JUSTIFIED COLOR 2.
    WRITE AT : /5 'Payer' LEFT-JUSTIFIED.
    WRITE AT : 30 wa_vbrk-kunrg LEFT-JUSTIFIED COLOR 3.
    WRITE AT: /5 'BILLING DATE' LEFT-JUSTIFIED.
    WRITE AT: 30 wa_vbrk-fkdat LEFT-JUSTIFIED COLOR 5.
    WRITE AT: /5 'NET VALUE' LEFT-JUSTIFIED.
    WRITE AT: 30 wa_vbrk-netwr LEFT-JUSTIFIED COLOR 6.
  ENDAT.
*Print Item details
  WRITE : /5 wa_vbrp-posnr LEFT-JUSTIFIED,
          15 wa_vbrp-arktx LEFT-JUSTIFIED,
          60 wa_vbrp-fkimg LEFT-JUSTIFIED,
          80 wa_vbrp-vrkme LEFT-JUSTIFIED
          105 wa_vbrp-netwr LEFT-JUSTIFIED,
          130 wa_vbrp-matnr LEFT-JUSTIFIED,
          150 wa_vbrp-mwsbp LEFT-JUSTIFIED.

```


*To print document total netvalue at the end of last item in the
 *document. The code in AT END OF will execute before the next new
 *document.

```
AT END OF vbeln
```

*The keyword SUM can be used within a control break to
 *automatically add up the column.

```
SUM.
```

*Prints total Sum for document

```
WRITE : / 105 WA_VBRP-NETWR LEFT-JUSTIFIED.
```

```
ENDAT.
```

*To identify the last record in the table

```
AT LAST.
```

```
WRITE : / 'END OF REPORT'.
```

```
ENDAT.
```

```
ENDLOOP.
```

Listing 5.22 Control Break Example

Listing 5.22 processes the header and item data of a billing document. The header data is fetched from the database table VBRK and the item data is fetched from the database table VBRP. We're providing a selection screen to the user via the SELECT-OPTIONS statement. The SELECT-OPTIONS statement allows the user to input multiple document numbers that should be processed by the program. You'll learn more about SELECT-OPTIONS in Chapter 6.

Based on the user's input, we fetch the relevant records from the database table and process the records to print in the output using the WRITE statement. We're using the control break statements within the LOOP statement of the item table IT_VBRP to format the output.

For example, we're using the AT FIRST statement to identify the first loop pass to print the column heading. AT NEW is used to identify a new document to read and print the header record. AT END OF is used to identify the last item of the current document to print the total sum for the document. Finally, we used the AT LAST statement to identify the last record of the internal table to print a report closing statement.

The WRITE statement supports various formatting techniques, such as specifying the position on the screen to print the record, using the WRITE AT addition, using / to print the record in a new line, and so on. We'll explore the WRITE statement further in Chapter 13.

The following rules are applicable while using control break statements:

- ▶ The internal table fields should be sorted by the fields used in the control break.
- ▶ The internal table can't be modified within the loop.
- ▶ The content of a work area specified in the LOOP statement after the INTO addition can't be modified.

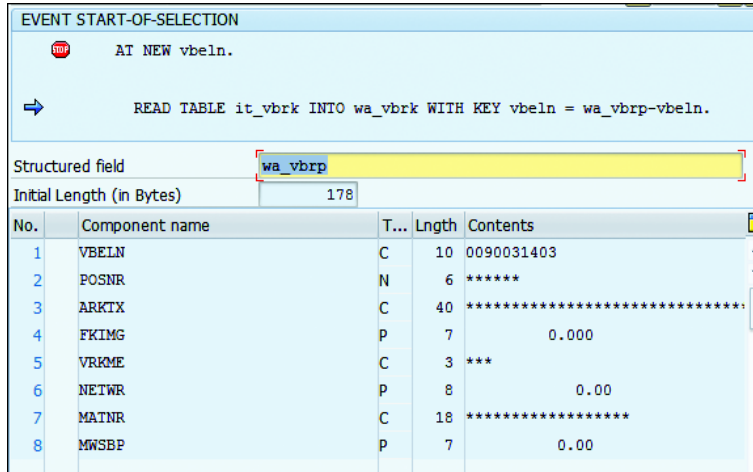
When the AT-ENDAT control structure is entered, the contents of the work area are modified as follows:

- ▶ The content of the components to which the control break is applied remain unchanged.
- ▶ All components with a character-like flat data type that are after the current control key are set to character * in every position.
- ▶ All the other components that are after the current control key are set to their initial values.

Figure 5.12 and Figure 5.13 display the contents of a work area before entering the control break and after entering the control break, respectively.

No.	Component name	T...	Lngh	Contents
1	VBELN	C	10	0090031403
2	POSNR	N	6	000001
3	ARKTX	C	40	Susi Süßkind
4	FKIMG	P	7	1.000
5	VRKME	C	3	ST
6	NETWR	P	8	600.00
7	MAINR	C	18	Erste Hilfe
8	MWSBP	P	7	96.00

Figure 5.12 Contents of Work Area before Control Break Is Applied, Viewed in Debugger



No.	Component name	T...	Length	Contents
1	VBELN	C	10	0090031403
2	POSNR	N	6	*****
3	ARKTX	C	40	*****
4	FKIMG	P	7	0.000
5	VRKME	C	3	***
6	NETWR	P	8	0.00
7	MATNR	C	18	*****
8	MWSBP	P	7	0.00

Figure 5.13 Contents of Work Area after Control Break Is Applied, Viewed in Debugger

When the AT-ENDAT control structure is exited, the content of the current table row is assigned to the entire work area.

5.3 Introduction to Open SQL Statements

SAP supports different relational database management systems (RDBMSs) like Oracle, Microsoft SQL, and SAP HANA. Structured Query Language (SQL) is a programming language designed for working with data in a relational database management system.

ABAP supports two types of SQL:

► Open SQL

Open SQL allows you to access database tables and perform certain actions independent of the underlying database that the SAP ERP system is using. When executing an Open SQL statement, the database interface of the work process will take care of converting the Open SQL statement to a Native SQL statement—that is, native to the underlying database. This ensures your ABAP code is portable and will work irrespective of the underlying database.

► Native SQL

Native SQL allows you to use database-specific SQL statements in an ABAP program. With a Native SQL statement, you can use database tables that are not

part of ABAP Data Dictionary. This allows you to integrate data that is not part of the SAP ERP system. To use a Native SQL statement in an ABAP program, precede the statement with the `EXEC SQL` keyword and close it with the `ENDEXEC` statement—for example:

```
EXEC SQL.  
<native sql statement>.  
ENDEXEC.
```

SQL statements are categorized into the following languages:

► **Data Definition Language (DDL)**

DDL statements are used to define the database structure or schema. Using a DDL statement, you can perform actions such as creating a table, dropping a table, changing the structure of the database table, and so on.

The following are some examples of DDL statements:

- **CREATE:** Creates objects in the database
- **ALTER:** Alters the structure of the database
- **DROP:** Deletes objects from the database
- **TRUNCATE:** Removes all records from a table, including all spaces allocated for the records
- **RENAME:** Renames an object

► **Data Manipulation Language (DML)**

DML statements manage data within schema objects. Using DML statements, you can't alter the database schema but you can manipulate the data. For example, you can fetch the data from a database table, modify the records of the database table, delete records from the database table, and so on.

The following are some examples of DML statements:

- **SELECT:** Retrieves data from a database
- **INSERT:** Inserts data into a table
- **UPDATE:** Updates existing data within a table
- **DELETE:** Deletes all records from a table, leaving the spaces for the records in place

With Open SQL, you can use DML statements. With Native SQL, you can use DDL statements. We generally use ABAP Data Dictionary to perform tasks that require using DDL statements with Native SQL.

In this section, we'll discuss various key concepts of an RDBMS, such as database design, tables and keys, foreign key relationships, and normalization. We'll also provide an introduction to Open SQL statements used to fetch data from database tables.

5.3.1 Database Overview

As discussed in [Chapter 2](#), the database is managed by an RDBMS. A relational database consists of multiple tables that store particular sets of data, and an RDBMS standardizes how the data is stored and processed. In a relational database, data is stored in multiple tables, with a relationship between the tables.

The relational database model was first conceived by E. F. Codd in 1969. The model is based on set theory and predicate logic (branches of mathematics). The idea behind the relational model is that the database consists of a series of tables that can be manipulated using a declarative approach (nonprocedural operations).

It's commonly thought that the word *relational* in a relational database pertains to the fact that you *relate* the tables in the database to one another. However, a table is also known as a *relation*, and the word *relational* takes its root from this. In fact, Codd and other database theorists use the terms *relations*, *attributes*, and *tuples* to refer to what developers call *tables*, *columns (fields)*, and *rows* (or what we'd call *files*, *fields*, and *records* in a physical sense).

In the following subsections, we'll explain how a database is modeled. We'll start by discussing the use of database tables and the significance of table keys. We'll then explore the concept of foreign key relationships to establish relationships between different tables of the database. We'll also discuss the concept of normalization, which helps to simplify database design and avoid redundancy. We'll conclude this section with an introduction to SAP HANA and the many benefits this database provides.

Relational Database Design

A database model to reflect a real-world system takes time and effort to conceive, build, and refine. You need to make decisions about the tables you need to create, what fields they'll contain, and how the tables can be related to each other.

A good database that's designed according to the relational model provides many advantages. A good design makes the data entries, updates, and deletions efficient, and it also helps make retrieval and summarization of data easy. Because most data is stored in the database and not in the application, the database should be sufficiently self-documenting. A good database design should also consider future changes in real-world requirements and should be flexible to allow for changes to the database schema.

Tables and Keys

A *table* is the basic entity in a relational model. Each table should represent an entity in the real world, and these entities can be real-world objects or events. For example, a customer is a real-world object, and the order placed by a customer is an event.

A table consists of rows and columns. The relational model dictates that each row in a table should be unique so that it can be uniquely addressed through programming. The uniqueness of a row is guaranteed by defining a primary key for the table. The primary key of a table consists of one or more fields that uniquely identify a row. Each table can have only one primary key. All fields from which a primary key is drawn are called *candidate keys*. Multiple secondary keys can be drawn from the other fields of the table.

Secondary keys can be unique or non-unique. Secondary keys are generally used to define secondary indexes for the table in order to speed up data retrieval. Keys can be simple or composite. If a key is made up of one field, it's called a *simple key*; if the key is made up of two or more fields, it's called a *composite key*.

When modeling a table, you should decide which candidate keys form the primary key of the table based on business requirements; there are no hard and fast rules to define the primary key of a table. In his book titled *SQL and Relational Basics*, Fabian Pascal notes that such a decision should be based upon the principles of minimalism (choose the fewest fields necessary), stability (choose a key that seldom changes), and simplicity/familiarity (choose a key that is both simple and familiar to users).

For example, if you have a table to store employee details—such as employee ID, first name, last name, address, telephone number, and zip code—which of these

fields will make for a good primary key? Following Pascal's guidelines, the address and telephone number can be ruled out because they change frequently. There's also a chance of the employee's name changing due to marriage or other reasons; plus, names can be misspelled, and there's a high likelihood that two employees will have the same first or last name.

Because the employee ID is unique across the organization, it makes the best candidate for the primary key. Even though there are no fixed rules for choosing a primary key, most developers prefer choosing a numeric primary key, because searches on numeric fields are more efficient than those on text fields.

Tip

In many situations, it's recommended to use static numbers, such as employee ID, customer ID, document number, and so on, as primary keys and to avoid descriptive texts.

Foreign Keys and Domains

Even though primary keys are defined in individual tables, there may be a need to define a relationship with different tables of the database using foreign key relations. A *foreign key* is a field in a table that is used to reference a primary key in another table.

Let's look at an example. [Table 5.2](#) shows a customer table in which customer details are stored. The primary key of this table is the *Cust ID* column, which uniquely identifies each row.

Cust ID	Name	Address	ZIP	Phone
1	Joey	171 CE	110099	99889900
2	James	345 DE	118899	88997788
3	John	563 WE	442299	88993354

Table 5.2 Customer Table

[Table 5.3](#) shows an orders table that stores information about customer orders. The primary key of this table is the *Order ID* column, which uniquely identifies each order.

Order ID	Cust ID	Item	Qty	UoM
2230	1	Pen	10	Pieces
2231	2	Pencil	10	Pieces

Table 5.3 Orders Table

In this example, the Cust ID field of [Table 5.3](#) is considered the foreign key of the table, because it can be used to refer to the customer table (see [Table 5.2](#)).

It's important that both the primary keys and foreign keys that are used to form a relationship share the same meaning and *domain*. A *domain* defines a possible set of values for a field.

For example, if a valid value for a customer ID can be a number between 1 and 10,000, then the Cust ID fields in both tables ([Table 5.2](#) and [Table 5.3](#)) should adhere to this range. By implementing a foreign key check, we can also ensure that the Cust ID field in the orders table ([Table 5.3](#)) consists of only the values available in the Cust ID field of the customers table ([Table 5.2](#)).

Relationship

Foreign keys model relationships between real-world entities. Such real-world entities can have complex relationships—for example, one entity may have multiple relations with other entities. In a relational database, relationships are defined between a pair of tables.

These tables can be related in one of three ways:

► One-to-one relationship

Two tables are said to be in a one-to-one relationship if for every row in the first table there is at most one row in the second table. One-to-one relationships are seldom used to model real-world entities; they are mostly used to split the data into multiple tables due to software limitations.

For example, you can split data into two tables if the number of fields exceeds the limitation of 249 per table, or you can model two tables with customer data if one of the tables can store more sensitive information about a customer. You have better access to control restrictions for this table compared to the table

that stores general information about the customer. The tables in a one-to-one relationship should have similar primary keys.

► **One-to-many relationship**

Two tables are said to be in a one-to-many relationship if for every row in the first table there can be zero, one, or multiple rows in the second table, but for every row in the second table there is exactly one row in the first table.

One-to-many relations are the most commonly modeled. The tables in a one-to-many relation are also called parent-child tables or header-item tables. Common tables with one-to-many relationships in the SAP system include order tables such as `VBAK-VBAP`, which stores header and item details of sales orders, and `EKKO-EKPO`, which stores the header and item details of purchase orders. The one-to-many relationship is also used to link a base table with a lookup table. For example, in the orders table you can store a two-character abbreviation for a unit of measure, and this info can be linked to a lookup table in which the descriptions for the abbreviations are maintained.

► **Many-to-many relationship**

Two tables are said to be in a many-to-many relationship when for every row in the first table there can be many rows in the second table, and for every row in the second table there can be many rows in the first table. Many-to-many relations cannot be directly modeled in a relational database, and they're typically broken down into multiple one-to-many relations.

Normalization

Normalization is the process of simplifying the design of a database. Normalization answers core questions regarding the number of tables required, what each table represents, the number of fields in each table, and the relationship between tables. Normalization also helps avoid redundancy and anomalies in database design.

Anomalies can occur while inserting, updating, or deleting records from a database table (if the database is poorly designed). For example, if you have a database table as shown [Table 5.4](#), when a record is updated to change the customer address, all the rows where the customer record exists should be updated. If any rows are missed, then it will lead to data inconsistency. This is called an *update anomaly*.

Item_ID	Customer_ID	Customer_name	Customer_address
1	921	Ryan	12E West End
2	921	Ryan	12E West End
3	728	John	44 Avenue Road

Table 5.4 Sample to Show Update Anomaly

If you try to insert a record for a customer that isn't available in the customer master table, it causes an *insertion anomaly*. If you delete item 3 in [Table 5.4](#), it will also delete the record of the customer John, because his details are linked to this item only. This is called a *delete anomaly*.

Normalization helps to overcome these anomalies and model a database that is consistent and predictable. Normalizations use the concept of normal forms to assist in designing an optimal structure. There are three available normal forms: first, second, and third.

The following subsections look at each of these normal forms.

First Normal Form

The first normal form (1NF) dictates that the field of a table should contain atomic values. The field should contain only one value. For example, the data in [Table 5.5](#) does not conform to 1NF, because the phone number field contains multiple values for the Emp_id 2290.

Emp_id	Emp_name	Phone_num
2289	John	99889988
2290	Mark	88998899 99887755

Table 5.5 Employee Not Conforming to 1NF

To make [Table 5.5](#) conform to 1NF, it should be adjusted as shown in [Table 5.6](#) by creating a new record for the second phone number in Emp_id 2290.

Emp_id	Emp_name	Phone_num
2289	John	99889988
2290	Mark	88998899
2290	Mark	99887755

Table 5.6 Employee Conforming to 1NF

Second Normal Form

For a table to conform to the second normal form (2NF), it should meet the following criteria:

- ▶ It should conform to 1NF.
- ▶ The non-key fields of the table should be fully dependent on all the primary key fields and not on the subset of the key fields.

For example, in [Table 5.7](#), let's assume that `Employee_ID` and `Department_ID` are the primary key fields. According to 2NF, the non-primary fields `Employee_name` and `Department_name` should be fully dependent on the complete primary key fields `Employee_ID` and `Department_ID`, not on the subset of the primary key fields.

That's not the case here; the `Employee_name` field can be identified by the `Employee_ID` field, and the `Department_name` field can be identified fully by the `Department_ID` field. Therefore, the non-primary fields of this table are only partially dependent on the primary key fields of the table, which is not allowed in 2NF.

Employee_ID	Department_ID	Employee_name	Department_name
1122	01	John	Sales
1123	02	Mark	Finance

Table 5.7 Not Conforming to 2NF

To make [Table 5.7](#) conform to 2NF, it should be split into two tables, as shown in [Table 5.8](#) and [Table 5.9](#). This ensures that there is no partial dependency on primary key fields.

Employee_ID	Department_ID	Employee_name
1122	01	John
1123	02	Mark

Table 5.8 Employee Table

Department_ID	Department_name
01	Sales
02	Finance

Table 5.9 Department Table

Third Normal Form

A table conforms to the third normal form (3NF) if it conforms to 2NF and all non-key fields of the table are mutually independent.

For example, [Table 5.10](#) does not conform to third normal form, because the field `Total` is dependent on the information in the nonkey fields `Quantity` and `Price`. To make this table conform to 3NF, it's best to remove the `Total` field from the table and handle the calculations in a query or report. This ensures that updating one of the fields of the table does not result in any anomalies.

Item_id	Quantity	Price	Total
1	10	10	100
2	20	15	300

Table 5.10 Conforming to 3NF

Every higher normal form is a superset of all lower normal forms. For example, if you design for 3NF, then by default it conforms to 1NF and 2NF. Normalization helps achieve optimal database design. A good database design takes time and effort and requires a good understanding of business requirements. A good design not only makes your applications efficient, but also prevents headaches down the line.

SAP HANA

There has been a growing need for real-time analytics over the years, which requires processing huge sets of data. Traditional database systems form data processing speed bottlenecks for large data volumes. SAP addressed this problem with its in-memory database system called *SAP HANA*.

SAP HANA has transformed the relational database industry with its innovations in hardware and software technology to process massive datasets in real time using in-memory computing. SAP HANA combines database, application processing, and integration services on a single platform.

SAP HANA stores compressed data in memory in a columnar format that enables faster scanning and quicker processing. Compared to traditional databases, programs take minutes to process the data from SAP HANA, as opposed to hours when using a traditional database.

The following are some important features to consider with SAP HANA when it comes to ABAP:

► **Parallelism**

One of the main features of the SAP HANA database is parallelism. SAP HANA takes advantage of the multiple cores of the processor to process information in parallel. Traditional database systems were built decades ago, when only single-core processors existed and the DRAM was extremely expensive.

The hardware and computing power of systems has changed tremendously over the years. SAP designed SAP HANA from the ground up to take advantage of modern hardware capabilities. Typically, an SAP HANA system performs around 3.5 billion scans per second per core and around twelve to fifteen million aggregations per second per core. This enables you to perform real-time analytics and process huge sets of data on the fly.

► **Code pushdown**

The SAP HANA database allows you to push certain data-intensive operations to the database using a technique known as *code pushdown*.

For example, if you want to calculate the sum of all invoices, you can use an aggregate function on the database instead of fetching all the invoice data into the application server and calculating the sum in an internal table loop. This can be achieved using ABAP Core Data Services (CDS) views.

► **ABAP Core Data Services (CDS) views**

ABAP CDS views are not persistent objects, but a projection of a database entity. An ABAP CDS view is created as a design-time file in the repository and exists as an ABAP Data Dictionary object once activated. This allows you to access the CDS view using Open SQL and to use aggregate functions.

5.3.2 Selecting Data from Database Tables

You can use internal tables to process data from a database table, allowing you to store multiple rows of data at once. In this section, we'll explore a few Open SQL statements that can be used to process data from a database.

The `SELECT` keyword fetches data from a database table. You can select a single record or multiple records, and you can select the data from the database into a structure or an internal table of your ABAP program. You can either select the complete row (all fields) of a database table or select only specific fields.

UNION Addition

As of SAP NetWeaver 7.5, the `UNION` addition can be used to create a union between the result sets of two `SELECT` statements.

When selecting data from a database, the row of the data object should be identical to the row of the database table. To ensure this, refer your data object to the table when selecting the complete row. If you need to select only certain fields from the row, make sure to define the fields in the same order in which you plan to select them. The `SELECT` statement supports an `INTO CORRESPONDING FIELDS` addition if the order of fields in your data object doesn't match the order of the selection.

[Listing 5.23](#) shows the usage of a `SELECT SINGLE` statement to select a single row from the database table. If more than one row satisfies the `WHERE` clause, the first matching row is fetched.

```
TYPES: BEGIN OF ty_vbrk,
        VBELN TYPE VBELN_VF, "Document Number
        FKDAT TYPE FKDAT, "Bill Date
        NETWR TYPE NETWR, "Net Value
        KUNRG TYPE KUNRG, "Payer
      END OF ty_vbrk.
```

```
DATA : it_vbrk TYPE STANDARD TABLE OF ty_vbrk,
      wa_vbrk TYPE ty_vbrk.
```

```
*Selecting a single row of specific fields from table vbrk.
SELECT SINGLE vbeln fkdat netwr kunrg FROM vbrk
              INTO wa_vbrk
              WHERE vbeln EQ '9000'.
```

Listing 5.23 Using SELECT SINGLE

Listing 5.24 uses SELECT...ENDSELECT to fetch all matching rows, one row at a time. The system loops between SELECT...ENDSELECT, transferring one row at a time to the structure wa_vbrk. You can process the contents of wa_vbrk in each loop by writing the statements before ENDSELECT. This is an efficient way to process the records from the database table if you plan to use a database table only once during program execution.

However, if you plan to reuse the data multiple times during the life of the program execution, we recommend using internal tables, which allows you to fetch the data once into your program and access it subsequently multiple times, as opposed to accessing the database every time you need the same data.

```
*Selecting all matching rows of specific fields from the table
*vbrk using a structure.
*Here the system fetches one row at a time and loops between
*select...endselect. The row can be processed between SELECT...ENDSELECT.
SELECT vbeln fkdat netwr kunrg FROM vbrk
              INTO wa_vbrk
              WHERE vbeln EQ '9000'.

ENDSELECT.
```

Listing 5.24 Using SELECT...ENDSELECT

Listing 5.25 shows using an internal table in a SELECT statement, also known as performing an *array fetch*. Once the SELECT statement is executed, you can access the data from the internal table as discussed previously.

```
*Selecting all matching rows of specific fields from the table
*vbrk using an internal table.
SELECT vbeln fkdat netwr kunrg FROM vbrk
              INTO TABLE it_vbrk
              WHERE vbeln EQ '9000'.
```

Listing 5.25 Selecting Data to Internal Table

[Listing 5.26](#) shows selecting the fields of the complete table row. Notice the difference in data object declaration and the usage of `*` as the `SELECT` statement.

```
DATA : it_vbrk TYPE STANDARD TABLE OF vbrk,
      wa_vbrk TYPE vbrk.

*Selecting a single row of all fields from table vbrk.
SELECT SINGLE * FROM vbrk
           INTO wa_vbrk
           WHERE vbeln EQ '9000'.
*Selecting all matching rows of all fields from table vbrk
*using a structure.
SELECT * FROM vbrk
           INTO wa_vbrk
           WHERE vbeln EQ '9000'.
ENDSELECT.
*Selecting all matching rows of all fields from table vbrk
*using an internal table.
SELECT * FROM vbrk
           INTO TABLE it_vbrk
           WHERE vbeln EQ '9000'.
```

Listing 5.26 Selecting All Fields of a Row

5.3.3 Selecting Data from Multiple Tables

The syntax we've discussed so far allows you to select data from a single table. However, in real-world applications, data is spread across multiple tables with foreign keys.

To select data from tables that are in a foreign key relationship, you can use *joins* or use the `FOR ALL ENTRIES` addition with the `SELECT` statement. The `FOR ALL ENTRIES` addition allows you to select the data from the database table and compare it to the data in an internal table that was previously selected from another database table. This is efficient if you're processing data from two tables.

[Listing 5.27](#) shows the syntax to use for all entries. Here, the data is first selected from database table `VBRK` into an internal table `IT_VBRK`; then the data in `IT_VBRK` is compared to select the data from database table `VBRP`.

When using `FOR ALL ENTRIES`, make sure the internal table used for comparison isn't empty. Otherwise, the system will execute the `SELECT` statement without restriction. Notice the `IF` condition used in the listing.

```
SELECT * FROM vbrk
           INTO TABLE it_vbrk
```



```

                WHERE vbeln EQ p_vbeln.
IF sy-subrc IS INITIAL.
SELECT * FROM vbrp
                INTO TABLE it_vbrp
                FOR ALL ENTRIES IN it_vbrk
                WHERE vbeln EQ it_vbrk-vbeln.
ENDIF.

```

Listing 5.27 Using FOR ALL ENTRIES

Alternately, *joins* allow you to select data from up to twenty-five database tables. There are two types of joins:

► **Inner joins**

Inner joins require an entry with the **KEY** of one table in the second table, to be extracted to an internal table.

► **Outer joins**

Irrespective of whether an entry exists or not in the second table, outer joins still extract data from the first table.

Listing 5.28 highlights the difference between an inner and outer join. If the internal table has three fields—FLD1, FLD2, and FLD3—the result for an inner join and outer join are shown in Listing 5.28.

Table A		Table B	
Field 1	Field 2	Field 1	Field 3
X	100	X	ABC
Y	200	X	DEF
Z	300	Z	WXY

Inner Join:

```

-----
FLD1   FLD2   FLD3
-----
X      100    ABC
X      100    DEF
Z      300    WXY

```

Outer Join:

```

-----
FLD1   FLD2   FLD3

```

```

-----
X      100      ABC
X      100      DEF
Y      200           "No entry for Key Y in table B so FLD3 is blank
Z      300      GHI

```

Listing 5.28 Inner Join vs. Outer Join

Listing 5.29 shows an example using an inner join. Here, we join the columns `carrname`, `connid`, and `fldate` of the database tables `scarr`, `spfli`, and `sflight` using two inner joins. This creates a list of flights from `p_cityfr` to `p_cityto`. An alias name is assigned to each table.

```

PARAMETERS: p_cityfr TYPE spfli-cityfrom,
            p_cityto TYPE spfli-cityto.

TYPES: BEGIN OF wa,
        fldate TYPE sflight-fldate,
        carrname TYPE scarr-carrname,
        connid  TYPE spfli-connid,
      END OF wa.

DATA itab TYPE SORTED TABLE OF wa
          WITH UNIQUE KEY fldate carrname connid.

SELECT c~carrname p~connid f~fldate
      FROM ( ( scarr AS c
              INNER JOIN spfli AS p ON p~carrid  = c~carrid
                                   AND p~cityfrom = p_cityfr
                                   AND p~cityto   = p_cityto )
            INNER JOIN sflight AS f ON f~carrid = p~carrid
                                   AND f~connid = p~connid )
      INTO CORRESPONDING FIELDS OF TABLE itab.

```

Listing 5.29 Inner Join Example

Listing 5.30 shows an example using an outer join.

```

PARAMETERS p_cityfr TYPE spfli-cityfrom.

TYPES: BEGIN OF wa,
        carrid  TYPE scarr-carrid,
        carrname TYPE scarr-carrname,
        connid  TYPE spfli-connid,
      END OF wa.

DATA itab TYPE SORTED TABLE OF wa
          WITH NON-UNIQUE KEY carrid.

START-OF-SELECTION.
SELECT s~carrid s~carrname p~connid

```

```

FROM scarr AS s
LEFT OUTER JOIN spfli AS p ON s~carrid = p~carrid
                        AND p~cityfrom = p~cityfr
INTO CORRESPONDING FIELDS OF TABLE itab.

```

Listing 5.30 Outer Join Example

Using joins may seem a bit overwhelming initially, so we suggest you practice using `FOR ALL ENTRIES` and use joins only if you need data from many tables.

5.4 Processing Data from Database via Internal Tables and Structures

Internal tables and structures are primarily used to process data from database tables. Because all transaction data is stored in database tables, we use internal tables and structures in almost all ABAP programs. Working with internal tables may seem a bit tricky initially, especially when working with multiple tables.

When processing data from multiple internal tables, you should always avoid nested loops (a loop within a loop) because they degrade performance. We always loop the main table and read the secondary tables.

When you have a requirement to process data from more than one table, you may struggle to identify which internal table to loop and which one to read. For example, say you have a requirement to process material and plant data, and this data is available in database table `MARC`. To print the plant details of a material in the output, write the code shown in [Listing 5.31](#).

```

TYPES: BEGIN OF ty_marc,
        matnr TYPE matnr,
        werks TYPE werks_d,
      END OF ty_marc.
DATA : it_marc TYPE STANDARD TABLE OF ty_marc,
      wa_marc TYPE ty_marc.
PARAMETERS p_matnr TYPE matnr.
SELECT matnr werks from marc INTO TABLE it_marc WHERE matnr EQ p_matnr.
LOOP AT it_marc INTO wa_marc.
WRITE: / wa_marc-matnr, wa_marc-werks.
ENDLOOP.

```

Listing 5.31 Sample Code to Print Material Plant Data

Now, if the code in [Listing 5.31](#) has to be expanded to print the plant description as the third column for every plant number, you need to select the plant description from another table, T001W, and compare against the plant data in table MARC, as shown in [Listing 5.32](#).

```
TYPES: BEGIN OF ty_marc,
        matnr TYPE matnr,
        werks TYPE werks_d,
      END OF ty_marc.
TYPES : BEGIN OF ty_t001w,
        Werks TYPE werks_d,
        Name1 TYPE name1,
      END OF ty_t001w.
DATA : it_marc TYPE STANDARD TABLE OF ty_marc,
      wa_marc TYPE ty_marc,
      it_t001w TYPE STANDARD TABLE OF ty_t001w,
      wa_t001w TYPE ty_t001w.
PARAMETERS p_matnr TYPE matnr.
SELECT matnr werks from marc INTO TABLE it_marc WHERE matnr EQ p_matnr.
IF sy-subrc IS INITIAL.
SELECT werks name1 FROM t001w INTO TABLE it_
t001w FOR ALL ENTRIES IN it_marc WHERE werks EQ it_marc-werks.
ENDIF.
LOOP AT it_marc INTO wa_marc.
READ TABLE it_t001w INTO wa_t001w WITH KEY werks = wa_marc-werks.
WRITE: / wa_marc-matnr, wa_marc-werks, wa_t001w-name1.
ENDLOOP.
```

Listing 5.32 Printing Plant Descriptions

As you can see in [Listing 5.32](#), we selected the required information from two database tables into two internal tables by matching the entries of the first table with the second table. After selecting the data into the respective internal tables, we loop one table and read the other table in the loop to print the output.

Now, how do you decide which table to loop, and which table to read? This is where many beginners become confused, but the answer is simple: Loop the table that has the main data, and read the table that has the secondary data. In this example, the main data we're printing is the plant details of the material; the plant descriptions are just secondary information. Therefore, we're looping the main table MARA and reading the secondary table T001W.

5.5 Introduction to the Debugger

We'll discuss debugging details in [Chapter 17](#), but for now we'll cover the basics of debugging so that you can check your program execution to understand the program flow.

The debugger is a useful tool not only to troubleshoot logical errors in your code, but also to understand the program flow, and it can be an invaluable tool to understand your code better.

Two types of debugger are available: the Classic Debugger and the New Debugger. The New Debugger provides a lot more options than the Classic Debugger. If you're a beginner, we suggest that you use the Classic Debugger if you're using a pre-SAP NetWeaver 7.5 system; as you progress through this book, you should feel at home exploring the New Debugger.

With the release of SAP NetWeaver 7.5, the Classic Debugger has been declared obsolete, and it's no longer possible to set the Classic Debugger as your default debugger. You can still use the Classic Debugger by clicking the information icon in the debugger settings, however. Alternately, you can refer to [Chapter 17](#) for a quick introduction to the New Debugger.

To always use the Classic Debugger, you need to make it the default debugger in your personal settings, as shown in [Figure 5.14](#).

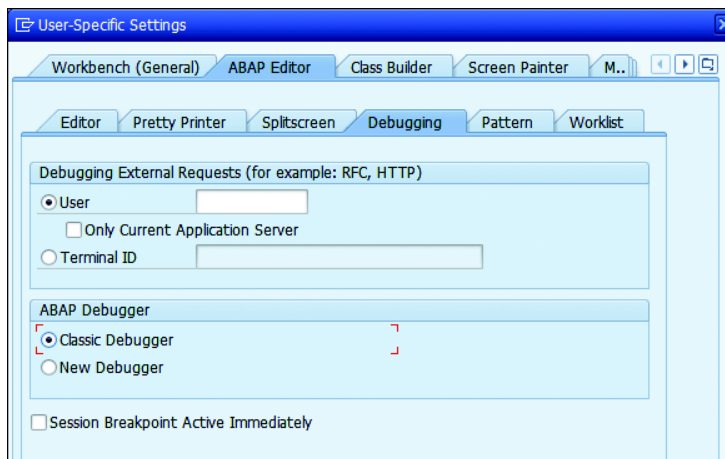


Figure 5.14 Setting Classic Debugger As Default

Open the USER-SPECIFIC SETTINGS dialog box by following the menu path UTILITIES • SETTINGS from any ABAP Workbench tool, and then select the Classic Debugger from the DEBUGGING tab under the ABAP EDITOR tab.

From now on, the system will always run the Classic Debugger.

To turn on debugging, you need to set a breakpoint in the program. A *breakpoint* interrupts program execution and loads the debugger when the program execution reaches the statement for which the breakpoint is set. You can set a breakpoint a couple of ways, as shown in [Figure 5.15](#).

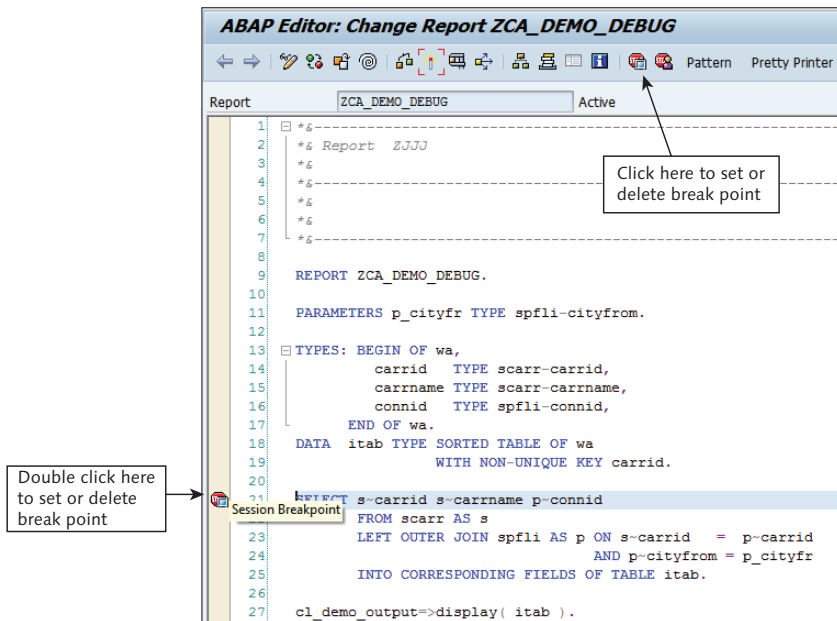


Figure 5.15 Setting Breakpoints

Once you set a breakpoint, execute the program normally; the system will automatically hold the program execution and present the debugger screen when it reaches the breakpoint, as shown in [Figure 5.16](#).

On the debugger screen, you can press **F5** to execute one statement at a time or **F8** to stop at the next breakpoint or to complete the program execution if there are no further breakpoints. If the debugger screen opens and you wish to exit the debugger, you can select the menu option DEBUGGING • DEBUGGING OFF in the debugger screen (see [Figure 5.17](#)).

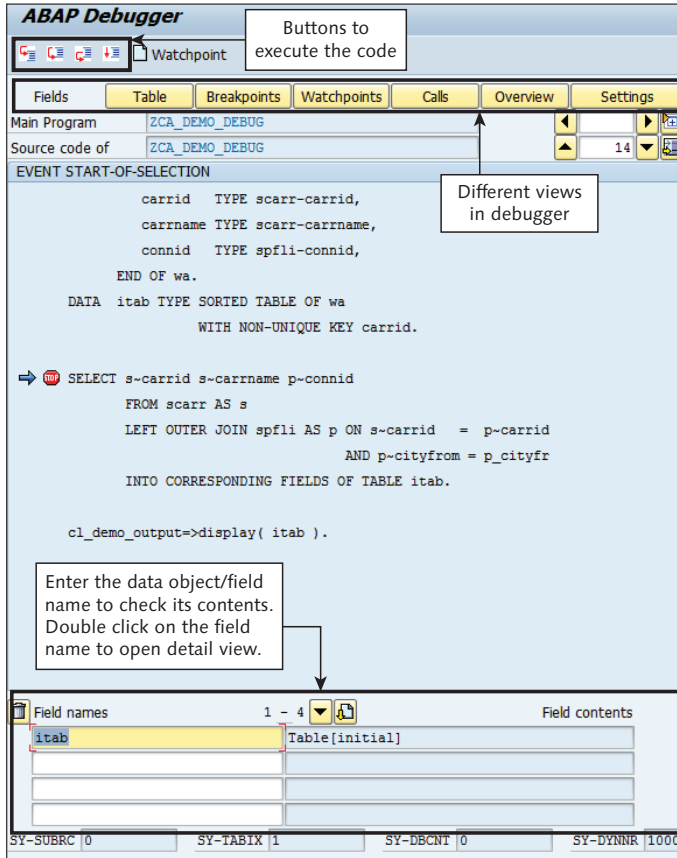


Figure 5.16 Debugger Screen

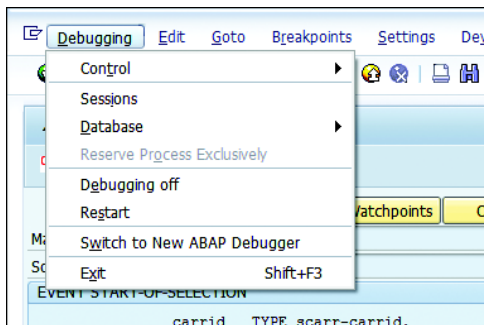


Figure 5.17 Menu Option to Exit Debugger

While you're in debug mode, you can type the name of a field to check its content. You can double-click on the field name to see a more detailed view of the field, as shown in [Figure 5.18](#). This helps confirm that your variables are filled with the correct data and to see what happens after each statement is executed. This way, you don't have to imagine the program execution; you can see it step by step.

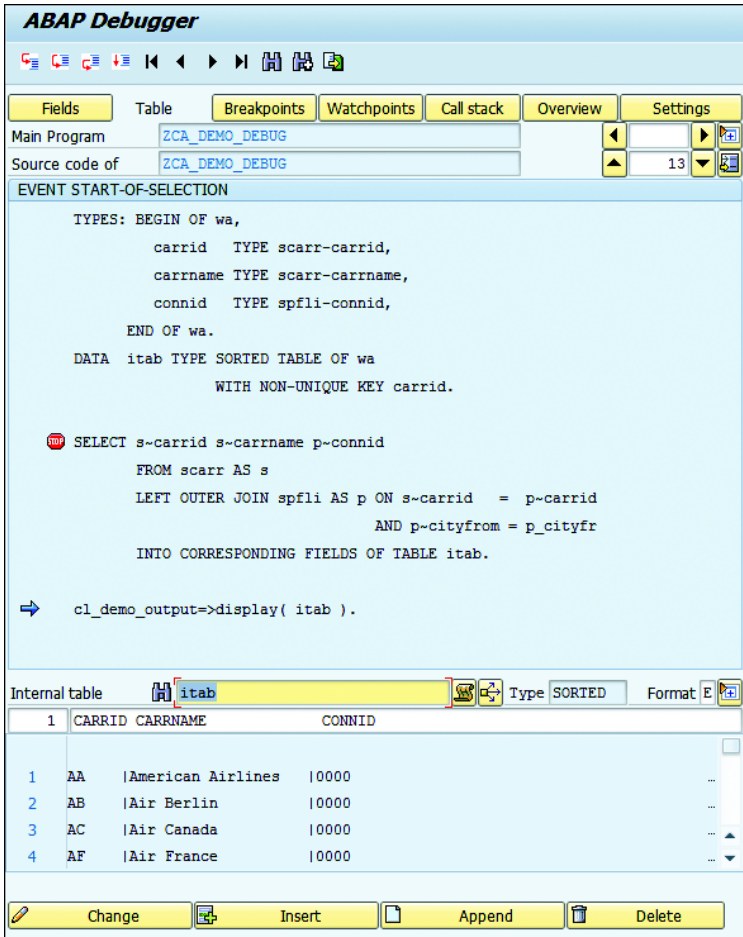


Figure 5.18 Detail View of Field

We'll explore all the options in the debugger later in the book in [Chapter 17](#), but for now you can use these basic options to better understand your program flow

and troubleshoot any issues. For example, if you've written some code but don't see the output, you can debug to see where the problem is, allowing you to check if the problem is with your selection logic or output logic.

5.6 Practice

We've explored quite a lot of concepts in this chapter, centering on processing data from database tables. The following task should help you put the theory you've learned into practice.

Ahead, you'll find a requirement to develop a simple report to display the header and item details of a billing document. Working on this report will require you to use the knowledge you've gained so far to define a simple selection screen for the report that will take the user input, fetch the relevant data from the database, and print it in the output:

Create a report to print the header and item details of a billing document. You can use Transaction VF03 as a reference. The report should take billing document number as the input and print the header and item details of the document as output.

The header details of the billing document are stored in table `VBRK`, and line items are stored in table `VBRP`. Identify the fields in table `VBRK` that contain the header details, such as document number, billing date, payer, and net value, and print them as header information in the output.

Identify the fields in table `VBRP` that contain the item details, such as item number, description, billed quantity, unit of measure, net value, and tax amount. Print these details below the header details in the output with respective column headings.

5.7 Summary

In this chapter, we looked at structures and internal tables. We use structures and internal tables while working with any data that is in a structure format with rows and columns. We also explored different types of internal tables and discussed the key differences between them. We examined basic Open SQL statements that

can be used to select data from database tables into your ABAP program for processing, and we also explored the basics of debugging to help you debug any minor issues and understand the program flow.

In the next chapter, we'll discuss how to enable user interaction with our code.

So far, we've covered basic concepts for writing ABAP programs. In this chapter, we'll introduce the concepts that enable user interaction with programs.

6 User Interaction

In an SAP GUI environment, there are typically three types of screens: the selection screen, the list screen, and the general screen. In ABAP, a screen is also known as a *dynamic program (dynpro)*. A dynpro is a component of a program and can be defined for three types of ABAP programs: *executable programs*, *module pool programs*, and *function groups*. A program can have multiple dynpros.

A dynpro consists of a screen, dynpro fields, and dynpro flow logic. General dynpros are created with the Screen Painter tool, and special dynpros (selection screens, list screens) are created implicitly using specific keywords. For example, `SELECT-OPTIONS` and `PARAMETERS` define selection screens, and `WRITE` defines a list screen.

Dynpro flow logic contains processing blocks for events that are triggered before a screen is displayed to the user and after user action on the displayed screen. Using these processing blocks, we can control the behavior of a dynpro and react to user actions. You'll learn about various processing blocks in [Chapter 7](#). Dynpros, together with the new web-oriented features (such as ABAP Web Dynpro) of SAP NetWeaver AS ABAP, form the basis of user dialogs in an ABAP system.

In this chapter, we'll discuss the rudimentary concepts of selection screens to get you started with basic report development, and you'll use all the concepts discussed so far in this book to develop a report program (see [Section 6.1](#)). This will allow you to understand the practical application of these concepts. We'll defer an in-depth discussion of selection screens to [Chapter 14](#), and we'll discuss the other two screens in [Chapter 12](#) (general screens) and [Chapter 13](#) (list screens).

In [Section 6.2](#), we'll discuss messages so that you cannot only take user inputs for your program through the selection screen but also validate the input and show a message to the user accordingly.

6.1 Selection Screen Overview

Selection screens are special dynpros that can be defined in certain program types—for example, executable programs, function groups, and module pools. Selection screens are defined in the global declaration area of an ABAP program with the statements `SELECT-OPTIONS`, `SELECTION-SCREEN`, and `PARAMETERS`, without using the Screen Painter tool. The screens of the selection screens can contain a subset of the screen elements of general dynpros.

Selection screens are typically used in report programs, in which we first present the user with a screen to provide the selection criteria and then based on user input pull the required data from the database to be presented as the report output.

For example, if we want to develop a report to show a flight's status, we first need to take the input (selection criteria) from the user to determine the flight number. The user typically provides this input on a selection screen. After processing, we show the result on a different screen (either list dynpro or general dynpro).

Selection screens essentially have two tasks:

- ▶ They enable users to provide selection criteria for a report program.
- ▶ They provide an interface for data transfer between two executable programs when a program is called internally using the `SUBMIT` statement.

Each screen is assigned a four-digit number. A program can have a *standard selection screen* and a *user-defined selection screen*. Screen number 1000 is assigned to the standard selection screen by default. The standard selection screen is created automatically when you use the `PARAMETERS` or `SELECT-OPTIONS` keyword in the global declaration area of the program. Other user-defined selection screens can be created for the program using the `SELECTION-SCREEN BEGIN OF SCREEN` keyword with each user-defined screen assigned a unique screen number.

The `SELECTION-SCREEN` statement can also be used to define different blocks with frames to better organize the selection screen fields (see [Figure 6.1](#)). All screens

(such as general screens, selections screens, and list screens) share the same namespace in the program, so you can't assign the same screen number to different types of screens in the same program.

Consignment and Pipeline Settlement

Selection

Company Code

Vendor

Plant

Material

Document Date

Posting Date

Material Document

Consignment

Pipeline

Processing

Display

Settle

Display Options

Withdrawals not settled

Settled withdrawals

Document Number

Layout

Figure 6.1 Selection Screen Fields Organized with Blocked Frames

When you use the `PARAMETERS` or `SELECT-OPTIONS` keyword in the program and activate it, the system automatically creates the screen with all the screen elements and dynpro flow logic. If the `SELECTION-SCREEN` keyword is not used to define a user-defined screen, then the program automatically creates a standard selection screen with dynpro number 1000 and with all the `PARAMETERS` and `SELECT-OPTIONS` fields belonging to this standard screen.

We can define various screen elements on the selection screen, such as input fields, checkboxes, radio buttons, list boxes, and push buttons. All the screen elements are defined in the program using the `PARAMETERS` keyword. The `SELECT-OPTIONS` keyword defines a range field on the selection screen, which supports more complex selection criteria.

In the following subsections, we'll discuss the `PARAMETERS`, `SELECT-OPTIONS`, and `SELECTION-SCREEN` keywords in more detail.

6.1.1 PARAMETERS

Parameters are components of a selection screen that are assigned a global elementary data object in the ABAP program and an input field on the selection screen. In other words, the `PARAMETERS` keyword performs two tasks:

- ▶ Defines a data object in the program
- ▶ Defines a screen element (such as an input field or checkbox) on the selection screen

The system automatically links a program data object with the screen field to facilitate automatic data transfer between the screen and the program. Thereby the user inputs on the screen are automatically transferred to the ABAP program and stored in the corresponding data object, which can then be accessed by ABAP statements.

The syntax for using the `PARAMETERS` keyword is somewhat similar to that of the `DATA` keyword, with its own set of additions. The `PARAMETERS` field is typed by referring to a data type from which it derives its technical attributes. [Listing 6.1](#) shows the syntax to use the `PARAMETERS` statement.

```
PARAMETERS {para TYPE data_type}
           [type_options]
           [screen_options]
           [value_options]
```

Listing 6.1 Parameters Keyword Syntax

[Listing 6.2](#) shows an example of the `PARAMETERS` keyword in action.

```
PARAMETERS p_name TYPE c LENGTH 20.
PARAMETERS p_name(20) TYPE c.
PARAMETERS p_carrid TYPE s_carr_id.
PARAMETERS p_chk AS CHECKBOX DEFAULT 'X'.
```

Listing 6.2 Parameters Keyword Example

The name of the parameter field `para` (in the syntax shown in [Listing 6.1](#)) can contain a maximum of eight characters. This statement is allowed in the global declaration part of the program.

Similar to the `DATA` statement, the length can only be specified in the `PARAMETERS` statement if the data type specified in `TYPE_OPTIONS` has a generic length (types `c`, `n`, `p`, and `x`).

The `PARAMETERS` statement has the following effects:

- ▶ Declares a global variable of the specified length and type in the program.
- ▶ Creates an input field with the same name and corresponding type in the new row of the current selection screen. The length of the input field on the screen is determined from the parameter definition. The maximum visible length of the input field is 45, so if the parameter length is more than 45, the field will be displayed with movable content. The visible length of the field can be controlled using the `VISIBLE LENGTH` addition with the `PARAMETERS` keyword.
- ▶ Automatically creates a label for each field to the left side of the input field. The maximum length of the field label is 30, and it displays either the name of the parameter or the selection text to which the parameter is assigned in the text elements (see [Section 6.1.4](#)) of the program.
- ▶ Allows for the attributes of the screen elements to be modified by passing `SCREEN_OPTIONS` during the declaration of the parameters.

In the following subsections, we'll discuss the type options, screen options, and value options that can be supplied with the `PARAMETERS` statement.

TYPE_OPTIONS

The `PARAMETERS` keyword can be supplied with different type options. These options define the data type of the parameter. If no type options are specified, the default option is `TYPE c` with `LENGTH 1`. The data type can be defined by referring to an existing data type of the program, to a data object, or to a data type from ABAP Data Dictionary.

If the `PARAMETERS` field is referred to a data type from ABAP Data Dictionary, it adopts all the screen-relevant properties (such as field labels or input helps) defined for the data type in ABAP Data Dictionary. If a domain is attached to the data element, conversion routines defined in the domain will be executed to convert the data to internal and external formats during the data transport from and to the input field, respectively.

There are three ways to provide `TYPE_OPTIONS`:

- ▶ `TYPE data_type [DECIMALS dec]`
- ▶ `LIKE dobj`
- ▶ `LIKE (name)`

The following subsections look at each of these options.

TYPE data_type [DECIMALS dec]

The `TYPE data_type [DECIMALS dec]` addition types the parameter with `data_type`, where `data_type` may be any of the following:

- ▶ A predefined ABAP type, with the exception of `f` and `xstring`;
- ▶ A non-generic data type from ABAP Data Dictionary;
- ▶ A non-generic public data type of a global class; or
- ▶ A user-defined elementary data type from the same program (which is not of the type `f` or `xstring`), already defined using the `TYPES` keyword

If the predefined `TYPE p` (packed decimal) is specified, the `DECIMALS` addition can be used to define the number of decimal places.

If the parameter is referred to a data element to which a domain of `TYPE CHAR, LENGTH 1`, and the fixed values "X" and a space is attached, the input field on the selection screen is automatically displayed as a checkbox.

The following is an example of an elementary data type:

```
PARAMETERS p_carrid TYPE c LENGTH 3.
```

The next example automatically inherits the `F1` and `F4` help from ABAP Data Dictionary and refers to the data element:

```
PARAMETERS p_carrid TYPE s_carr_id.
```

The following example is a declaration of a parameter with reference to the component `carrid` of the database table `spfli`:

```
PARAMETERS p_carrid TYPE spfli-carrid.
```

LIKE dobj

Using the `LIKE dobj` addition, the parameter adopts all the properties of a data object `dobj` already declared in the program.

Here's an example of this addition in use:

```
DATA v_carrid TYPE s_carr_id.
PARAMETERS p_carrid LIKE v_carrid.
```


TYPE vs. LIKE

TYPE is used to refer to an elementary data type, local data type in a program, or data type in ABAP Data Dictionary. LIKE is used to refer to another data object in the program.

To support backward compatibility, it's still possible to use LIKE to refer to an ABAP Data Dictionary table or flat structure; however, this usage is obsolete and should be avoided.

LIKE (name)

The LIKE (name) addition allows you to supply the field reference dynamically. name expects a flat character data object containing the name of a component in a flat structure from ABAP Data Dictionary in uppercase. The parameter is created internally with data TYPE c and LENGTH 132, but it's displayed on the selection screen with a length, field label, input help, and field help as per the data type specified in name.

[Listing 6.3](#) shows an example of dynamically assigning the data type for the input field. In this example, we're passing the field name SCOL_FLIGHT_MEAL-CARRID at runtime. The statement comp = 'SCOL_FLIGHT_MEAL-CARRID' in the example is executed before the selection screen is called because it's maintained under the AT SELECTION-SCREEN OUTPUT event. The ABAP runtime environment triggers various selection screen events that we can maintain in the program to process selection screens.

You'll learn about various selection screen events in detail in [Chapter 14](#).

```
DATA comp TYPE c LENGTH 30.  
PARAMETERS p_dyn LIKE (comp).  
AT SELECTION-SCREEN OUTPUT.  
    comp = 'SCOL_FLIGHT_MEAL-CARRID'.
```

Listing 6.3 Dynamically Assigning Data Type for Input Field

In [Listing 6.3](#), the keyword AT SELECTION-SCREEN OUTPUT defines an event that ensures the code under it is executed before the selection screen is called. Because this event is triggered before showing the selection screen to the user, we can make use of it to dynamically assign the properties to the selection screen field.

We'll explore all the program events in [Chapter 7](#).

SCREEN_OPTIONS

These options allow you to manipulate screen fields, including declaring the input field as a required field or hiding the input field on the selection screen. Using `SCREEN_OPTIONS`, the input field can be displayed as a checkbox, radio button, or dropdown list box.

The following subsections look at the different additions for `SCREEN_OPTIONS`.

OBLIGATORY

The `OBLIGATORY` addition defines the input field on the selection screen as a required (obligatory) field. The program can't be executed if the obligatory field is empty. The runtime system automatically performs a validation to check if all obligatory fields are filled with input values when the user selects the `EXECUTE` function (F8) to exit the selection screen. If the obligatory field isn't filled, the user can only use the functions `BACK`, `EXIT`, or `CANCEL` to leave the selection screen.

An example of this addition is as follows:

```
PARAMETERS p_carrid TYPE s_carr_id OBLIGATORY.
```

NO-DISPLAY

When the `NO-DISPLAY` addition is used, no screen elements are generated for the parameter on the selection screen. This is useful when you want to use the parameter only as part of the interface defined by the selection screen in an executable program that can be supplied with a value by the calling program with the `SUBMIT` statement. The `SUBMIT` statement is used to execute an executable program internally from another program.

An example of this addition is as follows:

```
PARAMETERS p_carrid TYPE s_carr_id NO-DISPLAY.
```

VISIBLE LENGTH vlen

The `VISIBLE LENGTH vlen` addition controls the visible length of a screen field. Here, `vlen` is entered directly as a positive number. If `vlen` is less than the length of the parameter and less than the maximum visible length (45), then the input field is displayed as per the length defined in `vlen`, with movable content. Otherwise, the addition is ignored.

An example of this addition is as follows:

```
PARAMETERS: p_carriid TYPE s_carr_id VISIBLE LENGTH 2.
```

AS CHECKBOX

The **AS CHECKBOX** addition generates the input field as a checkbox with the corresponding description next to it on the right. The value of the field is filled with 'X' or a space when the checkbox is selected or deselected, respectively. The checkbox can be selected by default by assigning a default value 'X' during the declaration. This parameter must be created with **TYPE c** and **LENGTH 1**. Checkboxes are used to allow multiple selection.

An example of this addition is as follows:

```
PARAMETERS p_chk AS CHECKBOX.
PARAMETERS p_chk AS CHECKBOX DEFAULT 'X'.
```

RADIOBUTTON GROUP

The **RADIOBUTTON GROUP** addition generates the input field as a radio button with the corresponding description next to it on the right. The radio button is selected if the value of the field is X; otherwise, it isn't selected.

Radio buttons are always grouped together because they enable the user to select from several options. Only one of the radio buttons can be selected at once in a group. *Groups* define the radio button groups for the parameter. Two or more radio buttons are assigned to the same group by entering the name of the group directly as a character string with a maximum of four characters. Within a selection screen, there must be a minimum of two parameters in the same radio button group. There cannot be more than one radio button group with the same name in one program.

In a radio button group, only one parameter can be selected by default, which means that the **DEFAULT** addition can be used with only one parameter of the group. By default, the first parameter in a radio button group is set to the value X and the rest are set to space. This parameter must be created with **TYPE c** and **LENGTH 1**.

An example of the radio button group is shown in [Listing 6.4](#).

```
PARAMETERS: p_day RADIOBUTTON GROUP rb1,
             p_month RADIOBUTTON GROUP rb1 DEFAULT 'X'.
```

```
PARAMETERS: p_hour RADIOBUTTON GROUP rb2,
             p_sec RADIOBUTTON GROUP rb2.
```

Listing 6.4 Radio Button Group

AS LISTBOX VISIBLE LENGTH vlen

The AS LISTBOX VISIBLE LENGTH vlen addition generates a dropdown list box for an input field on the selection screen. The list box is automatically filled with values if the parameter is created with a data type from ABAP Data Dictionary and the data type is associated with the input help in ABAP Data Dictionary. Otherwise, a single-row list box is displayed containing the current value of the parameter.

You must use the VISIBLE LENGTH addition to specify the visible length of the input field. An example of this addition is shown in [Listing 6.5](#).

```
PARAMETERS p_carrid TYPE spfli-carrid
             AS LISTBOX VISIBLE LENGTH 20
             DEFAULT 'LH'.
```

Listing 6.5 AS LISTBOX VISIBLE LENGTH vlen

[Figure 6.2](#) shows different screen elements on the selection screen defined using the PARAMETERS keyword.

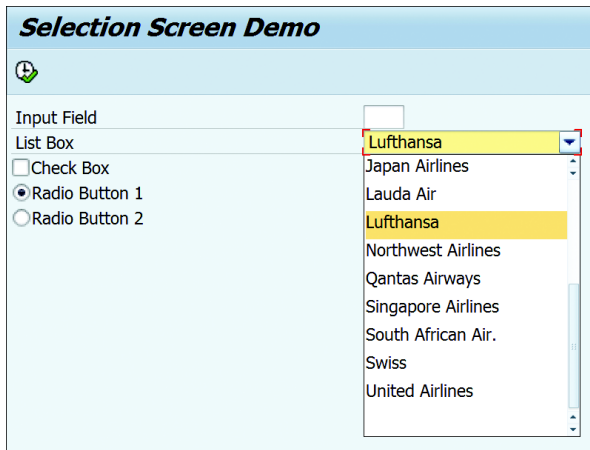


Figure 6.2 Different Screen Elements on Selection Screen

VALUE_OPTIONS

Using `VALUE-OPTIONS`, it's possible to define a default value or allow lowercase letters to be transported to the data object. It's also possible to set a search help, an SPA/GPA parameter to be bound, or a check to be executed against a value list.

For simplicity, we'll defer the discussion of these options to [Chapter 14](#).

6.1.2 SELECT-OPTIONS

The `PARAMETERS` statement defines a single input field in which the user can input a single value. For example, say that a user wants to run a report to see information about a billing document. With `PARAMETERS`, the user can run the report to check the details of one document at a time, because the parameters field can only take one document number as input.

But what if the user wants to check the details of multiple documents at once? That's when `SELECT-OPTIONS` comes into play. `SELECT-OPTIONS` defines an internal range table for a data object.

To better understand how `SELECT-OPTIONS` works, we need to first understand the range table, and then we'll discuss multiple selection windows.

Range Table

A *range table* is an internal table with a special structure, the content of which can be analyzed with the relational operator `IN`. For example, to check if a particular material number falls within a given range of materials, we can use the following statement:

```
IF v_matnr IN rt_matnr.
ENDIF.
```

In this example, `v_matnr` contains the material number we want to check, and `rt_matnr` is a special range table that contains the range of materials. The `IN` operator automatically scans the range table to check if the value falls within the range.

A range table can be defined in the program via the `TYPE RANGE OF` addition with the `DATA` keyword—for example:

```
DATA rt_carrid TYPE RANGE OF s_carr_id.
```

A range table has four fields (columns): SIGN, OPTION, LOW, and HIGH. [Figure 6.3](#) shows the components of a range table as seen in debug mode; note the TYPE and length (LNGTH) of each field.

No.	Component name	Type	Lngth	Contents
1	SIGN	C	1	
2	OPTION	C	2	
3	LOW	C	3	
4	HIGH	C	3	

Figure 6.3 Components of Range Table

The four range table components can be described as follows:

► **SIGN**

This field is of TYPE `c` and LENGTH 1. The content of SIGN determines for every row whether the values in the LOW and HIGH fields are included or excluded. Valid values are I for include and E for exclude.

► **OPTION**

This field is of TYPE `c` and LENGTH 2. OPTION contains the relational operator for the selection condition of the row. Valid operators are EQ (equal to), NE (not equal to), GE (greater than or equal to), GT (greater than), LE (less than or equal to), LT (less than), CP (matches pattern), and NP (does not match pattern) if column HIGH is initial. If column HIGH is not initial, BT (between), NB (not between) are valid. For the options CP and NP, the data type of the columns LOW and HIGH must be `c`.

► **LOW**

The length and type of the LOW field depends on the data type reference provided while declaring the range table. In [Figure 6.3](#), the range table is defined with reference to `s_carr_id`, so it's of TYPE `c` and LENGTH 3. This column is designated for the comparison value or the lower interval limitation.

► **HIGH**

Similar to the LOW field, the LENGTH and TYPE of the HIGH field depends on the data type reference provided while declaring the range table. This column is designated for the upper interval limitation.

The system automatically analyzes the row of a range table when used with the IN operator. IN operators are very useful when a complex comparison of values is required.

When we use `SELECT-OPTIONS`, the system defines an internal range table with a header line in the program and creates two input fields, low and high, on the current selection screen.

To define a `SELECT-OPTIONS` field, use the `FOR` addition with the `SELECT-OPTIONS` keyword. The system expects a data object after `FOR`—for example:

```
DATA v_carrid TYPE s_carr_id.
SELECT-OPTIONS s_carrid FOR v_carrid.
```

This code creates a range table with a header line for which the `LOW` and `HIGH` fields are of type `v_carrid` in the program. It also creates two input fields on the selection screen, `s_carrid-low` and `s_carrid-high`, with a `MULTIPLE SELECTION` button, as shown in [Figure 6.4](#)

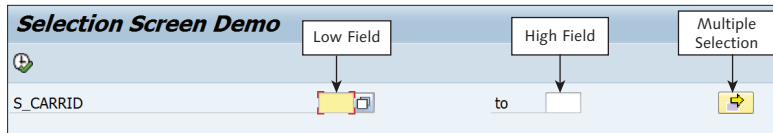


Figure 6.4 Range Field on Selection Screen

Multiple Selection Window

With `SELECT-OPTIONS`, the user can provide a value range or use the `MULTIPLE SELECTION` button to enter a more complex selection criterion. [Figure 6.5](#) shows the `MULTIPLE SELECTION` dialog box that's presented upon clicking the `MULTIPLE SELECTION` button.

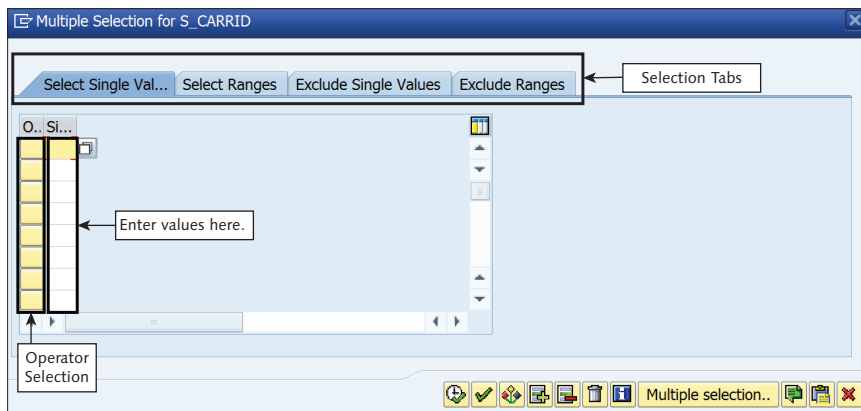


Figure 6.5 Multiple Selection Dialog Box

The MULTIPLE SELECTION dialog box includes four tabs to input selection criteria:

► SELECT SINGLE VALUES

The SELECT SINGLE VALUES tab allows the user to input multiple single values to be considered for selection. The user can also select the options for each value by using the OPTIONS button, as shown in [Figure 6.6](#).

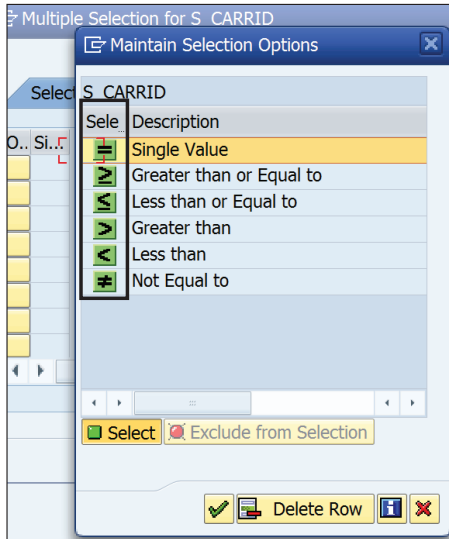


Figure 6.6 Options Button for Single Values

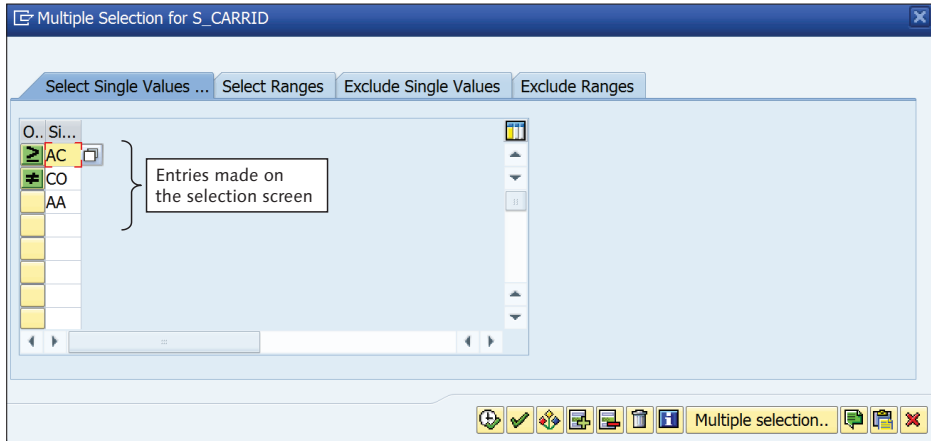
The entries made in this tab will be stored in the range table with SIGN as I (include), OPTION as the operator selected (default is EQ), and LOW field with the input value. The HIGH field for this row will be empty.

[Figure 6.7](#) shows the rows of the SELECT-OPTIONS range table as seen in debug mode for the corresponding entries made in the SELECT SINGLE VALUES tab.

When used with the IN operator, the system automatically processes the row as per the values in the SIGN, OPTION, LOW, and HIGH fields.

► SELECT RANGES

This tab allows you to enter values in a range that should be included in the search. The operator allows you to specify whether the values are inside the range or outside the range. [Figure 6.8](#) shows the options available for ranges.



Internal table `s_carrid[]`

1	SIGN	OPTION	LOW	HIGH
1	I	GE	AC	
2	I	NE	CO	
3	I	EQ	AA	

Entries stored in the range table

Figure 6.7 Range Table in Debug Mode for Multiple Selection Dialog Box Entries

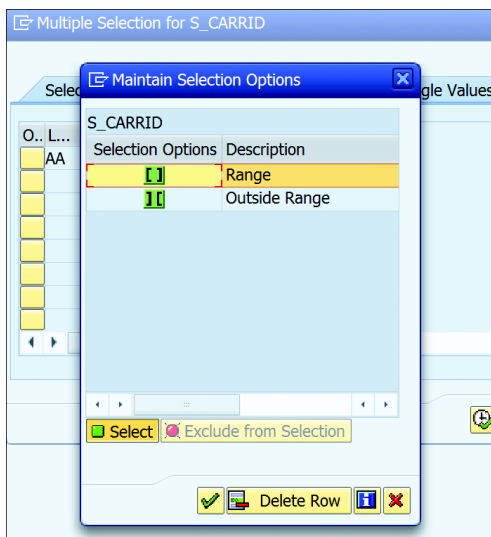


Figure 6.8 Operator for Ranges

The SIGN field of the range table will be I, OPTIONS will either be BT or NB based on the options selected for the row, the LOW field will be filled with the low value, and the HIGH field will be filled with the high value entered on the screen. The high value entered on the screen should be greater than the low value; otherwise, the screen will throw an error message. Figure 6.9 shows the corresponding entries in the range table for the values entered on the screen.

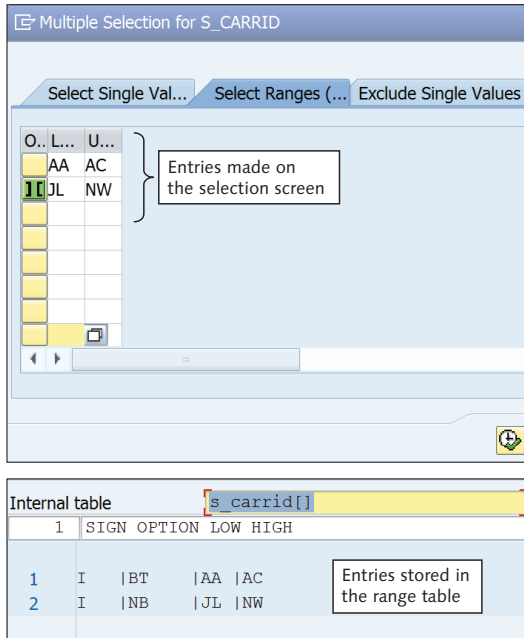


Figure 6.9 Range Table Values per Select Ranges Tab

► **EXCLUDE SINGLE VALUES**

The entries in this tab are similar to those of the SELECT SINGLE VALUES tab, with the exception that the values maintained in this tab are excluded from selection. In other words, the SIGN field of the row will be filled with E (exclude).

► **EXCLUDE RANGES**

The entries in this tab are similar to those of the SELECT RANGES tab, with the exception that the values maintained in this tab are excluded from selection.

The SIGN field for rows in SELECT VALUES (both single values and ranges) will be filled with I, and EXCLUDE VALUES (both single values and ranges) will be filled with E.

Let's consider a requirement in which the user wants to run a report to process billing documents. The user's selection criteria are as follows:

1. Select document numbers 100, 122, 140.
2. Select all the documents from 200 to 400 and from 500 to 600.
3. Exclude document numbers 222 and 320 from the previous range.
4. Exclude document numbers 520 to 540.

For the example selection criteria, the user will make the entries in the multiple selection window in different tabs, as shown in [Figure 6.10](#).

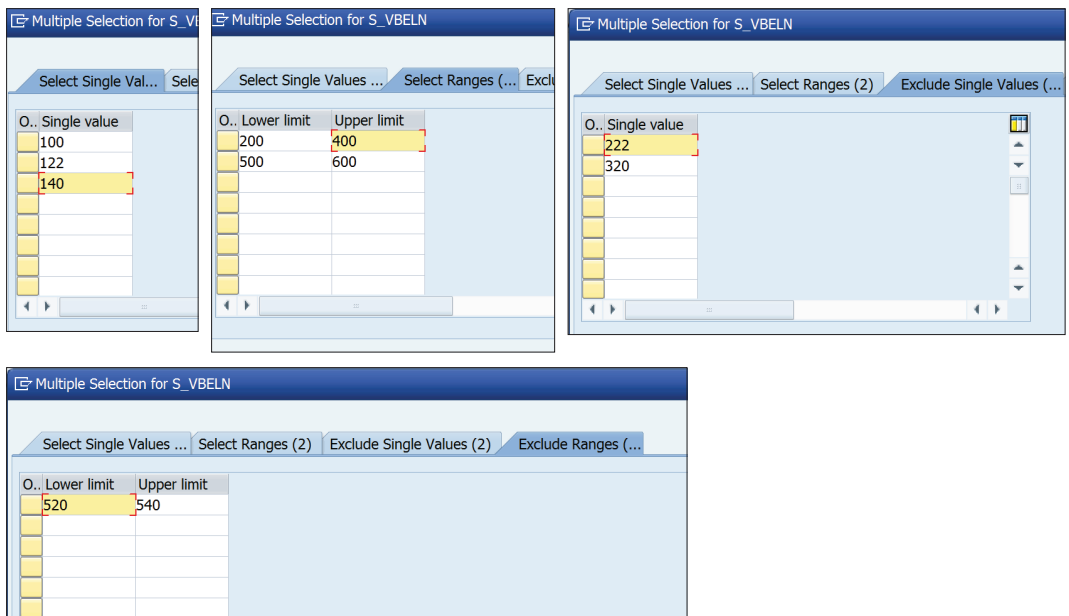


Figure 6.10 Entries in Different Tabs of Multiple Selection Window

The entries in [Figure 6.10](#) will be transferred to the range table, as shown in [Figure 6.11](#). Notice the value in the SIGN field for the entries made in the SELECT and EXCLUDE tabs.

For the entries made in the SELECT tabs of the MULTIPLE SELECTION window, the SIGN field is filled with the value I, and for the entries made in the exclude tabs of the MULTIPLE SELECTION window, the SIGN field is filled with the value E. The

values in the LOW and HIGH fields are automatically filled with leading zeros with the help of a conversion routine attached to the domain.

Internal table		s_vbeln[]		Type
1	SIGN	OPTION	LOW	HIGH
1	I	EQ	0000000100	
2	I	EQ	0000000122	
3	I	EQ	0000000140	
4	I	BT	0000000200 0000000400	
5	I	BT	0000000500 0000000600	
6	E	EQ	0000000222	
7	E	EQ	0000000320	
8	E	BT	0000000520 0000000540	

Figure 6.11 Corresponding Entries in Range Table for Selection Criteria in [Figure 6.10](#)

The code in [Listing 6.6](#) shows the usage of `SELECT-OPTIONS` to process these multiple documents. In this code, we're processing the basic header data of the documents provided in the selection criteria. The code highlights the declaration of `SELECT-OPTIONS` and its subsequent use in the `SELECT` query to filter the records per the user selection criteria using the `IN` operator.

```

TYPES : BEGIN OF ty_vbrk,
        VBELN TYPE VBELN_VF, " Document Number
        FKART TYPE FKART,   " Billing Type
        FKDAT TYPE FKDAT,   " Billing Date
        NETWR TYPE NETWR,   " Net Value
        KUNRG TYPE KUNRG,   " Payer
      END OF ty_vbrk.
DATA : it_vbrk TYPE STANDARD TABLE OF ty_vbrk,
      wa_vbrk TYPE ty_vbrk.
SELECT-OPTIONS s_vbeln FOR wa_vbrk-vbeln.
SELECT  VBELN
        FKART
        FKDAT
        NETWR
        KUNRG
      FROM VBRK
      INTO TABLE it_vbrk
      WHERE vbeln IN s_vbeln.
LOOP AT it_vbrk INTO wa_vbrk.
WRITE:/ 'DOCUMENT NUMBER:' , wa_vbrk-vbeln,
      / 'Billing Type :' , wa_vbrk-fkart.
      / 'PAYER:' , wa_vbrk-kunrg,
      / 'BILLING DATE :' , wa_vbrk-fkdat,

```

```
    / 'NET VALUE :' , wa_vbrk-netwr LEFT-JUSTIFIED.
```

```
ENDLOOP.
```

Listing 6.6 Sample Code to Process Multiple Billing Documents

6.1.3 SELECTION-SCREEN

The SELECTION-SCREEN keyword has several uses. It creates stand-alone selection screens and selection screen layouts, and it uses elements from other selection screens. We'll provide some examples of using this keyword to define selection screen layouts in this section. A more in-depth discussion of the SELECTION-SCREEN keyword can be found in [Chapter 14](#).

The SELECTION-SCREEN keyword can be supplied with various layout options to influence the screen layout:

► **SKIP**

Creates a blank line.

Example: SELECTION-SCREEN SKIP.

► **ULINE**

Creates a horizontal line.

Example: SELECTION-SCREEN ULINE.

► **BEGIN OF BLOCK**

This option allows you to organize the selection screen elements into multiple blocks. The block is closed with the END OF BLOCK statement, and it supports defining frames and titles for the frames. Arranging the screen elements into multiple blocks is very useful on selection screens with many fields, as shown in [Listing 6.7](#).

```
SELECTION-SCREEN BEGIN OF BLOCK b1 WITH FRAME
                                     TITLE text-001.
PARAMETERS : p_carrid TYPE s_carr_id,
              p_connid TYPE s_conn_id.
SELECTION-SCREEN END OF BLOCK b1.

SELECTION-SCREEN BEGIN OF BLOCK b2 WITH FRAME
                                     TITLE text-002.
PARAMETERS : p_alv RADIOBUTTON GROUP rb1,
              p_list RADIOBUTTON GROUP rb1.
SELECTION-SCREEN END OF BLOCK b2.
```

Listing 6.7 Organizing Selection Screen Fields into Multiple Blocks

The code in [Listing 6.7](#) generates the selection screen layout shown in [Figure 6.12](#). The frame titles are maintained using text elements.

Figure 6.12 Selection Screen Layout Generated by Code in Listing 6.7

6.1.4 Selection Texts

We can maintain labels for selection screen fields via selection texts. Selection texts are maintained in the ABAP program by following the menu path **GoTo • TEXT ELEMENTS • SELECTION TEXTS**.

[Figure 6.13](#) shows the selection text maintenance screen.

Figure 6.13 Selection Text Maintenance Screen

Here, you can manually maintain the text—or pick it from ABAP Data Dictionary, if the screen field is referred to a data element. To pick the text from ABAP Data

Dictionary, simply select the checkbox to the right of the corresponding field in the `DICTIONARY REF.` column, as shown in [Figure 6.13](#). This checkbox is disabled if the field is not referred to any ABAP Data Dictionary object.

Click the `ACTIVATE` icon in the application toolbar to apply the changes. The selection screen fields will show the label, as shown in [Figure 6.12](#). The selection screen is generated when the program is activated. If any changes are made to the selection screen fields, the program should be activated to regenerate the selection screen and reflect the changes. The selection screen of the program should not be edited manually using the Screen Painter tool, because any changes made manually will be lost when the selection screen is regenerated.

6.2 Messages

Messages allow you to communicate with users from your programs. They're used mainly when the user has made an invalid entry on a screen. Messages are raised in the program code via the `MESSAGE` statement. There are different types of messages that can be displayed based on requirements. The message text can be maintained locally in the program or globally using a message class.

An example of a `MESSAGE` statement in use is as follows:

```
MESSAGE 'The input is invalid' TYPE 'E'.
```

The addition `TYPE 'E'` in the statement specifies it as an error message.

In this section, we'll look at the various types of messages that can be displayed in an ABAP program. We'll also explore the different ways messages can be displayed and how messages can be translated to other languages.

6.2.1 Types of Messages

Message processing depends on the message type specified in the `MESSAGE` statement and the program context in which the statement occurs.

The system behavior after a particular message type is sent is context-dependent. [Table 6.1](#) lists all the possible types of messages that can be sent from ABAP programs. The type of message shown influences how the program behaves in a post-error situation and defines how the ABAP runtime should process the message.

Always use the right message type for the condition, and keep the message informative enough for the user to understand the error.

It's highly recommended to always validate the user inputs and show specific and relevant messages to the user to identify any issues with input values.

Message Type	Meaning	Notes
A	Termination message	The message appears in a dialog box and the program terminates. When the user has confirmed the message, the control returns to the next-highest area menu. This type of message should be limited to situations in which the error can't be handled by the current task or when system-related errors occur.
E	Error message	Depending on the program context, an error dialog appears or the program terminates. Program execution won't continue unless the error is corrected. These types of messages are used for input validations or when program execution can't proceed due to an error.
I	Information message	The message appears in a dialog box. Once the user has confirmed the message, the program continues immediately after the MESSAGE statement. This type of message is used to show informative messages, such as successful database updates and so on.
S	Status message	The program continues normally after the MESSAGE statement, and the message is displayed in the status bar of the next screen. This type of message is used to display the status of any action.
W	Warning message	Depending on the program context, an error dialog appears or the program terminates. The user can press <code>[Enter]</code> to continue with program execution. This type of message is used to warn the user of missing information.

Table 6.1 Types of Messages

Message Type	Meaning	Notes
X	Exit message	No message is displayed, and the program terminates with a short dump. Program terminations with a short dump normally only occur when a runtime error occurs. Message type X allows you to force a program termination. The short dump contains the message ID.

Table 6.1 Types of Messages (Cont.)

The type of message is specified in the MESSAGE statement directly—for example:

```
MESSAGE 'The input was invalid' TYPE 'E'
MESSAGE 'Database update successful' TYPE 'I'
MESSAGE 'Key parameters are missing' TYPE 'W'
```

6.2.2 Messages Using Text Symbols

One major disadvantage of using text literals to maintain a message in program code is that they can't be reused in the program or customized for the user logon language. By using text symbols, we can reuse the messages within the program and maintain translations wherein the translated text can be automatically picked up by the system based on a user's logon language.

To maintain text symbols, follow the menu path GOTO • TEXT ELEMENTS • TEXT SYMBOLS in ABAP Editor, as shown in [Figure 6.14](#).

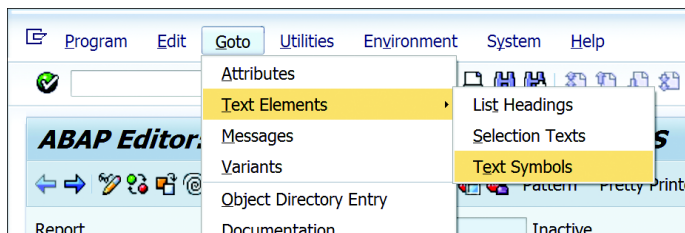


Figure 6.14 Menu Path to Maintain Text Symbols in ABAP Program

On the CHANGE TEXT SYMBOLS screen (see [Figure 6.15](#)), you can assign a unique three-digit alphanumeric value to each text symbol that can then be called from the ABAP program.

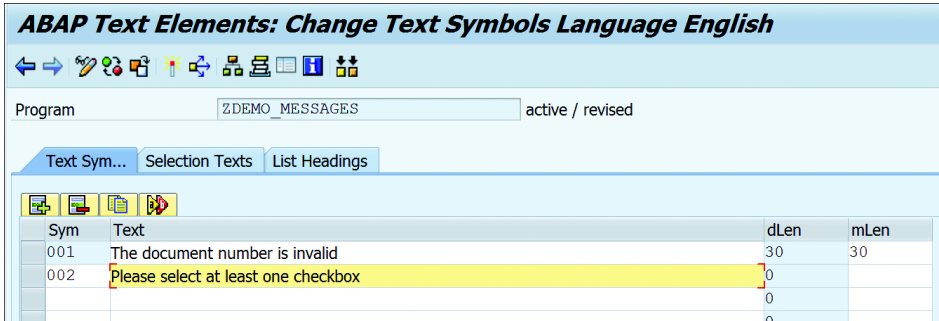


Figure 6.15 Change Text Symbol Screen

The text symbol editor also has the option to maintain translations for each text via the menu path **GoTo • TRANSLATION** from the **CHANGE TEXT SYMBOLS** screen, as shown in [Figure 6.16](#).

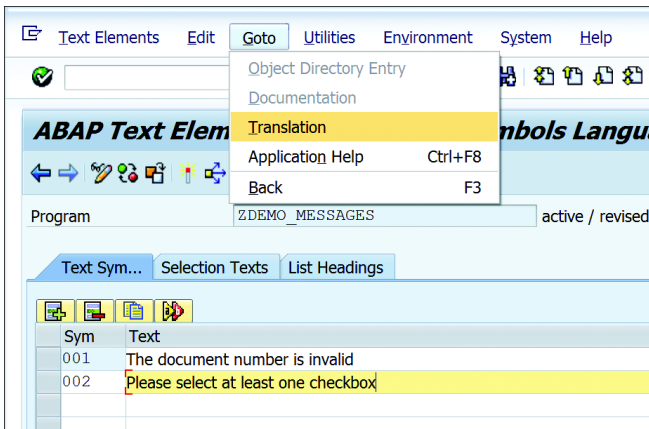


Figure 6.16 Menu Path to Access Translation Editor

To use a text symbol in a message, simply use `text-nnn` to call the text symbol. Here, *nnn* is the three-digit text symbol number—for example:

```
MESSAGE text-001 TYPE 'E'.
```

The system will automatically take care of pulling the respective translated texts to support multiple logon languages. If there's no translation maintained for the given logon language, a blank message will be shown.

6.2.3 Messages Using Message Classes

Text symbols maintained in the program are local, so they can only be reused within the program. However, what if you need to share message texts across multiple related programs? Maintaining the same message texts and their translations in multiple programs can be redundant and a waste of time, because any changes to texts should also be manually maintained in each program. To maintain message texts globally, use a *message class*.

Global message texts are stored in table T100 and can be configured using Transaction SE91. When creating a new message class, it should be in a customer namespace, starting with Y or Z.

To create a new message class, open Transaction SE91. On the MESSAGE MAINTENANCE initial screen, enter a unique name for your message class and click the CREATE button, as shown in [Figure 6.17](#).

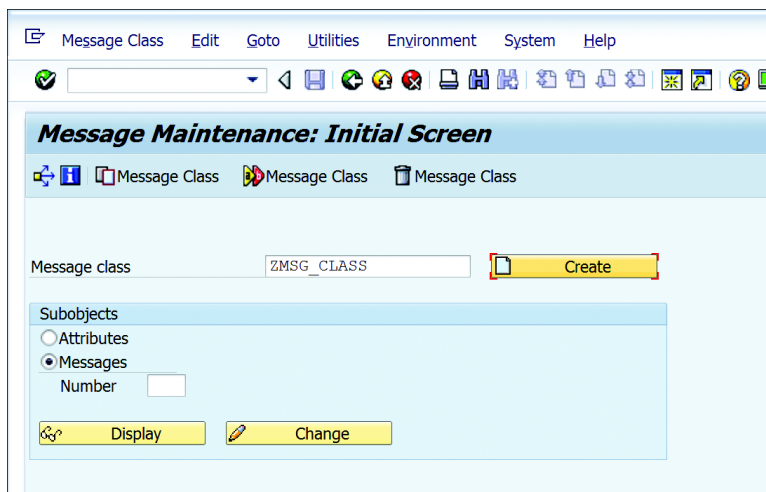


Figure 6.17 Message Maintenance Initial Screen

On the next screen, select the MESSAGES tab to edit your messages. The system will prompt you to assign a package when navigating to the MESSAGES tab, because a message class is a repository object. Message texts are similar to text symbols; each message text is assigned a unique three-digit alphanumeric value.

You can select the SELFEXPLANATORY checkbox (see [Figure 6.18](#)) if the message explains itself sufficiently, or you can create long texts for each message to show

more information about a message to the user. The long text is displayed when the user double-clicks the message (see [Figure 6.19](#)). The long text is maintained by clicking the LONG TEXT button, as shown in [Figure 6.18](#).

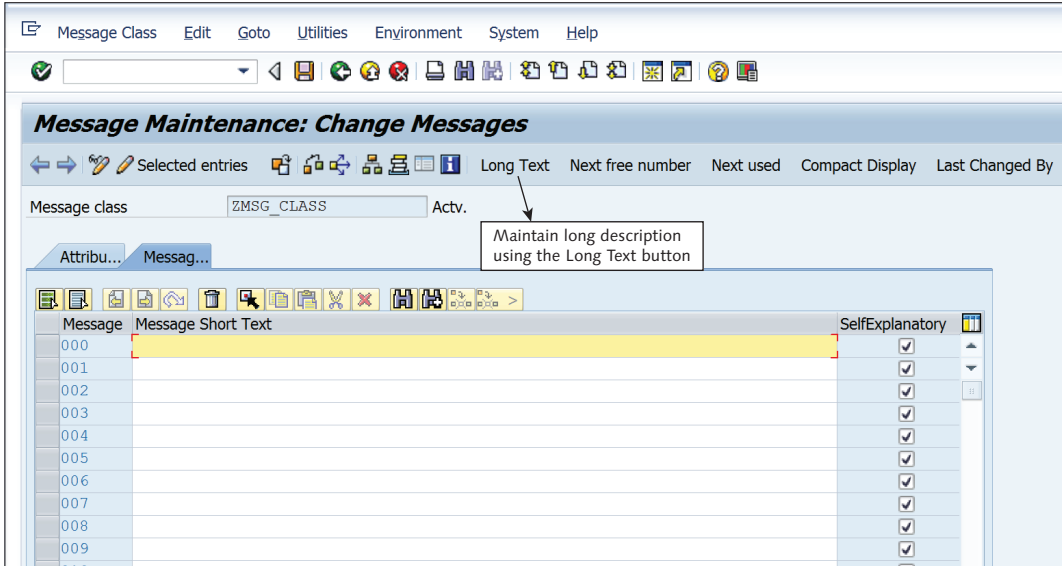


Figure 6.18 Maintaining Messages in Message Class

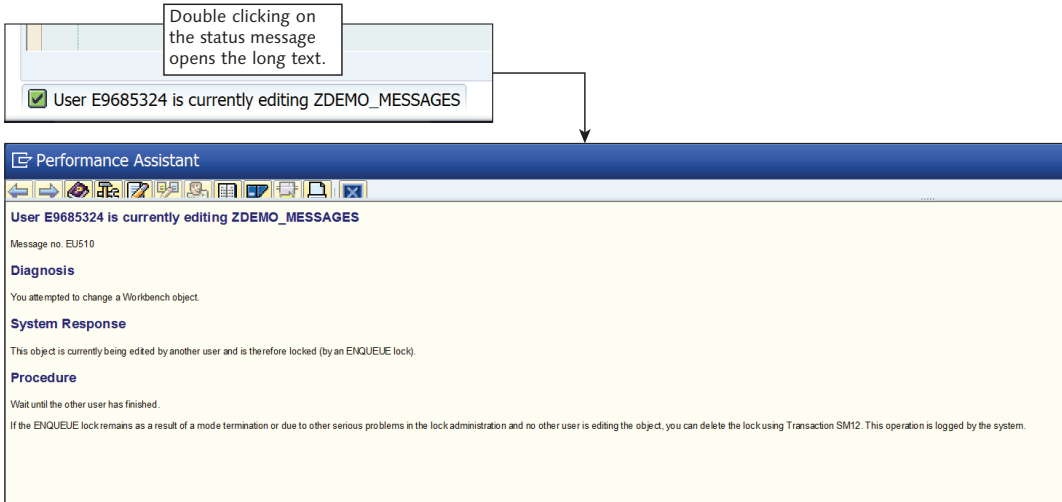


Figure 6.19 Long Text Displayed in Performance Assistant Window

The syntax to display a message from the message class is a bit different than displaying it using a text symbol. Here, the message number is preceded by the type of message as opposed to using the `TYPE` addition. The message class name is supplied in parenthesis—for example:

```
MESSAGE e000(ZMSG_CLASS).
```

In this statement, we're displaying the message with the message number 000, maintained in the message class `ZMSG_CLASS` as an error message. The message type `e` precedes the message number 000. You can use other message types similarly; for example, `w000` will display the message as a warning message.

The message class name can also be maintained in the `REPORT` statement for executable programs, using the `MESSAGE-ID` addition, instead of maintaining it in parenthesis for each message—for example:

```
REPORT ZDEMO_MESSAGES MESSAGE-ID zmsg_class.
```

```
MESSAGE E000.
```

You can maintain translations for each message via the menu path `GOTO • TRANSLATION` from the `MESSAGE MAINTENANCE` screen.

6.2.4 Dynamic Messages

You may have noticed that [Figure 6.19](#) contains some dynamic elements. The message shows the username and the program name as part of the message text. We can use an ampersand (&) as a placeholder in message text, and it can be replaced dynamically in the program via the `WITH` addition to the `MESSAGE` keyword. You can use up to four placeholders in the message text.

[Figure 6.20](#) shows the message with two placeholders that will be replaced with a value supplied in the `MESSAGE` statement via the `WITH` addition.

The message can be sent using the syntax `MESSAGE s000(ZMSG_CLASS) WITH sy-username sy-repid`. Here, `sy-username` is a system field that stores the username, and `sy-repid` is the system field that stores the current program name. The value of the variable `sy-username` will replace the first placeholder in the message, and `sy-repid` will replace the second placeholder. The fields are inserted sequentially into the message text in place of the placeholders.

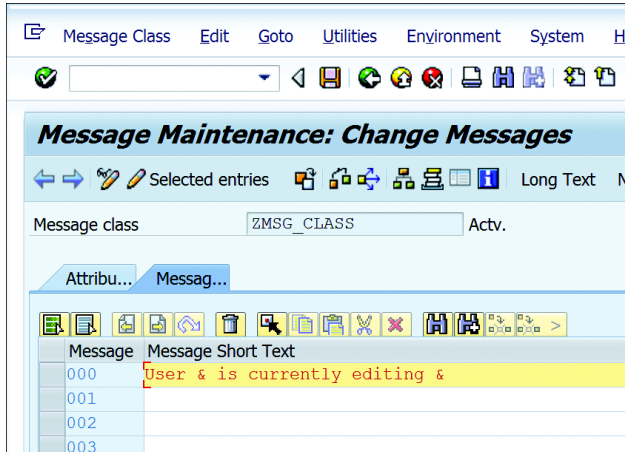


Figure 6.20 Using Placeholders in Message Text

Alternately, you can use the placeholders with position numbers such as &1, &2, &3, and &4 in the message to specify where the value is inserted in the message. Placeholders that are not supplied with a value via the WITH addition will be ignored.

Placeholders can also be used in the long text of a message by maintaining &V1& to &V4& as the placeholders. When the long text is called, the values supplied with the message will replace the placeholders in the long text.

6.2.5 Translation

Because the SAP system supports multiple logon languages, it's recommended to create translations for all texts maintained in the program (in case users with different logon languages access your program).

The translation editor can be accessed by following the menu path GOTO • TRANSLATION from ABAP Editor or from text maintenance screens like those for the message class or the text symbols. After selecting the menu path GOTO • TRANSLATION, the system presents a dialog box to specify the target language into which the text needs to be translated, as shown in [Figure 6.21](#).

Once the target language is selected, you can maintain translations manually for each text individually and click SAVE, as shown in [Figure 6.22](#).

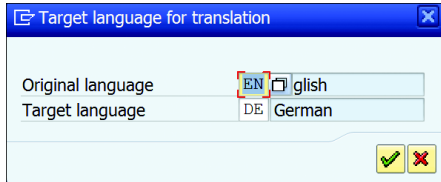


Figure 6.21 Target Language Selection

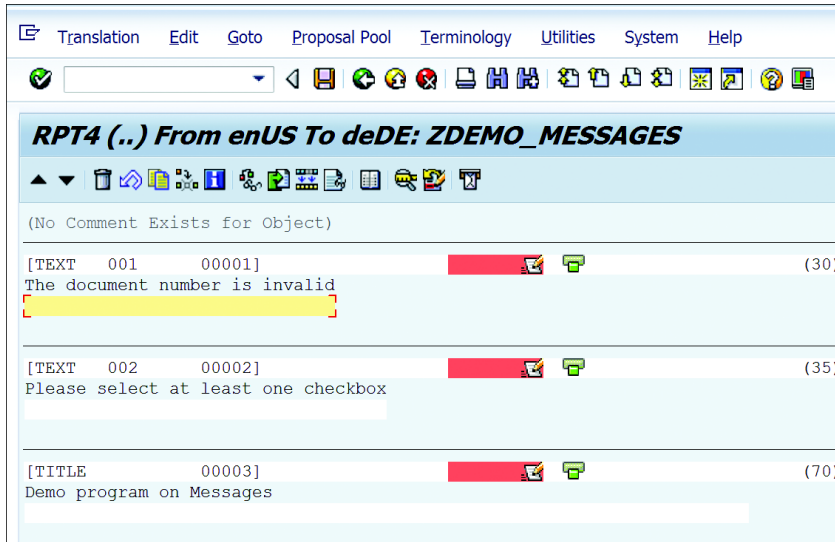


Figure 6.22 Maintain Translation Screen

Translations can be created for the program title, selection texts, text symbols, and list headings.

6.3 Summary

In this chapter, we discussed the concepts behind selection screens and messages. Understanding both these concepts is required to create your first complete report program. We also explored different ways to use the `PARAMETERS` and `SELECT-OPTIONS` keywords to create different selection screen elements. We've created the groundwork to create a report program, which we'll explore further with selection screens and their events in [Chapter 14](#).

We also looked at messages and the different types of messages that can be sent from your ABAP programs. We touched upon various ways to maintain messages and discussed both static and dynamic messages. Finally, we concluded by discussing translation. It's important to note that all texts can be translated to multiple languages.

We'll wrap up our discussion of ABAP basics in the next chapter, in which we'll discuss key concepts in modularization techniques. The concepts in the next chapter are crucial to understanding the advanced topics in ABAP.

ABAP programs are modular in nature. Taking advantage of modularization techniques will provide a number of advantages, including better readability and organization. In this chapter, we'll discuss ABAP modularization.

7 Modularization Techniques

Modularization is breaking down and organizing program code into a modular structure, rather than dealing with the program as a single, indivisible unit. There are many advantages to modularization: reusability of code, which helps in reducing code redundancy, and better organization of program code, which in turn makes your ABAP programs maintenance-friendly.

In this chapter, we'll discuss how the ABAP program flow and the runtime environment control program execution through events and procedures. We'll also look at the concept of processing blocks, their features, the statements that define processing blocks, how they're called, whether they can keep local data, and ABAP programs in which they can be defined.

In [Section 7.1](#), we'll begin with an overview of modularization in general. We'll introduce key concepts and provide an example to get you started. [Section 7.2](#) then explores the program structure, including the use of event blocks, dialog modules, and procedures.

[Section 7.3](#) dives into the different event type categories: program constructor events, reporting events, selection screen events, list events, and screen events. Then, in [Section 7.4](#), we take a look at procedures—specifically, subroutines, function modules, and methods. Finally, we end the chapter in [Section 7.5](#), using inline declarations for declaring data objects inline.

7.1 Modularization Overview

ABAP programs are modular in nature, allowing you to break down the program functionality into smaller modules. These modules are known as *processing blocks*. Processing blocks can be put together to achieve the overall functionality of a program.

To better illustrate this process, let's look at an example. Say you're developing an application for your HR department, and one of the requirements of the application is to find the day an employee joined the organization. This is just a *part* of the overall functionality the application provides, but this feature is required in many applications in the HR module.

To find the weekday of a given date, you may have to write some forty lines of code. If you're a new developer, you may spend a few hours or days getting the logic right. If you chose to implement the code directly in the main program, you'll have to repeat the code in all the other applications that require similar functionality.

Worst of all, if none of the other developers in your organization are aware that you've already spent hours or days implementing the logic for this requirement, they may try to implement the same logic on their own, which leads to a situation in which multiple people are spending time working on the same task.

By implementing this functionality in a module (let's say a *function module*) that can be called from any program, you avoid code redundancy and save a lot of time and effort.

Modularizing your ABAP programs leads to many benefits, including the following:

- ▶ Improves code readability
- ▶ Eases maintenance
- ▶ Helps develop reusable software blocks
- ▶ Designs applications with scope for future enhancements
- ▶ Saves development effort
- ▶ Helps break down big, complex requirements into smaller, easy-to-maintain modules that can be combined to meet your requirements
- ▶ Splits development across team members without locking the code

As you spend more time developing ABAP programs, you may appreciate the advantages of modularization because it helps you simplify the development and maintenance of large applications.

In ABAP, we have various kinds of processing blocks. Some processing blocks are called by the ABAP runtime environment from outside the program, and some are called by ABAP statements from inside the program. The processing blocks that are called by ABAP statements within the program are called *procedures*.

Table 7.1 lists the various processing blocks and their key features.

Processing Block	Features
Called by ABAP Runtime Environment	
Event blocks	Called by the ABAP runtime environment when a predefined event is triggered while the program is running. These processing blocks are maintained in the program using specific event keywords.
Dialog modules	Called by the ABAP runtime environment from the screen flow logic while processing general screens. These processing blocks are maintained in the main program of the screen between <code>MODULE...ENDMODULE</code> statements and are called from the screen flow logic using the <code>MODULE</code> statement.
Called by ABAP Statements	
Subroutines	Useful for modularization within the program. Although subroutines can be called both internally from ABAP statements within the program or externally from other programs, they are typically used for reusability within a program. Calling subroutines externally is not recommended and should be avoided. Subroutines are maintained between <code>FORM...ENDFORM</code> statements and are called using the <code>PERFORM</code> statement.
Function modules	Procedures that are maintained globally in the system and can be called from any ABAP program. They are maintained using the Function Builder tool (Transaction SE37), which also serves as a central library to search for existing function modules. Function modules are maintained between <code>FUNCTION...ENDFUNCTION</code> statements in a <i>function pool</i> type of program and are called using the <code>CALL FUNCTION</code> statement. Although function modules work similarly to subroutines, they can only be defined globally, which in turn makes them easier to search for.

Table 7.1 Processing Blocks

Processing Block	Features
Methods	<p>Contain the functions of classes in ABAP Objects. Methods are defined in classes and support object-oriented programming concepts (see Chapter 8 for more information on object-oriented programming).</p> <p>Methods are maintained between <code>METHOD...ENDMETHOD</code> statements and are called using the <code>CALL METHOD</code> statement.</p> <p>Classes can be defined both locally within an ABAP program and globally in Class Builder (Transaction SE24). ABAP Objects is an important topic that we will discuss in detail in Chapter 8.</p>

Table 7.1 Processing Blocks (Cont.)

In general, you use subroutines for reusability within a program, function modules for reusability across multiple programs in the system, and methods to implement object-oriented programming concepts along with reusability.

Modularization is not just limited to processing blocks; you can also modularize your ABAP source code to organize the source code better with the help of *include programs* or *macros*:

► **Include programs**

Include programs are repository objects for which the sole purpose is to modularize source code. Include programs are built into other programs and can't be executed independently.

► **Macros**

Macros are another way to modularize code, but they're considered obsolete technology and shouldn't be used. Macros are difficult to debug, but can be used to run a statement more than once in a code.

To better understand include programs, think of them this way: If you want to use a similar set of lengthy data declarations in multiple programs, you can keep the declarations in an include program that can be included in any ABAP program. This way, you can reuse the source code without having to type the same code in multiple programs. This also helps maintain the code centrally.

In ABAP Objects, include programs are useful to split up classes into visibility parts, declarations, methods, and so on. The sole purpose of the include program is to modularize the source code. The ABAP runtime environment doesn't

recognize include programs as independent main programs; they're never directly executed and are used to be included in another main program.

Figure 7.1 shows the use of an include program.

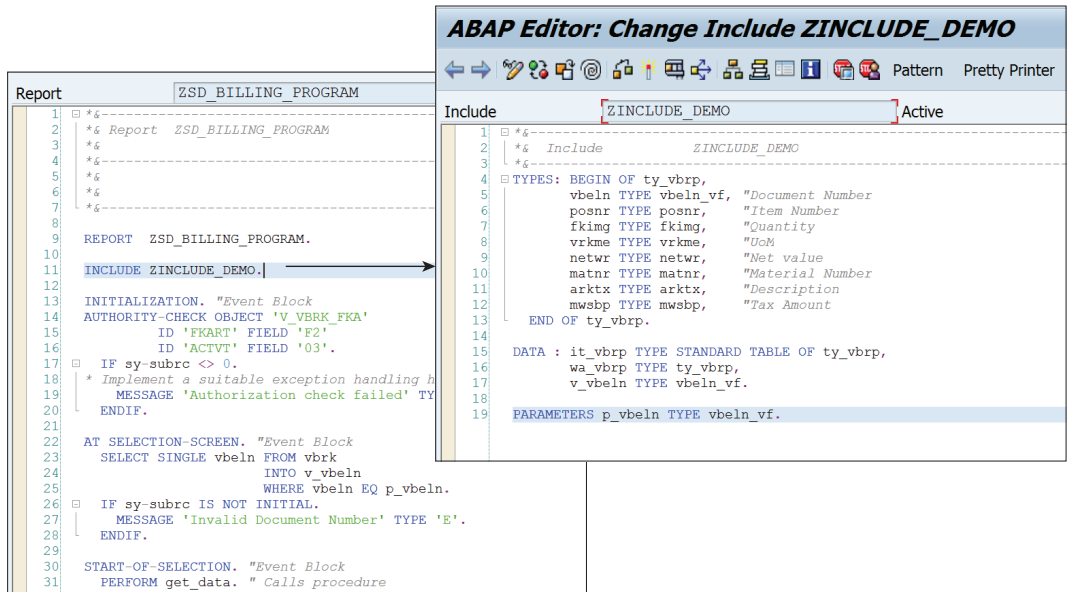


Figure 7.1 Include Program

Modularization in event blocks and dialog modules helps with the general execution of ABAP programs, and modularization in procedures and include programs helps improve the readability and maintainability of ABAP programs, and allow you to avoid code redundancies, reuse functionality, and encapsulation data.

Now that you have a basic understanding of the key concepts and methods used in modularization, we'll explore these techniques in greater depth in the remaining sections of this chapter.

7.2 Program Structure

ABAP was originally designed for the development of applications that process data from the database; users could interact with the application using SAP GUI screens. In a classic programming method based on screens (like reports and

transactions) that require user interaction, the control is passed back and forth between the presentation layer and the application layer. For example, when the user is making an input on the selection screen, the screen is active on the presentation layer, but there's no activity at this point in the application layer.

Once the user clicks the EXECUTE button, even though the selection screen is still displayed on the screen, it's not active. At this point, the application layer is actively processing the request and the presentation layer is idle. After the application layer processes the user request, the result is sent to the presentation layer. Now, the selection screen is active again, or a new screen is called.

The execution of the program between consecutive screens is called a *dialog step*. A dialog step in a report program starts when the user clicks the EXECUTE button on the selection screen and ends when the results are displayed on the list screen. Again, once the user performs an action on the list screen, a new dialog step starts, which ends when a new screen is called and displayed to the user.

In such a scenario, you can't construct the program as a single sequential unit. You must divide the program into dialog steps, with each step comprising the programming logic between two successive screens.

Because ABAP programs are modular in structure, we divide the program into modules that can be assigned to the individual dialog steps. Each module is called a *processing block*, which consists of a set of ABAP statements. In other words, when you run a program, you're calling a series of processing blocks.

In this section, we'll discuss the use of processing blocks, event blocks, dialog modules, and procedures in the ABAP program structure.

7.2.1 Processing Blocks

In general, processing blocks can be broadly classified as event blocks, dialog modules, or procedures.

In the following subsections, we'll discuss the use of processing blocks in the program structure before discussing the different types of processing blocks that can be used.

Declaration Area

Each ABAP program consists of a global declaration area followed by a number of processing blocks. The objects defined in the global declaration part are visible in all processing blocks of the ABAP program. In some processing blocks (such as procedures), data declarations can be made within the processing block that are locally visible.

Listing 7.1 shows an example of global and local declarations.

```
REPORT ZCB_EXAMPLE.
*Global declarations of the program
DATA : f1 TYPE c VALUE 'A',
      f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output.
*Local declaration that can be accessed (visible) within
*this subroutine
  DATA f3 TYPE c VALUE 'C'.
  WRITE f1.
  WRITE f2.
  WRITE f3.
ENDFORM.
```

Listing 7.1 Local and Global Data

In Listing 7.1, we defined data objects `f1` and `f2` in the global declaration area of the program. The program contains a subroutine with the name `OUTPUT`, within which we declared the data object `f3`. Because the data object `f3` is declared within the subroutine `OUTPUT`, it's local to this subroutine. In other words, the data object `f3` is locally visible within the subroutine `OUTPUT` and can be accessed only by ABAP statements that are part of the subroutine.

You can't access this data object `f3` from other processing blocks of the program. However, the global data objects `f1` and `f2` can be accessed by ABAP statements from any processing block of the program.

Using Processing Blocks

Processing blocks are indivisible units that contain ABAP statements. For example, in Listing 7.1, `END-OF-SELECTION` is an event block, which is a type of processing

block that contains the ABAP statement `PERFORM. FORM OUTPUT . . . ENDFORM` is a subroutine, which is also a type of processing block that contains the `WRITE` statements.

Note

Processing blocks can't be nested, so you can't include one processing block in another processing block. For example, ahead in [Listing 7.8](#), you can't include the `END-OF-SELECTION` processing block within the `FORM OUTPUT...ENDFORM` processing block. However, you can call one processing block from another processing block. In [Listing 7.1](#), we're calling the `OUTPUT` processing block from the `END-OF-SELECTION` processing block using the `PERFORM` statement.

Calling Processing Blocks

Processing blocks are called either externally by the ABAP runtime environment or internally by specific statements in the program.

For example, `END-OF-SELECTION` is an event block called by the ABAP runtime environment when the program is running, and `FORM OUTPUT` is a procedure called by the `PERFORM` statement in the code.

Sequence of Processing Blocks

When we divide program code into various processing blocks, the program code isn't executed in the sequence in which the processing blocks are defined in the program. When a particular processing block is called depends totally on the ABAP runtime environment (in the case of event blocks and modules) or when a specific processing block call statement is executed (in the case of procedures).

Therefore, the order in which the processing blocks are maintained in the program source code is totally irrelevant to the sequence in which they're executed. Once the processing block is called, the code in a processing block is executed sequentially.

For example, in [Listing 7.1](#), the order in which the processing blocks are maintained in the code is irrelevant. [Listing 7.2](#) shows the same code but rearranges the order of the `END-OF-SELECTION` and `FORM OUTPUT` processing blocks in the program; it returns the same output.


```

REPORT ZCB_EXAMPLE.
*Global declarations of the program.
DATA : f1 TYPE c VALUE 'A',
       f2 TYPE c VALUE 'B'.
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output.
*Local declaration that can be accessed (visible) within
*this subroutine
  DATA f3 TYPE c VALUE 'C'.
  WRITE f1.
  WRITE f2.
  WRITE f3.
ENDFORM.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine

```

Listing 7.2 Changing Processing Block Order

After executing the code in a processing block, control is returned to its caller.

Ending Processing Blocks

The execution of a processing block ends when the last statement of the processing block has been executed. However, it can also be ended programmatically by using certain statements, like `CHECK`, `EXIT`, or `RETURN`. For example, after checking for a certain condition, if you don't want to execute the remaining code in a processing block if the condition isn't met, then you can use the `CHECK`, `EXIT`, or `RETURN` statement to end the processing block without executing the remaining code in the processing block.

The `CHECK` and `EXIT` statements behave differently if they're used inside a loop statement (`DO` loop, `WHILE` loop, `LOOP AT`, etc.). When these statements are used within a loop, they don't end the processing block; they only affect the loop processing. For example, the `CHECK` statement will quit the current loop and process the next iteration when used in a loop.

However, the `EXIT` statement will quit the loop processing completely. Both the `EXIT` and `CHECK` statements will continue executing the remaining code in the processing block after loop processing. The `RETURN` statement always quits the processing block irrespective of whether it's used inside a loop or outside the loop.

No matter how the processing block ends, on each termination control is returned to the caller of the processing block.

Listing 7.3 shows the usage of the `EXIT` statement. In Listing 7.3, we're using the `EXIT` statement within the subroutine `OUTPUT`. The statement `WRITE f3` maintained after the `EXIT` statement will not be executed, because the program will quit the processing block when it reaches the `EXIT` statement.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
      f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output.
DATA f3 TYPE c VALUE 'C'.
  WRITE f1.
  WRITE f2.
EXIT.
  WRITE f3.
ENDFORM.
```

Listing 7.3 Using `EXIT` Statement within Processing Block

Listing 7.4 shows an example using the `EXIT` statement within a loop. In this example, we're using the `EXIT` statement inside a `DO` loop. After the first loop, the system will quit the loop when it encounters the `EXIT` statement but will continue executing the remaining code in the subroutine.

Therefore, in this case, the statement `WRITE f3` will be executed.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
      f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output.
DATA f3 TYPE c VALUE 'C'.
DO 3 TIMES.
  WRITE f1.
  WRITE f2.
EXIT.
```

```

ENDDO.
  WRITE f3.
ENDFORM.

```

Listing 7.4 EXIT Statement within Loop

Listing 7.5 shows an example using the CHECK statement to quit the processing block. In this example, we're using the CHECK statement to check that the contents of the data objects f2 and f3 do not match.

If the condition is satisfied, the remaining code of the processing block will be executed; otherwise the system will quit the processing block without executing further code in the processing block. In our example, the statement WRITE f3 won't be executed, because f2 and f3 have the same value.

```

REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
      f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output.
  DATA f3 TYPE c VALUE 'B'.
  WRITE f1.
  WRITE f2.
  CHECK f2 <> f3.
  WRITE f3.
ENDFORM.

```

Listing 7.5 Using CHECK Statement in Processing Block

Listing 7.6 shows an example using the CHECK statement within a loop. In the example, we're using the CHECK statement within a DO loop to check if the contents of the data objects f1 and f2 are the same.

When the system encounters the CHECK statement, it will execute the statements after the CHECK statement only if the CHECK condition is satisfied. If the condition is not satisfied, the system will skip to the next loop pass without executing further code in the current loop pass.

Once the loop processing is complete, the system will continue executing the remaining code in the processing block. Therefore, in this example, the WRITE f1

statement is executed three times, the `WRITE f2` statement is never executed, and the `WRITE f3` statement is executed once.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
      f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output.
  DATA f3 TYPE c VALUE 'B'.
DO 3 TIMES.
  WRITE f1.
CHECK f1 = f2.
  WRITE f2.
ENDDO.
  WRITE f3.
ENDFORM.
```

Listing 7.6 CHECK Statement within Loop

Listing 7.7 shows an example using the `RETURN` statement. In this example, we're using the `RETURN` statement within the `DO` loop. Because the `RETURN` statement will always quit the processing block irrespective of whether it's used within a loop or outside the loop, the `WRITE f1` statement is executed once, and the `WRITE f2` and `WRITE f3` statements are never executed.

```
REPORT ZCB_EXAMPLE.
DATA : f1 TYPE c VALUE 'A',
      f2 TYPE c VALUE 'B'.
END-OF-SELECTION.
  PERFORM output. "Calling a subroutine
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output.
  DATA f3 TYPE c VALUE 'B'.
DO 3 TIMES.
  WRITE f1.
RETURN.
  WRITE f2.
ENDDO.
  WRITE f3.
ENDFORM.
```

Listing 7.7 RETURN Statement

Statements

All ABAP program statements, except global data declarations (also referred to as *virtual processing blocks*), belong to a processing block. If you write any statements between the global declaration area and a processing block (the statement is not explicitly assigned to a processing block and is not a declarative statement), they're automatically assigned to the beginning of the `START-OF-SELECTION` processing block. The `START-OF-SELECTION` processing block plays the role of a default processing block in executable programs.

Listing 7.8 contains sample code using various processing blocks. Here, we're showing item data for a given billing document number.

```

TYPES: BEGIN OF ty_vbrp,
        vbeln TYPE vbeln_vf, "Document Number
        posnr TYPE posnr,    "Item Number
        fking TYPE fking,    "Quantity
        vrkme TYPE vrkme,    "UoM
        netwr TYPE netwr,    "Net Value
        matnr TYPE matnr,    "Material Number
        arktx TYPE arktx,    "Description
        mwsbp TYPE mwsbp,    "Tax Amount
      END OF ty_vbrp.

DATA : it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
        wa_vbrp LIKE LINE OF it_vbrp,
        v_vbeln TYPE vbeln_vf.

PARAMETERS p_vbeln TYPE vbeln_vf.

INITIALIZATION. "Event Block
AUTHORITY-CHECK OBJECT 'V_VBRK_FKA'
  ID 'FKART' FIELD 'F2'
  ID 'ACTVT' FIELD '03'.
IF sy-subrc <> 0.
* Implement a suitable exception handling here
  MESSAGE 'Authorization check failed' TYPE 'E'.
ENDIF.

AT SELECTION-SCREEN. "Event Block
  SELECT SINGLE vbeln FROM vbrk
    INTO v_vbeln
    WHERE vbeln EQ p_vbeln.
  IF sy-subrc IS NOT INITIAL.
    MESSAGE 'Invalid Document Number' TYPE 'E'.
  ENDIF.

START-OF-SELECTION. "Event Block

```

```

PERFORM get_data. "Calls procedure
PERFORM show_output. "Calls procedure
*&-----*
*&      Form  GET_DATA
*&-----*
FORM get_data . "Subroutine
  SELECT vbeln
         posnr
         fking
         vrkme
         netwr
         matnr
         arktx
         mwsbp
         FROM vbrp
         INTO TABLE it_vbrp
         WHERE vbeln EQ p_vbeln.
ENDFORM.              " GET_DATA

*&-----*
*&      Form  show-output
*&-----*
FORM show_output . "Subroutine
  FORMAT COLOR COL_HEADING.
  WRITE : / 'Item',
         10 'Description',
         33 'Billed Qty',
         48 'UoM',
         57 'Netvalue',
         70 'Material',
         80 'Taxamount'.
  FORMAT COLOR OFF.
  LOOP AT it_vbrp INTO wa_vbrp.
    WRITE :/ wa_vbrp-posnr,
           10 wa_vbrp-arktx,
           33 wa_vbrp-fking LEFT-JUSTIFIED,
           48 wa_vbrp-vrkme,
           57 wa_vbrp-netwr LEFT-JUSTIFIED,
           70 wa_vbrp-matnr,
           80 wa_vbrp-mwsbp LEFT-JUSTIFIED.
  ENDLOOP.
ENDFORM.              " SHOW-OUTPUT

```

Listing 7.8 Code Showing Various Processing Blocks

In the report that results from [Listing 7.8](#), the user inputs the document number of a billing document on a selection screen, and based on that document number, the item details are fetched from table `VBRP`.

In this code, we're using five processing blocks: three event blocks (`INITIALIZATION`, `AT SELECTION-SCREEN`, `START-OF-SELECTION`) and two procedures (`FORM get_data` and `FORM show_output`).

The event blocks allow us to control the sequence in which our code is executed. For example, the code related to the authorization check is maintained under `INITIALIZATION` event block (we can also use another event block, `LOAD-OF-PROGRAM`, as an alternative). Because this event is always triggered when the program is initialized, we can ensure that the first element checked when the user runs the report is whether that user has the required authorizations to run the report.

Similarly, the `AT SELECTION-SCREEN` event is triggered when the user clicks the `EXECUTE` button on the selection screen. By keeping the code to validate the user input under this event block, we can ensure that the program execution doesn't continue without a valid document number.

We've also separated the selection logic from output logic by maintaining two procedures in the program: `FORM get_data` and `FORM show_output`. These subroutines are called by the `PERFORM` statements in the `START-OF-SELECTION` event block.

Thus far, we've looked at how processing blocks are used in an ABAP program structure. In doing so, we've already touched on the different types of processing blocks available. In the next section, we'll take a closer look at these processing block types.

Types of Processing Blocks

As previously mentioned, processing blocks can be classified as event blocks, dialog modules, or procedures. The differentiating factors among the types of processing blocks lie in how they're introduced in the program and how they're called. The type of processing block also establishes whether or not it can keep local data and its support for the parameter interface.

Notice that in [Listing 7.8](#), although procedures are closed with their own `END` statements (`FORM...ENDFORM`), event blocks are introduced with event keywords and end implicitly by the beginning of the next processing block, by the end of the program, or with the first definition of a modularization unit. In addition, the event blocks are called by the ABAP runtime environment automatically when

the program is running. Alternately, subroutines (procedures) are called with the specific `PERFORM` keyword in the code under the `START-OF-SELECTION` event block.

Note

As discussed earlier, you can call processing blocks from outside the ABAP program or by using ABAP commands (which are themselves part of a processing block).

Event blocks behave differently from other processing blocks. Event blocks are always called by the ABAP runtime environment, and the call is triggered by an event. For example, user actions on selection screens and lists trigger certain events for which corresponding event blocks are called. The ABAP runtime environment also triggers events (while the ABAP program is running) that can be processed in ABAP programs.

For example, an event triggers when the program is loaded into the memory, before a selection screen is sent. You can make the program react to these events by defining the corresponding event blocks in your program and ignoring an event block if you don't want the program to react to that event. This is unlike a subroutine call, where you must have a corresponding subroutine. In other words, you aren't forced to react to a particular event, because the event is simply ignored if a corresponding event block is not maintained in your ABAP program, while allowing you to react to an event by maintaining the event block.

Please note that event blocks and dialog modules do not have a local declaration area. Any data declaration within these blocks are automatically added to the global declaration area of the program, making these blocks behave like global data (visible in all internal processing blocks). However, any data declarations within procedures (methods, subroutines, function modules) are called *local data declarations* and are only visible within the procedure in which they are declared. They are created when the procedure starts and destroyed when the procedure ends.

Processing blocks can be maintained in any order in the program. The code in [Listing 7.9](#) shows the same code as in [Listing 7.8](#), but the order of the processing blocks is changed to highlight the fact that the order of the processing blocks in the program is irrelevant.

```
TYPES: BEGIN OF ty_vbrp,
        vbeln TYPE vbeln_vf, "Document Number
        posnr TYPE posnr,   "Item Number
        fking TYPE fking,   "Quantity
```



```

        vrkme TYPE vrkme,      "UoM
        netwr TYPE netwr,     "Net Value
        matnr TYPE matnr,     "Material Number
        arktx TYPE arktx,     "Description
        mwsbp TYPE mwsbp,     "Tax Amount
    END OF ty_vbrp.

DATA : it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
      wa_vbrp TYPE ty_vbrp,
      v_vbeln TYPE vbeln_vf.
PARAMETERS p_vbeln TYPE vbeln_vf.

AT SELECTION-SCREEN. "Event Block
    SELECT SINGLE vbeln FROM vbrk
        INTO v_vbeln
        WHERE vbeln EQ p_vbeln.
    IF sy-subrc IS NOT INITIAL.
        MESSAGE 'Invalid Document Number' TYPE 'E'.
    ENDIF.

START-OF-SELECTION.
    PERFORM get_data. "Calls procedure
    PERFORM show_output. "Calls procedure

INITIALIZATION. "Event Block
AUTHORITY-CHECK OBJECT 'V_VBRK_FKA'
    ID 'FKART' FIELD 'F2'
    ID 'ACTVT' FIELD '03'.
    IF sy-subrc <> 0.
* Implement suitable exception handling here
        MESSAGE 'Authorization check failed' TYPE 'E'.
    ENDIF.

*&-----*
*&      Form show-output
*&-----*
FORM show_output . "Subroutine
    FORMAT COLOR COL_HEADING.
    WRITE : / 'Item',
           10 'Description',
           33 'Billed Qty',
           48 'UoM',
           57 'Netvalue',
           70 'Material',
           80 'Taxamount'.
    FORMAT COLOR OFF.
    LOOP AT it_vbrp INTO wa_vbrp.
        WRITE :/ wa_vbrp-posnr,
              10 wa_vbrp-arktx,
              33 wa_vbrp-fking LEFT-JUSTIFIED,

```

```

        48 wa_vbrp-vrkme,
        57 wa_vbrp-netwr LEFT-JUSTIFIED,
        70 wa_vbrp-matnr,
        80 wa_vbrp-mwsbp LEFT-JUSTIFIED.
    ENDLOOP.
ENDFORM.                " SHOW-OUTPUT

*&-----*
*&      Form  GET_DATA
*&-----*
FORM get_data . "Subroutine
    SELECT vbeln
           posnr
           fking
           vrkme
           netwr
           matnr
           arktx
           mwsbp
           FROM vbrp
           INTO TABLE it_vbrp
           WHERE vbeln EQ p_vbeln.
ENDFORM.                " GET_DATA

```

Listing 7.9 Code from [Listing 7.8](#), Rearranged to Change Order of Processing Blocks

Demo Program							
Item	Description	Billed Qty	UoM	Netvalue	Material	Taxamount	
000010	Sony Sunny01	5,000	PC	8.150,00	M-01	1.222,50	
000020	Sony Xal	5,000	PC	9.440,00	M-02	1.416,00	
000030	Highscreen MS 1775P	4,000	PC	9.400,00	M-10	1.410,00	
000040	MAG DX 15F/Fe	4,000	PC	5.848,00	M-12	877,20	

Figure 7.2 Output of Code in [Listing 7.8](#) and [Listing 7.9](#)

The code in [Listing 7.8](#) and [Listing 7.9](#) behave exactly the same and produce the same results (refer back to [Figure 7.2](#)).

The following sections discuss the different processing blocks in greater depth.

7.2.2 Event Blocks

Event blocks are introduced in the program through event keywords. Each event has a specific keyword, and we can use the respective event keyword in

the program to make it react to that event. When the event is triggered by the ABAP runtime environment, the code under the event keyword is executed.

An event block ends implicitly when the next processing block starts, as opposed to specifically by its own keyword. Therefore, unlike procedures, we don't have to use any `END` statement to end the event block explicitly.

Event blocks are internal to the program and can't be called programmatically using any ABAP statements. They're always triggered by events in the ABAP runtime environment. They don't support local data or parameter interfaces. When you use declarative statements within an event block, the system will not throw any syntax errors, but rather treats the declarative statement as a global declaration.

If we maintain an appropriate event block in the ABAP program for an event of the ABAP runtime environment, the code in the event block is executed; otherwise, the event is simply ignored.

Conversely, the code in an event block won't be executed unless the associated event takes place while the program is being executed. In other words, just because you've maintained an event block in your program, that doesn't guarantee its execution. It's only executed if the associated event is triggered while the program is running.

For example, the `AT LINE-SELECTION` event is triggered when the user double-clicks any output line on the list screen. If you want your program to react to the double-click event, you can maintain this event keyword in your program and write the corresponding code under it.

When the user double-clicks the output line, the code under the `AT LINE-SELECTION` event block is executed. However, if the user doesn't double-click the output line, this code is never executed. For example, in the following code, the statement `CALL TRANSACTION 'MM03'` will be executed when user double-clicks a line in the list output; if the user doesn't double-click, this statement is never executed:

```
AT LINE-SELECTION.  
CALL TRANSACTION 'MM03'.
```

We'll discuss all the available events and their keywords in [Section 7.3](#).

7.2.3 Dialog Modules

Dialog modules are implemented in executable programs, module pool programs, or function groups between `MODULE` and `ENDMODULE`.

We use dialog modules when working with dialogs (screens). Because executable programs, module pools, and function groups are the only programs that support screens, we can use dialog modules only in these type of programs. Note that you can't define screens and modules with ABAP Objects.

You maintain dialog modules in the main program where a screen is defined and call these modules from the screen flow logic using the `MODULE` keyword, as shown in [Figure 7.3](#).

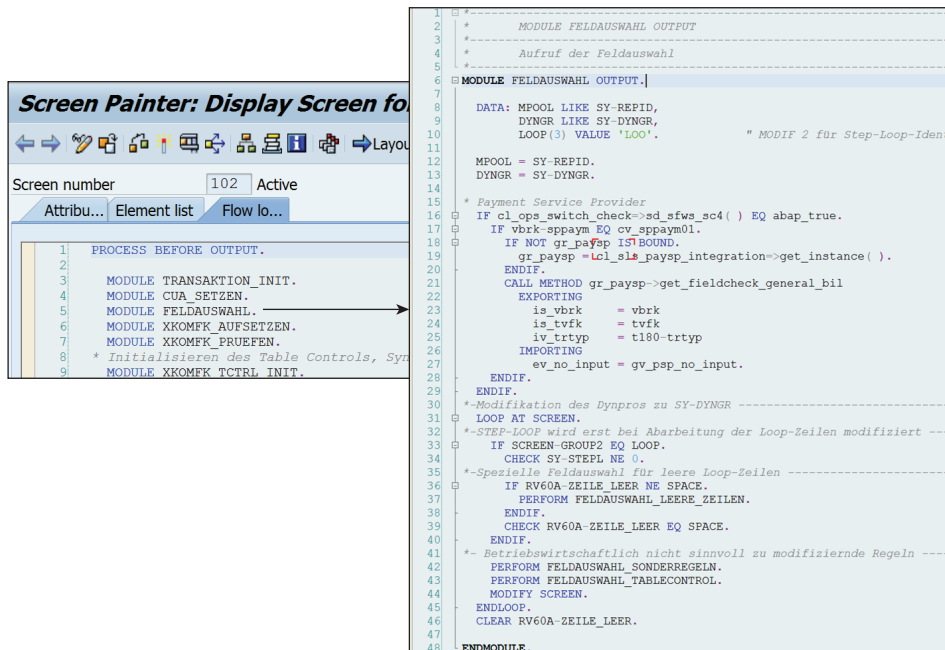


Figure 7.3 Module Call

Like event blocks, dialog modules have no local data area and no parameter interface. A data declaration in a dialog module is also added to the global data.

We'll discuss dialog modules further when we cover dialog programming in [Chapter 12](#).

7.2.4 Procedures

Procedures are callable processing blocks—which is to say that they're called by ABAP statements in the program. Procedures support a mechanism to import and export data (parameter interface) for the calling program and contain local data area. You can exchange data with the procedure using its parameter interface. They can be called internally and externally, meaning that they can be called from within the same program or externally from other ABAP programs.

The main purpose of procedures is to create reusable software blocks. For example, say you need a specific function in many programs or many times within the same program. You can simply create a procedure and call it where you need that functionality, instead of manually developing the same program logic many times. SAP provides many standard procedures out of the box that can be called in your ABAP programs.

Understanding procedures is crucial, because doing so helps you take advantage of many SAP-delivered reusable libraries and lays the foundation to grasp certain advanced concepts in ABAP, such as BAPIs, forms, SAP List Viewer (ALV), and so on. For example, BAPIs are technically implemented as remote function calls (RFCs), so developing a BAPI requires an understanding of function modules. Similarly, an understanding of ABAP Objects helps you work with ALV Object Models.

7.3 Events

Events can be broadly classified into the following categories:

- ▶ Program constructor events
- ▶ Reporting events
- ▶ Selection screen events
- ▶ List events
- ▶ Screen events

The next sections take a look at each of these event types.

7.3.1 Program Constructor Events

The `LOAD-OF-PROGRAM` event block serves as a *program constructor*. A program constructor serves the same purpose for ABAP programs as a constructor serves for a class (i.e., to initialize program variables).

This type of event block is called by the ABAP runtime environment when an executable program, module pool program, function group, or subroutine pool is loaded into memory (internal session). This event is triggered only once per program execution. If the program call opens a new internal session for each call (such as when using the `SUBMIT` or `CALL TRANSACTION` statements to call another program), then the `LOAD-OF-PROGRAM` event is triggered once per every program call.

To make the program react to this event, we use the `LOAD-OF-PROGRAM` event keyword, which is the first event to be called before any other ABAP code is processed. The statements in this block allow you to initialize the data of the program.

The `LOAD-OF-PROGRAM` event is mainly used when calling external procedures to initialize the global data of the called procedure. For example, in [Listing 7.10](#) the data object `DATE` is initialized with the current date via the `LOAD-OF-PROGRAM` event.

By maintaining the statement `DATE = SY-DATUM` under the `LOAD-OF-PROGRAM` event, we ensure that the data object `DATE` is always initialized with the current date when the program is executed.

```
REPORT ZCB_EXAMPLE.
DATA date TYPE d.
LOAD-OF-PROGRAM.
date = sy-datum.
```

Listing 7.10 Using `LOAD-OF-PROGRAM` to Initialize Data

7.3.2 Reporting Events

Reporting events can be used in executable programs (report programs). Executable programs are event-driven. When you run an executable program, a series of events are triggered, one after the other, by the ABAP runtime environment. You can define event blocks to react to these events. Reporting events that can be used in an executable type of program include the following:

- ▶ INITIALIZATION
- ▶ START-OF-SELECTION
- ▶ GET table
- ▶ GET table LATE
- ▶ END-OF-SELECTION

In the following subsections, we'll take an in-depth look at each of these reporting events.

INITIALIZATION

INITIALIZATION is the first reporting event triggered when an executable program is running. If the program also has a LOAD-OF-PROGRAM event block, then the INITIALIZATION event block is executed immediately after the LOAD-OF-PROGRAM event block.

If the executable program has a selection screen defined, an INITIALIZATION event is called after the ABAP selection screen code has been processed, but before the selection screen is displayed to the user.

Therefore, you can use it to initialize input fields of the selection screen or change the default values of these at runtime before the user gets to enter data into them, as shown in [Listing 7.11](#).

```
PARAMETERS p_carrid TYPE s_carr_id DEFAULT 'LH'.
INITIALIZATION.
p_carrid = 'AA'.
```

Listing 7.11 INITIALIZATION Event

However, an INITIALIZATION event is triggered only once per program execution, so any subsequent call to the selection screen will not execute the code under the INITIALIZATION event. It just works when the selection screen is called by the ABAP runtime environment for the first time. For this reason, we recommend using selection screen-related events to initialize or modify the selection screen fields and using the INITIALIZATION event to initialize all of the values in the program.

In [Listing 7.11](#), the `p_carrid` parameters field is assigned a default value `LH`. However, this value can be changed at runtime via the `INITIALIZATION` event; as a result, the user will see `AA` as the default value on the selection screen.

START-OF-SELECTION

The `START-OF-SELECTION` event is called after the `INITIALIZATION` event, when an executable program is running. If the program has a selection screen defined, this is the first event that is triggered after all selection screen processing has been completed (after executing all selection screen events). Selection screen events are executed between `INITIALIZATION` and `START-OF-SELECTION` events.

The `START-OF-SELECTION` event plays the role of a standard event, which means that if no other events are declared, there's no need to manually code this event. All processing statements will be automatically assigned to an implicit `START-OF-SELECTION` block, as shown in [Figure 7.4](#).

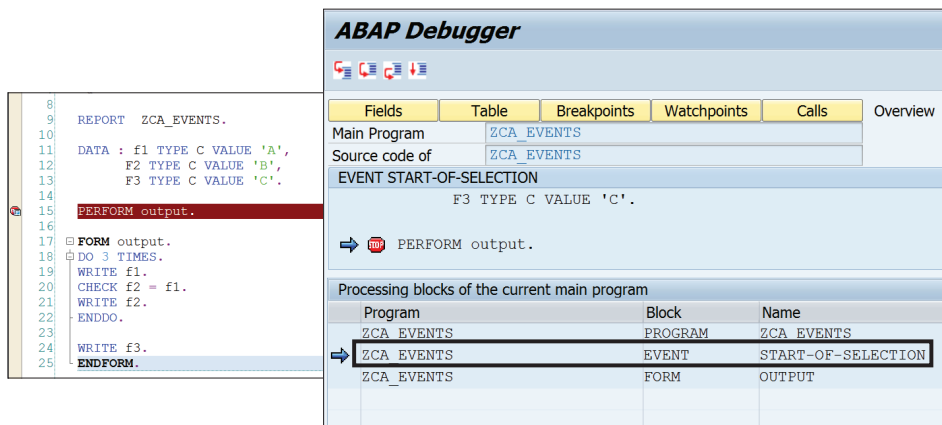


Figure 7.4 `START-OF-SELECTION` as Standard Event

Similarly, if you have any statements between the global declaration area and the first processing block of the program, they're automatically inserted at the beginning of this event block and executed.

In [Figure 7.5](#), the statement at line 28 is not explicitly assigned to any event, so the system automatically adds it to the top of the `START-OF-SELECTION` event and executes it before the `PERFORM` statement at line 46.


```

22| DATA : it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
23|       wa_vbrp TYPE ty_vbrp,
24|       v_vbeln TYPE vbeln_vf.
25|
26| PARAMETERS p_vbeln TYPE vbeln_vf.
27|
28| ► CLEAR wa_vbrp.
29|
30| INITIALIZATION. "Event Block
31| AUTHORITY-CHECK OBJECT 'V_VBRK_FKA'
32|       ID 'FKART' FIELD 'F2'
33|       ID 'ACTVT' FIELD '03'.
34| □ IF sy-subrc <> 0.
35|   * Implement a suitable exception handling here
36|   MESSAGE 'You do not have the authorization to run the report' TYPE 'E'.
37|   ENDF.
38|
39| AT SELECTION-SCREEN. "Event Block
40| SELECT SINGLE vbeln FROM vbrk INTO v_vbeln WHERE vbeln EQ p_vbeln.
41| □ IF sy-subrc IS NOT INITIAL.
42|   MESSAGE 'Invalid Document Number' TYPE 'E'.
43|   ENDF.
44|
45| START-OF-SELECTION.
46|   PERFORM get_data. " Calls procedure
47|   PERFORM show_output. "Calls procedure
48|
49|

```

Figure 7.5 START-OF-SELECTION Acts as Default Event

In general, you put all the data selection logic of your ABAP program within the START-OF-SELECTION event. Here, the program starts selecting values from tables. In a typical report program, you initialize the program data under the INITIALIZATION event, validate the selection screen inputs under the selection screen events, and, if everything looks fine, proceed with the data selection (based on the selection criteria) under the START-OF-SELECTION event.

GET table

The GET table event is used when the executable program uses a logical database for data selection. A *logical database* is a special ABAP program that retrieves data from the database and makes it available to the application program. In other words, when you use a logical database in your program, you do not need to code the SQL statements to retrieve the data from the database.

The logical database itself contains the Open SQL statements that retrieve the data from the database and pass the data line by line to the *interface work area* of your application program.

A logical database consists of a program with selection screen and authorization checks. When a logical database is used for the report program, the program can automatically inherit the selection screen and authorization checks.

Interface work areas are defined using the `TABLES` or `NODES` keywords, and they work similarly to the work areas defined using the `DATA` keyword, with just one basic difference. These interface work areas are created in the shared data area of between the programs. In other words, when you define a work area using the `DATA` keyword, it is defined in the memory space of the program. Interface work areas are defined in the shared memory space of the session, which all the programs running in the same session can have access to. For this reason, using `TABLES` is not recommended, as it can produce side effects.

In older versions of the SAP ERP system, the only way to use a logical database was to link it to an executable program and use `GET` events to retrieve the data. However, since release SAP R/3 4.5A, it's been possible to call a logical database in any program using the `LDB_PROCESS` function module. This module allows you to call many logical databases in your program, nested in any order. In this respect, the use of interface work areas is practically obsolete, available only for backward compatibility with earlier releases.

The `GET` event is triggered when the logical database has read a line from the table mentioned in the event keyword and made it available to the program in the work area declared using the `TABLES <table>` or `NODES <table>` statement (as seen in [Listing 7.12](#)).

```

NODES: SPFLI.
GET SPFLI.
  SKIP.
  WRITE: / 'From:', SPFLI-CITYFROM,
        'TO :', SPFLI-CITYTO.

```

Listing 7.12 Code Showing GET Event

GET <table> LATE

Once all of the data records for a node of the logical database have been read, the ABAP runtime environment triggers the `GET <table> LATE` event. By maintaining this event in the program, you can process the steps that should be executed at the end of the block. For example, if you want to print an itemized summary of sales orders, you can do so in this event block.

Listing 7.13 shows the implementation of this event block.

```

NODES: SPFLI, SFLIGHT, SBOOK.
DATA WEIGHT TYPE I VALUE 0.
GET SPFLI.
SKIP.
WRITE: / 'Carrid:', SPFLI-CARRID,
'Connid:', SPFLI-CONNID,
/ 'From: ', SPFLI-CITYFROM,
'To: ', SPFLI-CITYTO.
ULINE.
GET SFLIGHT.
SKIP.
WRITE: / 'Date:', SFLIGHT-FLDATE.
GET SBOOK.
WEIGHT = WEIGHT + SBOOK-LUGGWEIGHT.
GET SFLIGHT LATE.
WRITE: / 'Total luggage weight =', WEIGHT.
ULINE.
CLEAR WEIGHT.

```

Listing 7.13 Code Showing GET <table> LATE Event

In Listing 7.13, we're printing the date-wise summary of the total luggage weight for each airline code (CARRID) and flight connection number (CONNID). We retrieve the flight details using the GET SPFLI event. The GET SFLIGHT event loops through all the flights for a given date (the user can restrict the range on the logical database selection screen).

The GET SBOOK event retrieves the luggage weight for each booking from table SBOOK for the respective flight. Within this event, we're aggregating the total weight for the given date.

Finally, the GET SFLIGHT LATE event is called at the end once all the records for the flights in a given date are processed. In this event, we're printing the total luggage weight for the given date for the respective CARRID and CONNID. For this example, we're using the logical database selection F1S, which you can check using Transaction SLDB.

END-OF-SELECTION

The END-OF-SELECTION event is used to mark the end of selection. If your report isn't using a logical database, then this event is used after a START-OF-SELECTION event to mark the end of selection and to continue with further processing of the

selected data, such as printing the selected data in the output. However, if your report is linked to a logical database, this event is called after the logical database has completely transferred all the records to the application program.

For example, in [Listing 7.13](#), if all the data from the logical database selection is updated in an internal table that you then want to process further, you can use the `END-OF-SELECTION` event to mark the end of data selection and implement the logic to further process the selected data. The `END-OF-SELECTION` event is called by the runtime environment after processing all the `GET` events.

Using this event becomes mandatory to separate the selection logic from the output logic when using a logical database. Otherwise, its purpose is purely for documentation and for making the code legible. We recommend using this event to mark the separation from selection logic to output logic even if no logical database is used in the program.

[Listing 7.14](#) shows an example using the `END-OF-SELECTION` event.

```

NODES: SPFLI, SFLIGHT, SBOOK.
DATA: gr_table TYPE REF TO cl_salv_hierseq_table.
DATA: it_spfli TYPE TABLE OF spfli.
DATA: it_sflight TYPE TABLE OF sflight.
DATA: it_binding TYPE salv_t_hierseq_binding.
DATA: wa_binding LIKE LINE OF it_binding.

```

```

GET SPFLI.
APPEND spfli to it_spfli.
GET SFLIGHT.
APPEND sflight to it_sflight.

```

```

END-OF-SELECTION.
wa_binding-master = 'CARRID'.
wa_binding-slave = 'CARRID'.
APPEND wa_binding TO it_binding.

```

```

wa_binding-master = 'CONNID'.
wa_binding-slave = 'CONNID'.
APPEND wa_binding TO it_binding.

```

```

cl_salv_hierseq_table=>factory(
EXPORTING
    t_binding_level1_level2 = it_binding
IMPORTING
    r_hierseq = gr_table
CHANGING
    t_table_level1 = it_spfli

```

```

    t_table_level2 = it_sflight ).
gr_table->display( ).

```

Listing 7.14 END-OF-SELECTION Event

In [Listing 7.14](#), we're filling the internal tables `IT_SPFLI` and `IT_SFLIGHT` within the `GET SPFLI` and `GET SFLIGHT` events, respectively. We're then displaying this data as a hierarchical sequential display using the ALV object model within the `END-OF-SELECTION` event.

The ALV object model makes it easy to generate the ALV output. We'll discuss the ALV object model further in [Chapter 15](#).

7.3.3 Selection Screen Events

Selection screens are special screens that allow the user to input the selection criteria for a program. These are typically used in report programs (executable programs), but can also be used in module pool programs or function groups.

Selection screen events typically help to prepare the selection screen and evaluate user actions on the screen. For example, we can use selection screen events to assign default values to the input fields at runtime, dynamically disable or enable certain fields, or validate user input and show appropriate messages to the user.

Selection screen events are triggered when a selection screen is defined for the program and is processed by the ABAP runtime environment. If there is no selection screen defined for the program, none of the selection screen events are triggered.

Selection screens are categorized as special screens that can be defined using ABAP statements (`PARAMETERS` or `SELECT-OPTIONS`) and are generated by the ABAP compiler automatically, rather than defined manually using the Screen Painter (like general screens).

Because selection screens are generated automatically, programmers don't have access to the screen flow logic in which they can define dialog modules to modify the screen or react to user actions. The ABAP runtime environment fully controls the screen processing, generating a number of selection screen events through which programmers can maintain the event blocks in the program to react to these events.

Screen flow logic is broadly divided into *process before output* (PBO) and *process after input* (PAI). PBO is triggered before the screen is displayed to the user, and PAI is triggered after user takes some action on the displayed screen (like clicking a button). The ABAP runtime environment triggers different events at PBO and PAI while processing the screen.

For example, `AT SELECTION-SCREEN OUTPUT` is a PBO event; we can edit this event block in the program to react to the event and execute the code to modify the screen or to assign default values. Similarly, `AT SELECTION-SCREEN` is a PAI event that is triggered once the user takes some action on the screen. We can use this event block to validate user inputs.

The basic form of the selection screen events is the `AT SELECTION-SCREEN` event. This event is triggered after the ABAP runtime environment has transferred all input data from the selection screen to the ABAP program.

We'll explore all the available selection screen events in [Chapter 14](#).

7.3.4 List Events

List screens are one of the three types of screens in the SAP system (along with dialog screens and selection screens). They are used to display output data and to support user interaction. Like selection screens, list screens are also created using ABAP statements. In addition, the output of the list screen can be sent to a printer.

List screens are generated by using the `WRITE`, `ULINE`, or `SKIP` statements, and the ABAP runtime environment calls list events while processing the list screen. List screens provide a freely definable area that can be filled with data. List screens are generally used to display report output; this is considered old-fashion. These screens mostly have been replaced by ALV technology in newer systems.

Because the only three types of programs that support screens are executable programs, module pool programs, and function groups, list events can be used only in these types of programs while a classical list is being generated or when the user performs an action on the displayed list.

There are five list events:

- ▶ `TOP-OF-PAGE`
- ▶ `TOP-OF-PAGE DURING LINE-SELECTION`

- ▶ END-OF-PAGE
- ▶ AT USER-COMMAND
- ▶ AT LINE-SELECTION

TOP-OF-PAGE and END-OF-PAGE events are triggered in PBO and can be used to print the header (top) and footer (end) information on the page.

AT USER-COMMAND and AT LINE-SELECTION are triggered in PAI and can be used to react to user actions on screen. TOP-OF-PAGE DURING LINE-SELECTION is triggered when preparing the secondary list. For example, AT USER-COMMAND is triggered when the user selects any of the application toolbar buttons, whereas AT LINE-SELECTION is triggered when the user double-clicks the output line. These events can be used to provide drill-down reports.

We'll discuss all the available list events in detail in [Chapter 13](#) when we cover list screens.

7.3.5 Screen Events

Screen events are triggered when processing general screens. *Special screens* (selection screens and list screens) are defined in ABAP programs using ABAP statements, but *general screens* are created using the Screen Painter. General screens are the most common screens used in transactions.

A general screen consists of the input/output fields and the screen flow logic. Like selection screens and list screens, general screens can be defined for executable programs, module pool programs, and function groups. General screens are the common screens used with module pool programs.

The screen flow logic is divided into the PBO event, which is processed before the screen is displayed, and the PAI event, which is processed after a user action on the screen.

There are four screen events triggered when the screen is processed:

- ▶ **Process before output (PBO)**

A PBO event is triggered before the screen is shown to the user and can be used to modify the screen.

► **Process after input (PAI)**

A PAI event is triggered when the user performs an action on the screen. This event can be used to evaluate user actions.

► **Process on help request (POH)**

A POH event is triggered when the user selects `[F1]` help on the screen field. This event can be used to show field documentation to help the user learn more about a field.

► **Process on value request (POV)**

POV is triggered when the user selects `[F4]` help on the screen field. This event can be used to show the possible entries for a field (a value list).

All the screen events other than PBO are triggered after showing the program screen.

We'll discuss all the available screen events in more detail in [Chapter 12](#) when we discuss dialog programming.

7.4 Procedures

Procedures are special modularization units that are called with ABAP statements. They provide reusable software blocks. Procedures contain a set of statements that are executed when called from ABAP programs.

As shown in [Figure 7.6](#), the ABAP program calls a procedure using specific ABAP statements (depending on the type of procedure being called). When the procedure is called, the code in the procedure is executed.

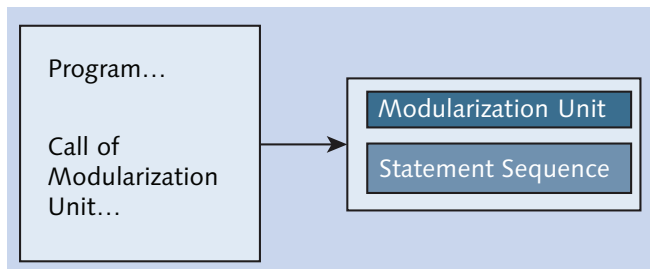


Figure 7.6 Call of Modularization Unit

Procedures are defined in ABAP programs, and upon activation of the program they remain as standalone modules. As discussed previously, procedures can be called from the same program in which they're maintained or from external programs. For example, you can call a subroutine of the same program using the `PERFORM` statement, as shown:

```
PERFORM prepare_data.
```

If the subroutine is maintained in a different program and you want to call it externally, you can do so by giving the program reference. The following statement calls an external subroutine (note that it isn't recommended to call subroutines externally, because they're not designed for external calls; for external calls, use function modules or ABAP classes):

```
PERFORM prepare_data IN PROGRAM ZCA_EXAMPLE.
```

When the procedure is called externally, the entire program in which the procedure is maintained is loaded into memory, but only the code inside the procedure is executed. Because procedures support the parameter interface (an interface to exchange data with the calling program), we can exchange data with the procedure to pass an input and receive a result.

Procedures support the local data area, the life of which is limited to the execution of the procedure. The data objects declared in the local data area of a procedure are created when the procedure is called and terminated once the execution of the procedure is completed.

For example, in the following code, the data object `F1` is defined within the subroutine `OUTPUT`. This data object `F1` is created when the subroutine is called and can be accessed by ABAP statements within the subroutine. Once the subroutine execution is completed (i.e., `ENDFORM` is executed), the data object `F1` is no longer available in the program memory and can't be accessed by any ABAP statements:

```
FORM output.  
DATA f1 TYPE C.  
ENDFORM.
```

Any data stored in the local data is cleared upon termination of the procedure. Local data objects are useful when you don't want to occupy the memory space of the main program unnecessarily, especially after the purpose of the data has been fulfilled.

For example, say you're writing code to select data from two different database tables, copying it into two internal tables, and finally consolidating the data to a final internal table. Once the data from these two internal tables is consolidated into the final table, you no longer need the data in the original internal tables.

If you defined these tables in the global declaration area, they continue to occupy memory throughout the runtime of the program, unless you manually clear the tables. However, if those internal tables are defined locally in a procedure, they're automatically cleared once the program exits the procedure.

ABAP contains the following kinds of procedures:

► **Subroutines**

Subroutines are primarily used for local modularization. Even though they can be called externally from other ABAP programs, they're generally called locally from the program from which they're defined. Subroutines are useful for reusability within the program, and they can be defined in any ABAP program.

► **Function modules**

Function modules are created globally and are used for global modularization. They're typically employed for reusability across multiple programs and are always called from a different program. Function modules are defined in a function group type of program but can be called from any ABAP program. They also support the RFC interface system and thus can be executed from remote systems.

► **Methods**

Methods support object-oriented programming concepts and contain the functions of classes and their instances in ABAP Objects. Methods are defined in classes and can be maintained locally in the program or globally in the system.

In the following subsections, we'll take a closer look at each of these different ABAP procedure types.

7.4.1 Subroutines

Subroutines are maintained between `FORM` and `ENDFORM` statements and are called using the `PERFORM` statement:

```
FORM subr.
...
ENDFORM.
```

The additions `USING` and `CHANGING` define the parameter interface for the subroutine. The parameter interface allows you to pass data to a subroutine and is defined by directly declaring variables on the `FORM` statement; these variables are called *formal* parameters. Formal parameters are filled by the `PERFORM` statement by maintaining the parameters in the `PERFORM` statement; such parameters are called *actual parameters*.

Parameters can be defined by referring either to local data types or to global types in ABAP Data Dictionary. The parameters defined exist as local variables to the subroutine; they're created when the subroutine is called and freed when it ends. If the subroutine has variables defined on the `FORM` statement, the `PERFORM` statement must pass a value to each of these variables in the defined sequence.

The parameter names that appear at the `FORM` level are called formal parameters, and the parameters that appear at the `PERFORM` level are called actual parameters.

The number of actual and formal parameters should always be same. For example, if a subroutine has three formal parameters, then it expects three actual parameters to be passed when the subroutine is called using the `PERFORM` statement. In [Listing 7.15](#), actual parameters `f1`, `f2`, and `f3` are passed to formal parameters `p1`, `p2`, and `p3`.

```
PERFORM abc USING f1 f2 f3.
```

```
FORM abc USING p1 p2 p3.
```

```
ENDFORM.
```

Listing 7.15 Example Syntax for Parameter Interface

While defining formal parameters, they can be either typed or untyped. If a formal parameter is referred to a data type, it is called a *typed formal parameter*; otherwise, it's an untyped formal parameter. In [Listing 7.15](#), all parameters are untyped.

When an *untyped formal parameter* is used, you can pass a variable of any data type or length to it. In such a case, the formal parameter uses the attributes of the actual parameter. For example, if you pass a four-byte integer to an untyped formal parameter `p1`, then `p1` becomes a four-byte integer. If you pass a character string of length three to the same parameter, it will become a three-character field. The data type and length of the untyped parameter is determined in runtime based on the actual parameter passed to it.

Even though using untyped parameters is flexible, it's no longer considered appropriate since the introduction of generic types. *Generic types* allow you to dynamically assign the typing for data objects. You'll learn more about generic types in [Chapter 16](#) when we discuss dynamic programming.

The following is the syntax for typed parameters:

```
FORM abc using u1 type t.
ENDFORM.
```

With typed parameters, you define the data type statically. Typed parameters are considered more efficient than untyped parameters. You must use generic types instead of untyped parameters if the requirement is to determine the data type dynamically during the runtime. For this reason, we discourage you from using untyped parameters.

The following points apply when working with typed formal parameters:

- ▶ The syntax of the `FORM` statement is a bit restricted, so you can only specify a data type on the `FORM` statement and can't use the `length` addition when using elementary types. For this reason, we recommend typing the formal parameters by referring them to either user-defined global types in the program or data types from ABAP Data Dictionary.
- ▶ The type and length of the actual parameter should match the formal parameter. You usually don't pass one data type to a different data type.
- ▶ Passing a variable of the wrong data type or length to a typed parameter causes a syntax error.
- ▶ Typed formal parameters are more efficient, because the system doesn't need to determine the data type at runtime to allocate memory.
- ▶ They help prevent coding errors, because the syntax checker will prevent you from passing incompatible parameters.
- ▶ They also help prevent runtime errors. If you perform an arithmetic operation on an untyped formal parameter and a character value is passed to it, it will result in a runtime error.
- ▶ You always use typed parameters when passing structures. If a structure is passed to an untyped formal parameter, you can't access the fields of the structure statically, because the system will identify the formal parameter as a structure only at runtime.

In the following subsection, we'll discuss how to pass parameters and internal tables to subroutines.

Passing Parameters to Subroutines

There are three different ways that parameters can be passed to subroutines:

- ▶ Parameters passed by reference (known as *pass by reference* parameters)
- ▶ Input parameters that pass values (known as *pass by value* parameters)
- ▶ Output parameters that pass values (known as *pass by value and result* parameters)

The `USING` and `CHANGING` additions for the `FORM` statement define the parameter interface and also determine how the parameters are passed. The same additions are also used for the `PERFORM` statement to pass the actual parameters, but purely for documentation purposes; they don't influence how the parameters are passed in that case.

The sequence of the additions is fixed. `USING` comes first, followed by `CHANGING`. You can use either one of the additions or both of them in the statement. Each addition can be followed by a list of any number of parameters.

The syntax at the `PERFORM` level is as follows:

```
PERFORM abc USING f1 f2 CHANGING f3
```

The parameters are listed either under `USING` or `CHANGING`, depending on whether the value of the actual parameter is changed after the subroutine call or remains unchanged. If the actual parameter is changed after the subroutine call, list it under `CHANGING`; otherwise, list it under `USING`.

Listing 7.16 shows an example of the `USING` addition.

```
DATA f1 TYPE c VALUE 'A'.
PERFORM abc USING f1.
FORM abc USING p1.
WRITE p1.
ENDFORM.
```

Listing 7.16 Passing Parameters with `USING` Addition

In [Listing 7.16](#), actual parameter `f1` is passed to formal parameter `p1` in the subroutine `abc`. Because the value of actual parameter `f1` doesn't change after the subroutine call, we used the `USING` addition at the `PERFORM` statement.

In [Listing 7.17](#), the value of actual parameter `f1` will change from `A` to `B`, so we used the `CHANGING` addition at the `PERFORM` statement. Please note that the `USING` or `CHANGING` additions are only to help other developers looking at the code to identify if the actual parameter simply is being used or actually is changed within the subroutine.

```
DATA f1 TYPE c VALUE 'A'.
PERFORM abc CHANGING f1.
FORM abc USING p1.
p1 = 'B'.
ENDFORM.
```

Listing 7.17 Passing Parameters with `CHANGING` Addition

The example in [Listing 7.16](#) will produce the same result and behave the same way even if we use the `CHANGING` addition at the `PERFORM` statement or the `USING` addition as in [Listing 7.17](#).

In the following subsections, we'll introduce the different methods for passing parameters to subroutines.

Parameters Passed by Reference

The syntax at the `FORM` statement determines how the parameters are passed. When the parameters are passed by reference, the formal parameter points to the same memory location as the actual parameter. In other words, a new memory isn't created for the formal parameter; instead, a pointer is created that directs to the memory location of the actual parameter. This way of passing the parameters is efficient (especially in case of internal tables), because no copy of the data is made and no extra memory is used.

When parameters are defined with just the `USING` or `CHANGING` additions at the `FORM` statement, they're passed by reference by default. Some examples include the following:

- ▶ `FORM abc USING p1.`
`ENDFORM.`
- ▶ `FORM abc CHANGING p1.`

```

ENDFORM.
▶ FORM abc USING p1 CHANGING p2.
ENDFORM.

```

When the parameters are passed by reference, the value of the actual parameter changes immediately the moment the contents of the formal parameter are changed, because both point to the same memory location. While programming, you need to be careful, because the original data is instantly changed. If any conditions are not met later in the program logic and you want to quit the subroutine, changes to the original data can't be reverted.

When passing the parameters by reference, `USING` and `CHANGING` are equivalent, but for documentation purposes, you should use `USING` for input parameters (which are not changed in the subroutine) and `CHANGING` for output parameters (which are changed in the subroutine).

If you don't want the value of an actual parameter to be changed automatically, you must pass it by value.

Input Parameters that Pass Values

Parameters are passed by value when the `VALUE` addition is used after the `USING` addition, as shown in [Listing 7.18](#).

```

FORM abc USING VALUE (p1)
                VALUE (p2).
ENDFORM.
FORM xyz USING VALUE (p1) CHANGING p2.
ENDFORM.

```

Listing 7.18 Passing by Value

In [Listing 7.18](#), the parameters `p1` and `p2` are passed by value in subroutine `abc`, whereas in subroutine `xyz` the parameter `p1` is passed by value and `p2` is passed by reference. You use the `USING` addition only once in the syntax and precede each parameter that needs to be passed by value with the `VALUE` addition.

When the parameters are passed by value, the actual parameter and the formal parameter occupy their own memory space. When you call the subroutine, the value of the actual parameter is passed to the formal parameter (it creates a copy of the original data). If the value of the formal parameter changes, this has no effect on the actual parameter. Passing parameters by value is to be avoided,

especially when passing large internal tables. However, passing by value allows you to work with a copy of the original data without worrying about changing the original data, because the value of the actual parameter is never changed. Actual parameters are used for input parameters that the subroutine receives.

Output Parameters that Pass Values

If you want to work with a copy of the output parameters, you can pass them by using the `VALUE` addition with the `CHANGING` statement. This way, a new memory is created for the formal parameter, and the contents of the actual parameter are copied to the formal parameter (similar to the input parameters passed by value).

However, the difference is that the actual parameter will be changed after the subroutine call. The contents of the formal parameter are copied to the actual parameter either once `ENDFORM` is executed or when the subroutine is terminated with the `CHECK` or `EXIT` statements.

The actual parameter remains the same if the subroutine is terminated prematurely with an error message. This allows you to change the actual parameter only when all the conditions are met, unlike pass by reference cases, in which the actual parameter is changed instantly.

The syntax to pass parameters by value and result is shown in [Listing 7.19](#).

```
FORM abc CHANGING VALUE (p1)
                VALUE (p2).
ENDFORM.
FORM xyz USING VALUE (p1)
                CHANGING VALUE (p2).
ENDFORM.
```

Listing 7.19 Passing by Value and Result

In [Listing 7.19](#), the parameters `p1` and `p2` of subroutine `abc` are passed by value and both are output parameters. In subroutine `xyz`, the parameter `p1` is an input parameter that is passed by value and `p2` is an output parameter that is passed by value. You use the `CHANGING` addition only once in the syntax and precede each parameter that needs to be passed with the `VALUE` addition.

Passing Internal Tables to Subroutines

Passing fields to subroutines is simple—but there are a few things to keep in mind when passing internal tables.

Internal tables can be passed a couple of ways. You can pass internal tables with `USING` or `CHANGING` statements. Alternately, you can use the `TABLES` keyword—for example:

```
FORM abc TABLES pt_sflight.
ENDFORM.
```

Using the `TABLES` keyword allows you to pass both internal tables and their header lines (if the actual parameter is an internal table with header line). However, using the `TABLES` keyword is obsolete, and we don't recommend it.

Always pass internal tables with the `USING` or `CHANGING` statements. In addition, always use typed parameters when passing internal tables. The syntax checker will throw an error if any internal table-specific statements, like `LOOP` or `READ`, are used statically on the formal parameter. This is because there is no way for the syntax checker to recognize the formal parameter as an internal table when using untyped formal parameters.

To type the formal parameter, you can refer the formal parameter to a structure or table type. The syntax to type an internal table is similar to that for the `DATA` keyword. When referring to a structure, you can use the syntax `TYPE TABLE OF`, as shown in [Listing 7.20](#).

```
TYPES: BEGIN OF ty_makt,
        matnr type matnr,
      END OF ty_makt.
FORM abc USING pt_makt TYPE STANDARD TABLE OF ty_makt.
ENDFORM.
```

Listing 7.20 Defining Internal Table as Formal Parameter

Table types are the data types for internal tables. Similar to how a structure type defines a structure, a table type defines an internal table. When a data object is referred to a table type using the `TYPE` addition, it creates an internal table. Table types can be defined either locally in ABAP programs or globally in ABAP Data Dictionary. To define a table type locally, use the `OCCURS` clause with the `TYPES` keyword, as shown in [Listing 7.21](#).

```
TYPES: BEGIN OF ty_makt,
        matnr type matnr,
      END OF ty_makt.
TYPES: tbt_y_makt TYPE ty_makt OCCURS 0.
```

```
FORM abc USING pt_makt TYPE tbtty_makt.
ENDFORM.
```

Listing 7.21 Defining Internal Table Using Table Type

Table types are more useful in function modules and methods, where the typing is restricted to `TYPE` or `LIKE` statements and doesn't allow the use of the `TABLE OF` addition.

You can also type the formal parameter by referring to a global internal table via the `LIKE` addition (see [Listing 7.22](#)).

```
TYPES: BEGIN OF ty_makt,
        matnr type matnr,
      END OF ty_makt.
DATA: it_makt TYPE STANDARD TABLE OF ty_makt.
FORM abc USING pt_makt LIKE it_makt.
ENDFORM.
```

Listing 7.22 LIKE Addition

Table types can be defined globally in ABAP Data Dictionary. To create a global table type, proceed as follows:

1. Open Transaction SE11 and select the `DATA TYPE` radio button. Provide the name of the table type and click `CREATE`. The name of the table type should be in a customer namespace, starting with `Y` or `Z` (as shown in [Figure 7.7](#)).

The screenshot shows the 'ABAP Dictionary: Initial Screen' with the following elements:

- Radio buttons for: Database table, View, **Data type** (selected), Type Group, Domain, Search help, Lock object.
- Input fields for each radio button. The 'Data type' field contains 'ztb_makt'.
- Buttons at the bottom: Display, Change, Create.

Figure 7.7 ABAP Data Dictionary: Initial Screen

2. In the modal dialog box that appears, select the TABLE TYPE radio button and click OK (see [Figure 7.8](#)).

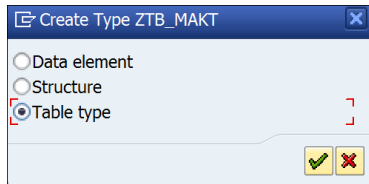


Figure 7.8 Modal Dialog Box to Select Table Type

3. On the next screen, provide short text and enter the structure name or ABAP Data Dictionary table name based on the line type you want the table type to contain, as shown in [Figure 7.9](#). The line type mentioned here will form the components of the table type.

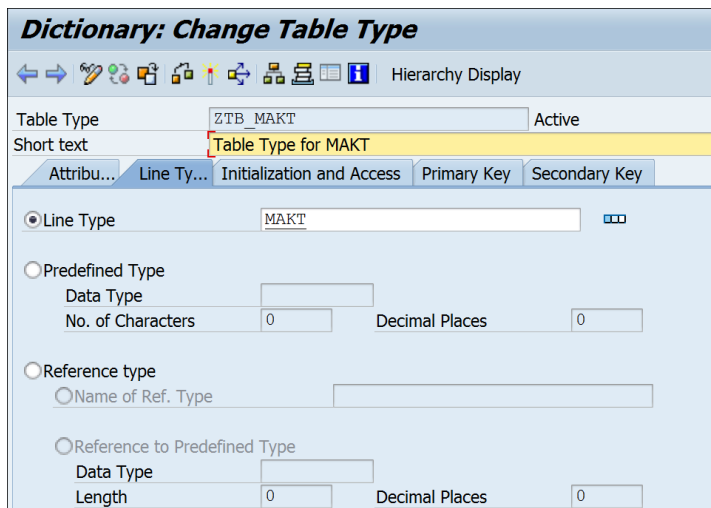


Figure 7.9 Maintaining Line Type for Table Type

4. Click ACTIVATE to activate the table type.

The preceding steps quickly create a table type with default options. However, we can determine more properties for the table type using different tabs on the screen. We'll dive into this topic further in [Chapter 10](#).

7.4.2 Function Modules

Function modules are defined in function groups but can be called from all ABAP programs. Function groups act as containers for function modules that logically belong together. You can create function groups and function modules in ABAP Workbench using the Function Builder tool (Transaction SE37).

Function modules are managed in a central function library, making it easy to look for existing function modules. The SAP system comes with many standard function modules that can be called from any ABAP program. Function modules also support RFC and thus can be called from remote systems using RFC.

One type of function module, the *update module*, is useful when performing database updates. Update modules allow you to extend the scope of the logical unit of work (LUW) from a single dialog step to multiple dialog steps. An LUW is a unit that exists between PAI of one screen and PBO of the next screen. We'll discuss LUWs further in [Chapter 12](#).

Unlike subroutines, function modules are not defined in an ABAP program's source code. Instead, they're defined using the Function Builder. Because function modules are always called externally and provide generic functionality, raising messages in a function module to handle error situations doesn't make sense. If an error occurs, function modules support exception handling, which allows you to catch certain errors while the function module is running.

Function modules are called by many programs. Changing an existing function module will affect all the programs that call that particular function module. If the parameter interface is changed, then all the programs that call the function module should be modified to reflect the new interface. If the function module isn't changed, the programs will throw runtime errors.

To prevent this, the Function Builder has a release process for function modules. By releasing the function module, we can ensure that incompatible changes can't be made to the function module, particularly to the interface. Programs that use a released function module will not cease to work if the function module is changed.

Function modules are created in *function groups* that act as containers for function modules. Function groups can't be executed directly. Rather, they're loaded into memory when any function module of the function group is called. Note that all of the function modules belonging to the function group are loaded into memory

when any one of its function modules is called. For this reason, even though you can define multiple function modules in a function group, we recommend keeping related function modules in the same group.

In this section, we'll discuss the steps for creating and calling function modules.

Creating Function Modules

The following steps walk you through creating a function module:

1. Open Transaction SE37 to access Function Builder.
2. As shown in [Figure 7.10](#), navigate to GOTO • FUNCTION GROUPS • CREATE GROUP to create a new function group.

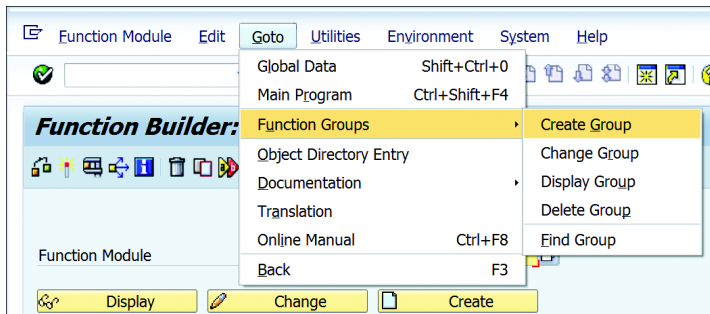


Figure 7.10 Menu Path to Create Function Groups Using Function Builder

3. On the dialog screen that appears, as shown in [Figure 7.11](#), enter a name for your function group (here, we entered "zcb_fg"). The name of a function group can be up to twenty-six characters long. This name is used by the system to create the components of the group (the main program and corresponding include programs).

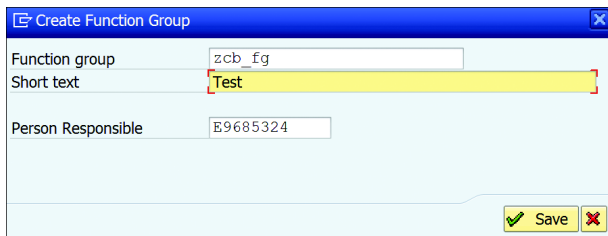


Figure 7.11 Create Function Group

When you create a function group or function module in the Function Builder, the main program and include programs are generated automatically (see [Figure 7.12](#)).

FunctionPool	SAPLZCB_FG	Inactive
1	*****	
2	* System-defined Include-files. *	
3	*****	
4	INCLUDE LZCB_FGTOP.	" Global Data
5	INCLUDE LZCB_FGUXX.	" Function Modules
6		
7	*****	
8	* User-defined Include-files (if necessary). *	
9	*****	
10	* INCLUDE LZCB_FGF...	" Subroutines
11	* INCLUDE LZCB_FGO...	" PBO-Modules
12	* INCLUDE LZCB_FGI...	" PAI-Modules
13	* INCLUDE LZCB_FGE...	" Events
14	* INCLUDE LZCB_FGP...	" Local class implement.

Figure 7.12 Function Group Includes

As shown in [Figure 7.12](#), the main program SAPLZCB_FG contains nothing but the INCLUDE statements for the following include programs:

- ▶ **LZCB_FGTOP**: Contains the FUNCTION-POOL statement and global data declarations for the entire function group.
- ▶ **LZCB_FGUXX**: Contains further INCLUDE statements for the include programs LfgrpU01, LfgrpU02, and so on. These INCLUDE statements contain the actual function modules.
- ▶ **LfgrpF01** and **LfgrpF02**: Subroutines can be maintained in the INCLUDE programs LfgrpF01, LfgrpF02, and so on and can be called from all function modules of the group as internal subroutines.

All the function modules in the function group can access the data declarations made in the global data area of the function group. For this reason, you should place all function modules that use the same data in a single function group instead of creating a new function group for each function module. For example, if you have a set of function modules that all use the same internal table, you could place them in a function group containing the table definition in its global data.

Function groups, like executable programs and module pools, can contain screens. Depending upon the type of screen, user inputs can be processed either in dialog modules or in the corresponding event blocks in the main program of the function group.

Data declarations made within the function module between the `FUNCTION...ENDFUNCTION` statements are local to the specific function module. These data declarations can be accessed only within the function module in which they are defined.

4. Once the function group is created, you can create the function module within this function group. To do so, enter the name of the `FUNCTION MODULE` on the initial screen of Function Builder and click **CREATE** (Figure 7.13).

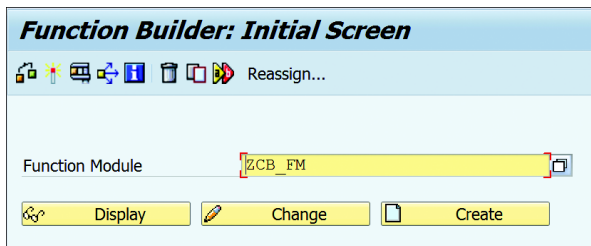


Figure 7.13 Creating Function Module

5. This will open a dialog box in which you'll input the name of the function group and a short description (Figure 7.14). Then, click **SAVE**.

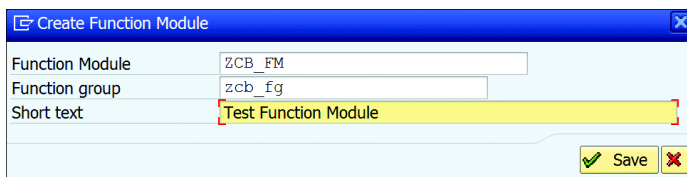


Figure 7.14 Assigning Function Module to Function Group

6. This will open the function module editor, as shown in Figure 7.15.

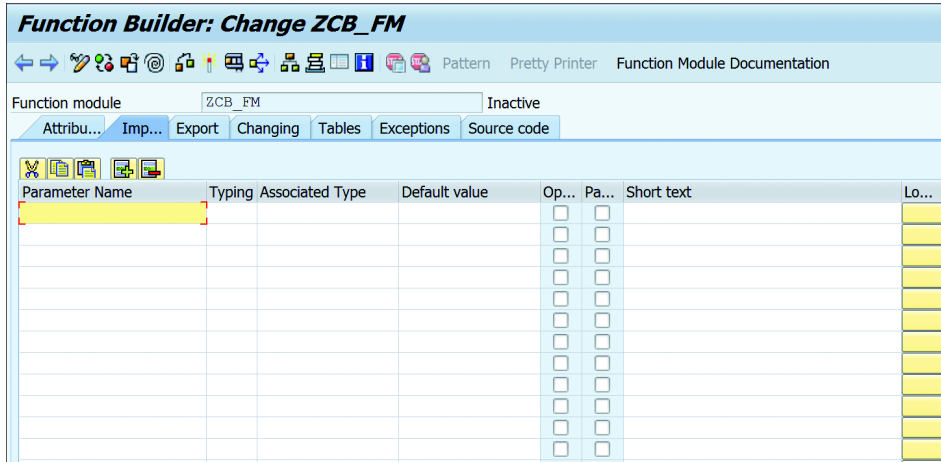


Figure 7.15 Function Module Editor

The editor contains seven tabs:

- ▶ **ATTRIBUTES:** The most important setting in the ATTRIBUTES tab is PROCESSING TYPE. Here, you can choose to create a NORMAL FUNCTION MODULE, a REMOTE-ENABLED MODULE, or an UPDATE MODULE. By default, the NORMAL FUNCTION MODULE is selected (as shown in [Figure 7.16](#)).

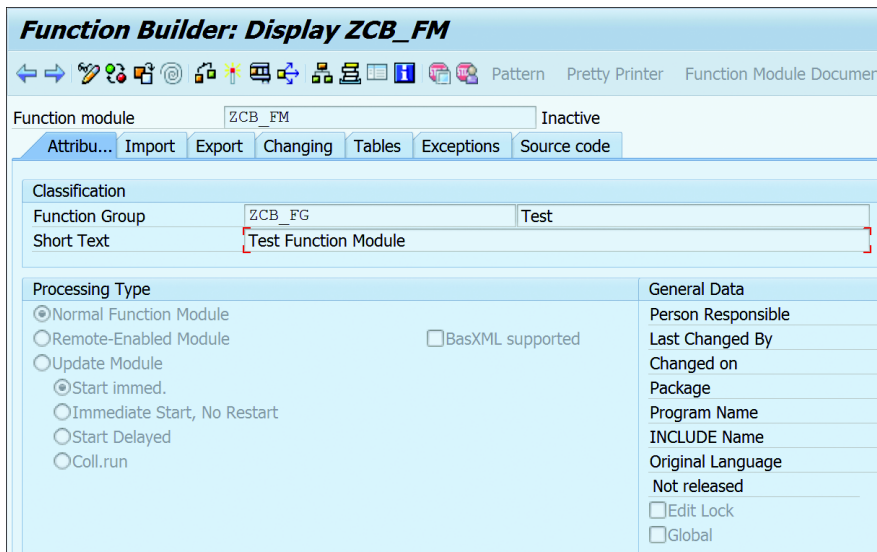


Figure 7.16 Attributes Tab

Normal function modules can be called from any ABAP program in the SAP system. Remote-enabled function modules can be called from a remote system. These types of function modules help SAP system to SAP system interactions, or interactions between SAP systems and remote systems (through remote communications). Update modules are used during updating. We'll look at update techniques in [Chapter 12](#).

- ▶ **IMPORT:** In this tab, you define all the importing parameters that will be imported from the calling program. Import parameters can't be changed in the function module. As shown in [Figure 7.17](#), in the PARAMETER NAME column, you enter the name of your importing parameter. In the second column, TYPING, you provide the type specification (either TYPE or TYPE REF TO). In the third column, ASSOCIATED TYPE, you provide the reference to a data type defined in ABAP Data Dictionary.

In the DEFAULT VALUE column, you can assign a default value to the parameter that's automatically filled when the function module is called in any ABAP program. The OPTIONAL column has a checkbox that can be ticked to make the parameter optional. Optional parameters can be ignored by the calling program. The PASS VALUE column checkbox allows you to pass the parameter by value. Otherwise, is the parameter's passed by reference. You can maintain SHORT TEXT and LONG TEXT in the corresponding columns to describe the parameter for other developers, as shown in [Figure 7.17](#).

Parameter Name	Typing	Associated Type	Default value	Optional	Pass Value	Short text	Long Text
IM_MATNR	TYPE	MATNR		<input type="checkbox"/>	<input type="checkbox"/>	Material Number	Create
IM_SPRAS	TYPE	SPRAS		<input type="checkbox"/>	<input type="checkbox"/>	Language Key	Create
				<input type="checkbox"/>	<input type="checkbox"/>		

Figure 7.17 Import Tab

- ▶ **EXPORT:** You define all the parameters that will be exported to the calling program under the EXPORT tab. Except for the missing DEFAULT VALUE and OPTIONAL columns, as shown in the IMPORT tab, all other columns are similar to the IMPORT tab ([Figure 7.18](#)). These parameters are filled in the function

module and sent to the calling program. The program can't send any values to the function module using these parameters.

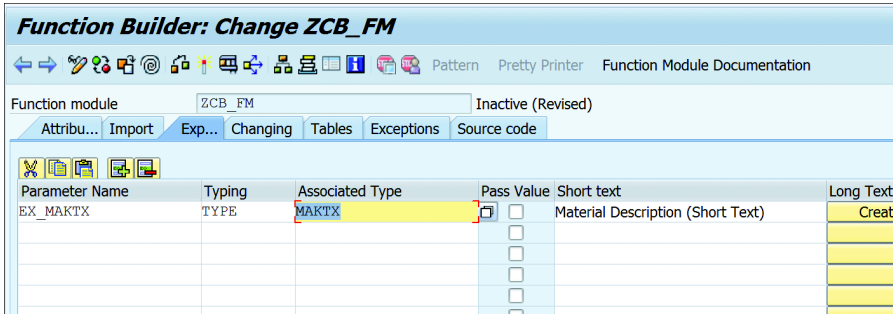


Figure 7.18 Export Tab

- ▶ **CHANGING:** In this tab, you define **CHANGING** parameters. **CHANGING** parameters work as both importing and exporting parameters (i.e., the function module can pass values to and import values using these parameters). The columns are identical to those in the **IMPORT** tab.
- ▶ **TABLES:** This tab is used to define internal tables. However, it's considered obsolete and it's recommended to define internal tables in the **IMPORT**, **EXPORT**, or **CHANGING** tabs per requirements. You refer the parameter to a structure or database table using the **LIKE** specification to define an internal table in the **TABLES** tab. This creates an internal table with a header line. To define an internal table in the **IMPORT**, **EXPORT**, or **CHANGING** tabs, use the **TYPE** specification and refer the parameter to a table type.
- ▶ **EXCEPTIONS:** In this tab, you can maintain exceptions that can be raised in the function modules. When an exception is raised, the function module is exited without executing further code. At this point, the control will return to the calling program and the system field `sy-subrc` will be set to the exception number assigned while calling the function module. The value of the `sy-subrc` field can be checked in the calling program to implement suitable error handling, as shown ahead at line numbers 26 to 31 in [Figure 7.22](#).

You raise exceptions using the **RAISE** keyword in the source code of the function module, as shown ahead at line number 15 in [Figure 7.20](#). To maintain the exception, enter the name of the exception in the **EXCEPTION** column and **SHORT** and **LONG TEXT** in the respective columns to provide additional information ([Figure 7.19](#)).

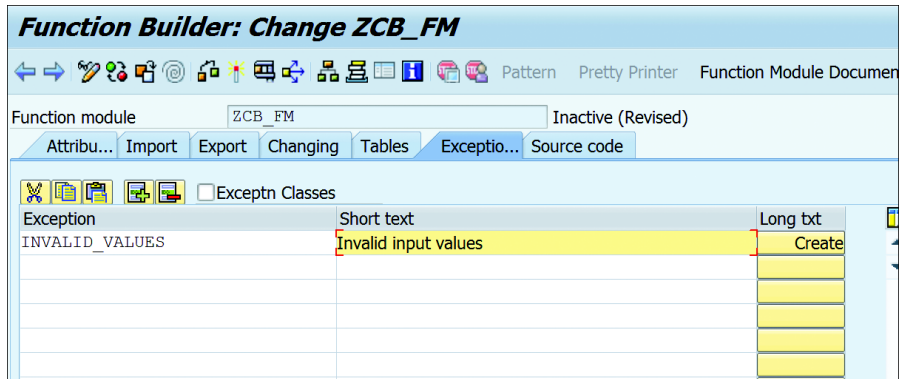


Figure 7.19 Exceptions Tab

- ▶ **SOURCE CODE:** In this tab, you write the ABAP code for the function module within the `FUNCTION...ENDFUNCTION` block. Figure 7.20 shows code in which a function module is importing the material number and language through the `im_matnr` and `im_spras` parameters, respectively, and exporting the material description in the `ex_maktx` export parameter. If the `SELECT` statement fails, an exception is raised.

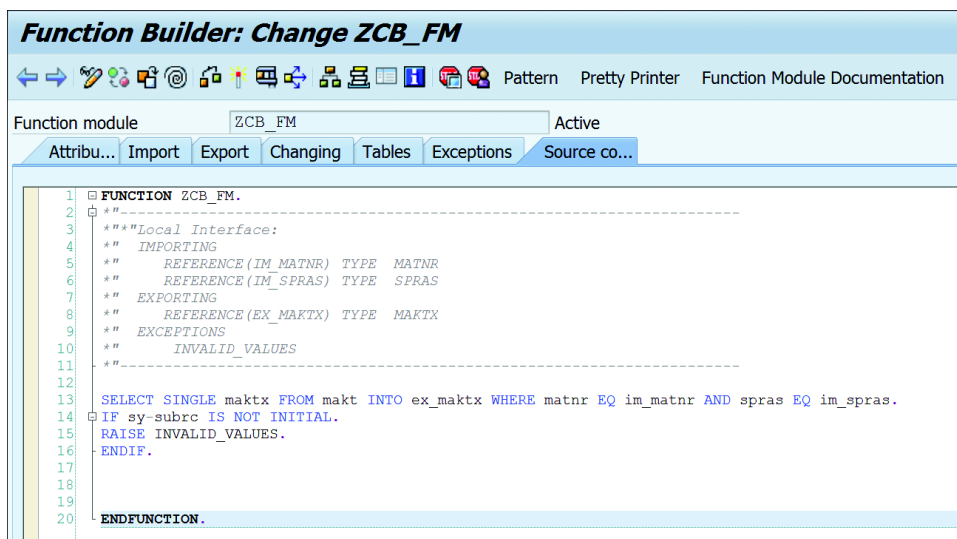


Figure 7.20 Source Code Tab

Once the source code is maintained, activate the function module by clicking the **ACTIVATE** button. Make sure to select all inactive objects when the system prompts you to do so; the function group we first created wasn't activated separately.

Calling Function Modules

Function modules are called using the `CALL FUNCTION` statement in any ABAP program. To call a function module, you need to fill the parameter interface in the calling program. You can use the **PATTERN** button in the application toolbar of the ABAP Editor to make this task easy (as shown in [Figure 7.21](#)).

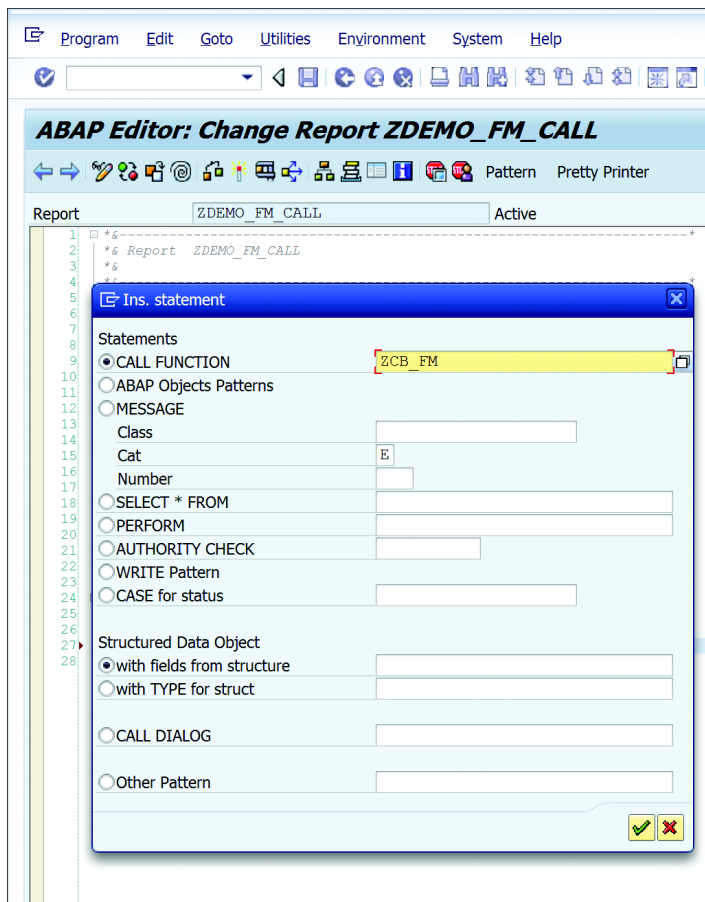


Figure 7.21 Using Pattern Button to Call Function Module

In the code generated by the PATTERN button, simply map your actual parameters to the corresponding formal parameters, as shown in [Figure 7.22](#). Always ensure that you've defined your actual parameters as the same type as the formal parameters. Passing incorrect or incompatible parameters results in runtime errors when the function module is called.

```

1  *%-----
2  *% Report  ZDEMO_FM_CALL
3  *%
4  *%
5  *%
6  *%
7  *%-----
8
9  REPORT  ZDEMO_FM_CALL.
10 DATA v_maktx TYPE maktx.
11 PARAMETERS : p_matnr TYPE matnr,
12              p_spras TYPE spras.
13
14 CALL FUNCTION 'ZCB_FM'
15   EXPORTING
16     im_matnr      = p_matnr
17     im_spras      = p_spras
18   IMPORTING
19     EX_MAKTX      = v_maktx
20   EXCEPTIONS
21     INVALID_VALUES = 1
22     OTHERS         = 2
23
24 IF sy-subrc <> 0.
25   * Implement suitable error handling here
26   CASE sy-subrc.
27     WHEN 1.
28       MESSAGE 'No records found' TYPE 'E'.
29     WHEN 2.
30       "Do nothing
31   ENDCASE.
32 ELSE.
33   WRITE v_maktx.
34 ENDF.

```

Figure 7.22 Function Module Call

7.4.3 Methods

Methods have been a part of procedures since ABAP Objects was first introduced. In this section, you'll learn how to work with methods and how to call methods into your ABAP programs. However, we'll have an in-depth discussion of object-oriented programming concepts and explore methods further in the next chapter.

Creating a Method

Object-oriented programming takes a different approach than procedural programming. In procedural programming, we create procedures that take some

input, process it, and export the results. In object-oriented programming, we deal with the objects we want to manipulate rather than the logic required to manipulate them. This approach helps developers identify the objects that require manipulation and establish a relationship between them. This process is called *data modeling*.

Once an object is identified, a class is defined representing the object. Methods are then defined as tasks that can be performed on the object. Methods are the procedures in object-oriented programming, and variables are the attributes.

A *class* is like a generic representation of an object. In other words, an object exists as an instance of a class. For example, an employee class can be defined with methods and attributes that perform various actions, like leaves, personal info, or compensation. In addition, an object named James can be defined as an instance of the class employee.

You can define classes either globally in the repository or locally in an ABAP program. Global classes are created in class pool types of programs using the Class Builder (Transaction SE24).

Like function groups, you can't execute class pools directly. Each class pool contains the definition of one class. Objects (runtime instances) are created using the `CREATE OBJECT` statement in ABAP programs. These objects execute the statements of the class pool.

In this section, you'll learn how to create a global class and how to define methods within it. Like the function module example, we'll create a method that imports a material number and language and exports a description.

Creating a Global Class

To create a global class and define the methods within, follow these steps:

1. Open the Class Builder tool using Transaction SE24.
2. In the `OBJECT TYPE` field, enter the name of the class you want to create and click the `CREATE` button. The name of the class should be in a customer namespace, starting with a Y or Z. We recommend starting the class name with the naming convention `ZCL_`; we'll discuss why shortly. [Figure 7.23](#) shows the initial screen of the Class Builder.

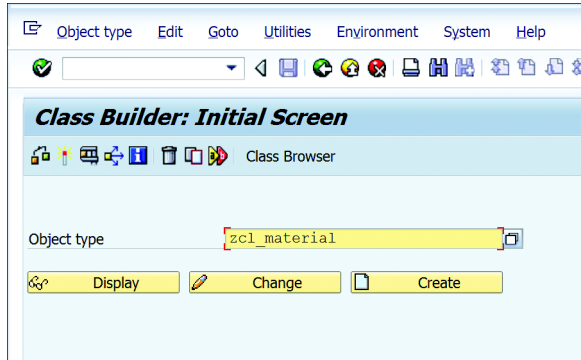


Figure 7.23 Class Builder: Initial Screen

Class Builder is used to create the class pool *and* the interface pool programs. If the first three letters of your object name are "zcl", then Class Builder will create a class directly. Similarly, if the first three letters are "zif", then it will create an interface pool. However, if you start your object name with any other letters, the builder will prompt you to select whether you want to create a class pool or interface pool.

Interface pools don't contain any executable statements. Instead, they function as containers for interface definitions. When you implement an interface in a class, the interface definition is implicitly included in the class definition.

3. In the dialog box that appears, maintain a short description for your class, and leave other options set to their default values; we'll explore these other options in the next chapter. Click **SAVE**, as shown in [Figure 7.24](#).

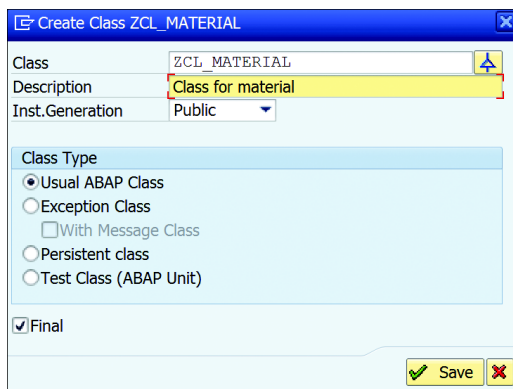


Figure 7.24 Maintain Class Properties

4. This will open the form-based editor, with the `METHODS` tab selected by default, as shown in [Figure 7.25](#). The editor has eight tabs that allow you to maintain different properties. For now, use the `METHODS` tab to create a method; we'll explore the other tabs in the next chapter once we discuss the basics of object-oriented programming.

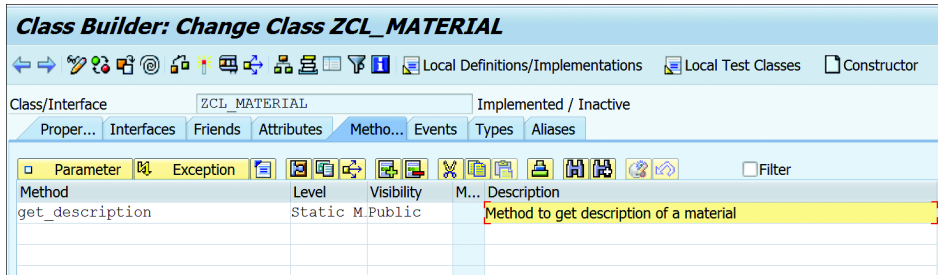


Figure 7.25 Form-Based Editor

The table control under the `METHODS` tab has five columns:

- ▶ **METHOD:** This is where you create the name of your method.
- ▶ **LEVEL:** Here, you maintain whether the method is a static or instance method. Static methods are similar to function modules in that one instance exists in the program when the method is called, whereas instance methods can have multiple instances in the same program. For now, create a static method. You can read more about instance methods in the next chapter. Use the `[F4]` help to select the level.
- ▶ **VISIBILITY:** In a class, attributes and methods can be set to a visibility level to encapsulate the implementation from the outside world. There are three visibility levels:
 - **PUBLIC:** Allows the attributes and methods to be accessed directly from any ABAP program
 - **PROTECTED:** Allows the attributes and methods to be accessed only from the methods of the same class or its subclasses
 - **PRIVATE:** Allows the attributes and methods to be accessed only from the methods of the same class

Setting the right visibility level at the design stage to encapsulate the implementation is a great way to reduce bugs in your applications. For now, set

the visibility of the method to PUBLIC because you'll be calling it in the program directly. We'll discuss encapsulation and implementation hiding in the next chapter.

- ▶ DESCRIPTION: Maintain a short description for your method so other programmers can understand its purpose.
5. Click SAVE in the standard toolbar to save your changes, and then click the PARAMETER button above the METHOD column (as shown in [Figure 7.26](#)) to maintain the parameter interface for the method. Make sure the cursor is on the method name when you click the PARAMETER button.

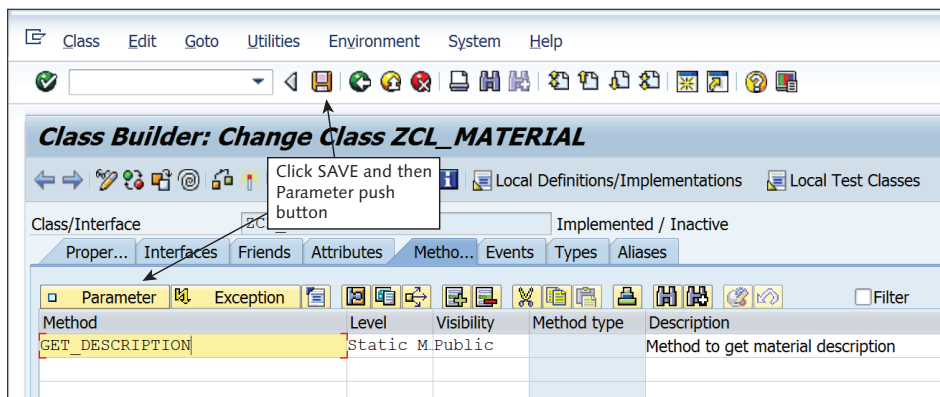
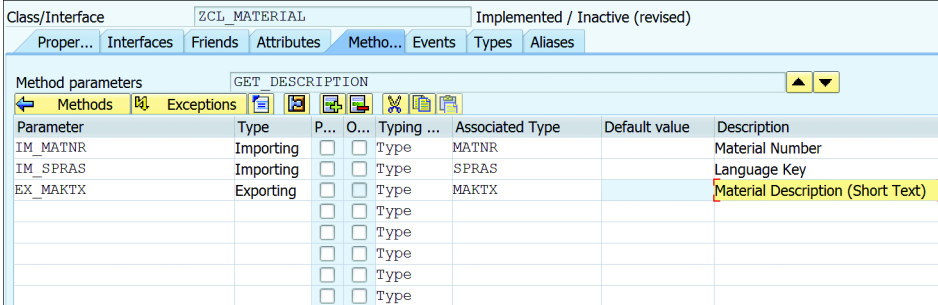


Figure 7.26 Save and Maintain Parameter Interface

6. On the parameter interface screen, as shown in [Figure 7.27](#), you can maintain importing, exporting, changing, and returning parameters. Importing, exporting, and changing parameters work like the parameters in a function module. Returning parameters are useful for special methods called functional methods, which just return one value, like validation logic in which the method just returns a Boolean value: true or false. When you use returning parameters, you can't have any exporting parameters for the method.
7. As shown in [Figure 7.27](#), enter the name for your parameter and use the **F4** help to maintain the parameter type (either IMPORTING or EXPORTING in this example).
8. In the ASSOCIATED TYPE column, enter the data element reference for each parameter, as you did for function modules.

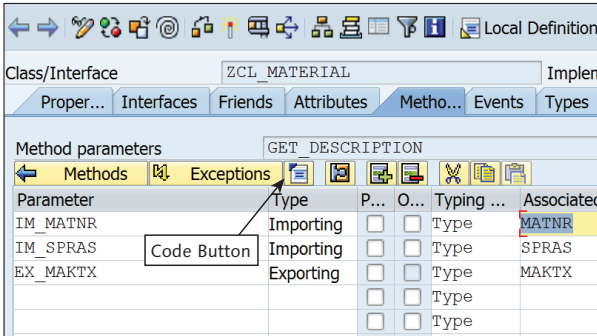
9. Maintain the description for the parameter in the DESCRIPTION column, or simply press `[Enter]` to pick the description associated with the data element from ABAP Data Dictionary.



Parameter	Type	P...	O...	Typing ...	Associated Type	Default value	Description
IM_MATNR	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	MATNR		Material Number
IM_SPRAS	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	SPRAS		Language Key
EX_MAKTX	Exporting	<input type="checkbox"/>	<input type="checkbox"/>	Type	MAKTX		Material Description (Short Text)
		<input type="checkbox"/>	<input type="checkbox"/>	Type			
		<input type="checkbox"/>	<input type="checkbox"/>	Type			
		<input type="checkbox"/>	<input type="checkbox"/>	Type			
		<input type="checkbox"/>	<input type="checkbox"/>	Type			

Figure 7.27 Maintain Parameter Interface

10. Click the **ACTIVATE** button in the application toolbar to activate your class and the method.
11. Once activated, click the **CODE** button, as shown in [Figure 7.28](#).



Parameter	Type	P...	O...	Typing ...	Associated
IM_MATNR	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	MATNR
IM_SPRAS	Importing	<input type="checkbox"/>	<input type="checkbox"/>	Type	SPRAS
EX_MAKTX	Exporting	<input type="checkbox"/>	<input type="checkbox"/>	Type	MAKTX
		<input type="checkbox"/>	<input type="checkbox"/>	Type	

Figure 7.28 Click Code Button to Maintain Method Code

12. This will open the editor, in which you can write the code between `METHOD` and `ENDMETHOD`, as shown in [Figure 7.29](#). You can also click the **SIGNATURE** button to see the parameter interface. Click the **ACTIVATE** button to save and activate your changes.
13. Click the **BACK** button twice to go back to the main screen.

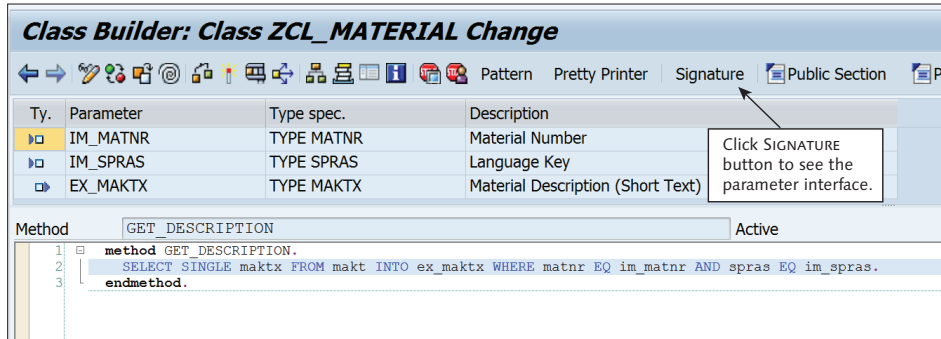


Figure 7.29 Writing Code in Method

Now that the method is implemented and maintained as a public method, it can be called from any program.

Calling a Method

Calling a static method is similar to calling a function module; it can be called directly using the `CALL METHOD` statement. We can use the `PATTERN` button in ABAP Editor to insert the statement, but to call an instance method, we first need to create a reference object, instantiate it, and then access the method using the reference object. We'll discuss instance methods in the next chapter.

Because we've created a static method in this example, use the `PATTERN` button in the ABAP Editor, as shown in [Figure 7.30](#).

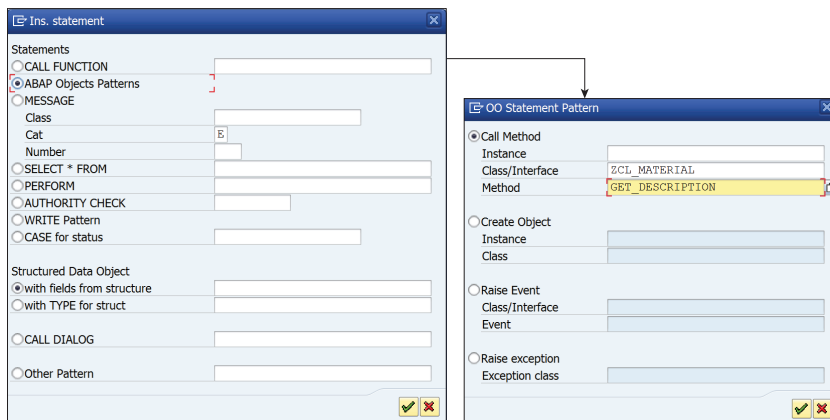


Figure 7.30 Calling Static Method

Select the second radio button, ABAP OBJECTS PATTERNS, and then the CALL METHOD radio button on next screen. Enter the class name and method name and click OK to insert the statement.

The syntax used to call a static method is `CALL METHOD class_name=>method_name`, as shown in [Figure 7.31](#).

```

1  *~
2  *~ Report ZDEMO_METHOD_CALL
3  *~
4  *~
5  *~
6  *~
7  *~
8  ;
9  REPORT ZDEMO_METHOD_CALL.
10 ;
11 DATA v_maktx TYPE maktx.
12 ;
13 PARAMETERS : p_matnr TYPE matnr,
14              p_spras TYPE spras.
15 ;
16 CALL METHOD ZCL_MATERIAL=>GET_DESCRIPTION
17 EXPORTING
18   IM_MATNR = p_matnr
19   IM_SPRAS = p_spras
20 IMPORTING
21   EX_MAKTX = v_maktx
22 .
23 ;
24 WRITE v_maktx.

```

Figure 7.31 Calling Method in ABAP Program

7.5 Inline Declarations

In SAP NetWeaver 7.4, SAP introduced inline declarations. *Inline declarations* allow you to declare data objects inline when required, as opposed to declaring them prior to their usage in an ABAP statement.

Most ABAP developers are taught to define data objects at the beginning of a program (for global data) or at the beginning of a procedure (for local data) and avoid using data declarations randomly throughout the code. However, many developers ignore following these rules, and declare the data object just before it is used in an ABAP statement, as it provides more clarity on where the data object is used. Many other programming languages, for example PHP, don't require you to declare a data object before its use. With the introduction of inline declarations,

you now have the flexibility to declare data objects when you need them as opposed to declaring them at the top of the program or a procedure.

Even though it is possible to use inline declarations with global scope (global data), it is advised not to use inline declarations for global data objects in order to avoid confusion. SAP recommends using inline declarations only within procedures to keep the scope of the data object local.

Inline declarations are very helpful when you have a long procedure with hundreds of lines of code, as it gets annoying to scroll up to the beginning of the procedure each time you want to define a new data object. Inline declarations are made using the keyword `DATA(. .)` within the ABAP statement's operand position where the data object is expected in a writer position (that is, where the data is written to a variable).

To get a better idea of how inline declarations have impacted ABAP coding, let's look at how to use inline declarations with examples comparing the old syntax. In the following subsections, we discuss some examples of when it's best to use inline declarations.

7.5.1 Assigning Values to Data Objects

In a pre-SAP NetWeaver 7.4 system, to assign a value to a data object we first define the data object and then assign the value as shown:

```
DATA lv_book TYPE string.  
lv_book = 'Complete ABAP'.
```

With SAP NetWeaver 7.4 and later, you can write the previous two lines of code as a single statement:

```
DATA(lv_book) = 'Complete ABAP'.
```

As you can see, inline declarations reduce the number of lines by half. When we use the statement `DATA(. .)`, the data object is automatically created when the code is compiled, irrespective of whether the statement is executed during runtime.

7.5.2 Using Inline Declarations with Table Work Areas

You can use inline declarations with table work areas as well. [Listing 7.23](#) shows the pre-SAP NetWeaver 7.4 table work area.

```
DATA: lt_spfli TYPE STANDARD TABLE OF SPFLI,
      lw_spfli LIKE LINE OF lt_spfli.
LOOP AT lt_spfli INTO lw_spfli.
ENDLOOP.
```

Listing 7.23 Pre-SAP NetWeaver Table Work Area

Listing 7.24 shows the transformed SAP NetWeaver 7.4 or higher change with the inline declaration.

```
DATA: lt_spfli TYPE STANDARD TABLE OF SPFLI.
LOOP AT lt_spfli INTO DATA(lw_spfli).
ENDLOOP.
```

Listing 7.24 SAP NetWeaver 7.4 and Higher Table Work Area

You can use inline declaration for work area with the READ statement as well:

```
READ TABLE lt_spfli INTO DATA(lw_spfli) INDEX 1.
```

7.5.3 Avoiding Helper Variables

Inline declarations are also useful when you want to avoid defining helper variables beforehand. For example, to find the number of rows in an internal table, we use the DESCRIBE TABLE statement in conjunction with a helper variable. A pre-SAP NetWeaver 7.4 approach would be:

```
DATA lv_lines TYPE i. "helper variable
DESCRIBE TABLE lt_spfli LINES lv_lines.
```

With inline declarations, we can avoid declaring the helper variable `lv_lines` in the above statement, as shown:

```
DESCRIBE TABLE lt_spfli LINES DATA(lv_lines).
```

7.5.4 Declaring Actual Parameters

Another area where inline declarations are very useful is with the declaration of actual parameters. For example, when calling the method of a class, the actual parameters should match the *typing* of the formal parameters. To ensure the actual parameters are of the same type as the formal parameter, we check the signature of the method and define the actual parameters using the same typing as the formal parameters.

Navigating to check the signature of the method for declaring each actual parameter is annoying. In addition, each time the typing for the formal parameter is changed, the typing for the actual parameter in the program should also be changed. Inline declarations help you to avoid all of these problems as shown below.

Listing 7.25 shows how actual parameters were used before SAP NetWeaver 7.4.

```
DATA: lv_a1 TYPE string,
      lv_a2 TYPE string.
oref->meth( IMPORTING fp1 = lv_a1
            fp2 = lv_a1 ).
```

Listing 7.25 Actual Parameters Pre-SAP NetWeaver 7.4

Listing 7.26 shows how we should use actual parameters after SAP NetWeaver 7.4.

```
oref->meth( IMPORTING fp1 = DATA(lv_a1)
            fp2 = DATA(lv_a1) ).
```

Listing 7.26 Post-SAP NetWeaver 7.4 Actual Parameter Usage

As you can see, inline declarations can be used in any statement in the writer position. If used correctly, inline declarations can make your code leaner and easier to understand. As previously mentioned, use inline declarations only within procedures to limit their scope to the local data of the procedure.

7.6 Summary

In this chapter, we discussed various modularization techniques that can be used in ABAP programs. We began this chapter with an overview of key concepts and elements. You then learned about processing blocks and how they're called and implemented. You saw that event blocks are implemented by event keywords and called by the ABAP runtime environment.

Procedures are implemented by their own keywords and are called by specific keywords in the code. With this understanding of processing blocks, we're ready to discuss their usage when we cover some of the advanced topics in ABAP in upcoming chapters.

In the last section of this chapter, we looked how inline declarations have simplified coding for ABAP developers. We did this by comparing a pre-SAP NetWeaver 7.4 line of code to an inline declaration-oriented code, to highlight the significance of the change.

In the next chapter, we'll explore concepts related to object-oriented programming and you'll learn more about classes and methods.

Some elements of ABAP programs can be designed using procedural or object-oriented programming. This chapter explores the concepts of object-oriented programming and their implementation in ABAP.

8 Object-Oriented ABAP

ABAP is a hybrid programming language that supports both procedural and object-oriented programming (OOP) techniques. In this chapter, we'll discuss the various concepts used in OOP and the advantages it provides over procedural programming techniques.

We'll start with a basic overview of OOP in [Section 8.1](#). This introduction should help you appreciate the advantages of using OOP techniques. To understand OOP, you'll need to understand concepts such as encapsulation, inheritance, polymorphism, data encapsulation, and information hiding. We'll start examining those concepts in [Section 8.2](#) with a look at encapsulation and the techniques to hide implementation from the outside world.

In [Section 8.3](#), we'll discuss inheritance and the techniques that allow you to leverage existing features and functionality without having to reinvent the wheel. In [Section 8.4](#), we'll look at polymorphism, which allows the same object to behave differently at runtime. We'll conclude the chapter with a discussion of XML in [Section 8.5](#).

8.1 Introduction to Object-Oriented Programming

Before OOP, everything was based on functions and variables without focusing on the object itself. With OOP, the focus is on *objects*, which represent real-life entities, with functions and variables approximating the objects.

Traditional programming focused on developing programming logic to manipulate data—for example, a logical procedure that takes input data, processes it, and

produces output. However, OOP focuses on the object being manipulated rather than the logic to manipulate the data.

If you look around, you'll see many real-world objects, such as your car, your dog, your laptop, and so on. Each of these real-world objects is represented as a software object in OOP. Real-world objects have two characteristics: states and behaviors. For example, a car has states such as its current gear, current speed, and so on, and behaviors such as changing gears, applying brakes, and so on. Software objects that represent real-world objects also have states and behaviors; software objects store their states in variables (called attributes) and contain functions (called methods) to manipulate or expose the states of the object. For example, a car object can have attributes to store the current speed, current gear, and so on, and methods to read the current speed or change the gear.

Methods operate on the internal state of the object; that is, they can access the attributes of their own object and serve as a mechanism for communication between objects. Hiding this internal state of the object and routing all access to the object through its methods is known as *encapsulation*. Encapsulation is a fundamental principle of OOP, which we'll discuss in [Section 8.2](#).

Sometimes, an object may share many similarities with other objects while having specific features of its own. For example, a mountain bike may have all the features of a road bike plus certain features unique only to mountain bikes. The concept of inheritance helps us leverage existing code and avoid code redundancy while extending functionality. We'll discuss inheritance in [Section 8.3](#).

Identifying the states and behaviors of real-world objects is the first step to start thinking in terms of OOP. Before exploring OOP further, let's consider a simple procedural programming application that calculates a person's body mass index (BMI). We'll then look at how this application can be designed using OOP. For this example, we can create a function module with three interface parameters: height, weight, and BMI. We can pass the person's height and weight to the function and receive the calculated BMI as a result, as shown in [Listing 8.1](#).

```
FUNCTION ZCALCULATE_BMI.
*"-----
**"Local Interface:
*"  IMPORTING
*"    REFERENCE(HEIGHT) TYPE I
*"    REFERENCE(WEIGHT) TYPE I
*"  EXPORTING
```

```
*"      REFERENCE(BMI) TYPE I
BMI = WEIGHT / HEIGHT.
ENDFUNCTION.
```

Listing 8.1 Example Function Module to Calculate BMI

Listing 8.2 shows the call to the function module `zcalculate_bmi` that we defined in Listing 8.1.

```
REPORT ZCA_BMI.
DATA : v_bmi TYPE I.
PARAMETERS: p_height TYPE I,
            P_weight TYPE I.

CALL FUNCTION 'ZCALCULATE_BMI'
EXPORTING
    height = p_height
    weight = p_weight
IMPORTING
    bmi = v_bmi.
WRITE: 'Your BMI is', v_bmi.
```

Listing 8.2 Program to Calculate BMI

When the function module is called in the program, the whole function group to which this function module belongs is loaded into the memory (internal session) the main program is assigned to. Each subsequent call to this function module in the program will access the existing instance in the memory as opposed to loading a new instance.

When a function module is called in a program, one instance of it is created in the internal session in which the program runs, and each subsequent call to this function module from the program will point to the existing instance that was previously loaded into the internal session. You can't have multiple instances of the same function module within the same internal session.

If you call the `zcalculate_bmi` function module in the program to calculate the BMI of multiple persons, there's no way for the function module to be aware of the person for which the BMI is calculated. You can define a global variable in the function group to store a person's name so that other function modules of the function group can use this information, but it's easy to lose track of all these variables in the code—and even then, this design can only support one person at a time in the application.

The focus of the function module here is to take a certain input and process it to produce a result. OOP shifts the focus to the object. In this example, we define an object that represents a person and then define attributes and methods to store and process the person's BMI.

In this section, we'll look at the defining elements of OOP, including classes, methods, instances and static components, and events.

8.1.1 Classes

Objects encapsulate data and create functions to manipulate that data. The characteristics (called *attributes*) and functions (called *methods*) of an object are put into one package. The definition of this package is called a *class*. To use objects in a program, classes must be defined, because an object exists as an instance of a class.

A class can be defined as shown in [Listing 8.3](#).

```
REPORT ZCA_BMI.
CLASS CL_PERSON DEFINITION.
PUBLIC SECTION.

TYPES ty_packed TYPE p LENGTH 4 DECIMALS 2.
METHODS set_height IMPORTING im_height TYPE ty_packed.
METHODS set_weight IMPORTING im_weight TYPE ty_packed.
METHODS set_bmi.
METHODS get_bmi RETURNING VALUE(r_bmi) TYPE ty_packed.
PRIVATE SECTION.
DATA : bmi TYPE I,
      height TYPE I,
      weight TYPE I.
ENDCLASS.
CLASS CL_PERSON IMPLEMENTATION.
METHOD set_height.
    height = im_height.
ENDMETHOD.
METHOD set_weight.
    weight = im_weight.
ENDMETHOD.
METHOD set_bmi.
    bmi = height / weight.
ENDMETHOD.
METHOD get_bmi.
    r_bmi = bmi.
ENDMETHOD.
ENDCLASS.
```

```

DATA john TYPE REF TO cl_person.
DATA mark TYPE REF TO cl_person.
START-OF-SELECTION.
CREATE OBJECT john.

john->set_height( exporting im_height = 165 ).
john->set_weight( exporting im_weight = 50 ).
john->set_bmi( ).
WRITE : / 'John's BMI is', john->get_bmi( ).

CREATE OBJECT mark.

mark->set_height( exporting im_height = 175 ).
mark->set_weight( exporting im_weight = 80 ).
mark->set_bmi( ).
WRITE : / 'Mark's BMI is', mark->get_bmi( ).

IF john->get_bmi( ) GT mark->get_bmi( ).
WRITE : / 'John is fatter than Mark'.
ELSE.
WRITE : / 'Mark is fatter than John'.
ENDIF.

```

Listing 8.3 Using Objects

A class consists of two sections: a definition section and an implementation section. In the *definition section*, we define the *components* of the class, such as attributes, methods, and events. The *attributes* of a class store the state of an object, and the *methods* implement the logic to manipulate the behavior of an object by accessing the attributes.

Different *events* can be defined for the class, which can be raised during the runtime of the object. Methods can be called directly or can respond to events defined in the class. In the *implementation section*, we implement the methods to manipulate the behavior of an object or to respond to the events of the class.

Each class component can belong to a specific visibility section, such as *public*, *protected*, or *private*. The visibility section enables data encapsulation by putting restrictions on how the class component can be accessed. For example, a component belonging to the public visibility section can be accessed by external objects and programs without any restrictions, whereas a component belonging to the private visibility section can be accessed only by the methods of the same class.

Components belonging to the protected section can be accessed by the methods of the same class and its subclasses. A *subclass* inherits its components from a parent class. Once a class inherits from a parent class, it can access the public and protected components of the parent class. We'll learn more about encapsulation in [Section 8.2](#).

In [Listing 8.3](#), we defined the `CL_PERSON` class consisting of `HEIGHT`, `WEIGHT` and `BMI` attributes in the private visibility section. Because these attributes are private, they can't be accessed by external objects or programs directly. To manipulate these attributes, we defined setter and getter methods in the public section of the class. *Setter methods* allow you to set the value of an attribute, and *getter methods* allow you to read the value of an attribute. Defining the attributes in the private section and providing setter and getter methods to manipulate the attributes allows for more control, such as validating the data before changing the state of the object.

For example, we can avoid an invalid value being set for a person's height by validating the input in the `SET_HEIGHT` method. In the program, we defined two *object reference variables* called `JOHN` and `MARK` referring to the `CL_PERSON` class. An object exists as an instance of a class, and each class can have multiple instances. The object reference variable initially contains an empty value until the object is instantiated using the `CREATE OBJECT` statement. Once the object is instantiated, the object reference variable contains the reference (pointer) to the object in memory, and the object is accessed using the object reference variable. The object reference variables are also called *reference objects* and are defined using the `TYPE REF TO` addition.

Unlike in function modules, here each reference object occupies its own memory space and multiple instances of the class can be defined in the program, with each instance working with its own data.

In this section, we'll discuss local and global classes, class parts and components, and the use of visibility in classes.

Local and Global Classes

As previously discussed, classes can be defined locally in an ABAP program or globally in the system using Class Builder (Transaction SE24). *Local classes* can be used only within the program in which they're defined, whereas *global classes* can

be used in any ABAP program. Irrespective of whether the class is defined locally or globally, the syntax to define the class remains the same, as shown in [Listing 8.3](#). Apart from visibility, there is no difference between a local class and a global class. When a class is used in an ABAP program, the system first searches if the class exists locally. If the class isn't defined locally, then the system looks for a global class.

When defining global classes, we can use the form-based editor of the Class Builder tool to easily define the components of the class and implement the methods without getting into the nitty gritty of the syntax.

Wherever applicable, we will show you the steps to define classes both locally and globally throughout this chapter. Define local classes if the scope of the object is limited to the program and the object may be irrelevant to other programs, but if you plan to use an object in multiple programs, we recommend defining the class globally in Class Builder.

Class Parts and Components

To break down the code in [Listing 8.3](#) further, a class in ABAP has two sections:

► Definition section

In the definition section, various components of a class, such as attributes, events, and methods, are defined. For example, in [Listing 8.3](#), we defined the attributes and the methods of the class in the definition section of the class. When defining the methods, we also defined the parameter interface for the methods.

For the sake of simplicity, we haven't defined any events in [Listing 8.3](#), but if you want to trigger various events to which the methods of the class can respond, you can do so in the definition part of the class. The methods that respond to events are called *event handler methods*. We'll learn more about defining events and implementing suitable event handler methods in [Section 8.1.4](#).

► Implementation section

The implementation section of the class is where we implement the methods defined in the definition section. The methods are implemented between the `METHOD` and `ENDMETHOD` statements, and each method defined in the definition part must have an implementation in the implementation part of the class.

There are three important types of components available in a class:

► **Methods**

Methods are used to change the state of an object and manipulate its behavior. For example, in [Listing 8.3](#), different methods were implemented in the `CL_PERSON` class to set the height, weight, and BMI of the person. Methods can also be defined as event handler methods that respond to specific events during the runtime of the object. By using setter and getter methods, we can ensure that all attributes of the class are always accessed through its methods, thus guaranteeing data consistency.

► **Attributes**

Attributes are the internal data fields of a class and store the state of an object. We can change the state of an object by changing the contents of an attribute.

► **Events**

Events are defined in the definition area of a class and can be raised during the runtime of the object. A suitable event handler method is maintained in the implementation part of the class and is triggered when the corresponding event is raised. Events allow you to call specific methods dynamically during the runtime of the object.

For example, if you want to perform specific tasks when the user double-clicks an output line, you can maintain a `DOUBLE_CLICK` event, which can be raised on that event. The registered event handler method will be called dynamically when the event is raised.

The components of a class can be categorized as follows:

► **Static components**

Static components can be accessed directly using the class component selector `=>`. Static components behave similarly to function modules in that only one instance of a static component exists in the program.

► **Instance components**

Instance components can't be accessed directly using the class name. To access an instance component, you need to define a reference object in the program using `TYPE REF TO` (for example, the objects `John` and `Mark` in [Listing 8.3](#)) and then access its attributes and methods through the reference object using the object component selector `->`.

Before the attributes and methods can be accessed through the reference object, the reference object needs to be instantiated using the `CREATE OBJECT` statement, which creates an instance of the class (an object described by the class) in the program.

We'll discuss static and instance components further in [Section 8.1.3](#).

Visibility

The components of a class can have different visibility sections, such as `public`, `protected`, and `private`. The visibility is set in the definition part of the class. Component visibility allows for data encapsulation and hiding the implementation from the external world.

Encapsulation is an important concept in OOP, and we'll discuss visibility further in [Section 8.2](#).

8.1.2 Methods

A *method* can access other components of the same object. To access the attributes and methods of the same class, a special self-reference variable, `me`, is used. Even though the components of the class can be accessed directly from within the methods of the same class, for better readability it's recommended to use the self-reference variable `me` when accessing the same class components. You can access both static and instance components from within an instance method, whereas you only can access static components from within a static method.

To call a method of the class, we can either use the `CALL METHOD` statement (e.g., `CALL METHOD John->get_bmi`) or use the object component selector and parentheses directly (e.g., `john->get_bmi()`).

A method can have importing, exporting, changing, and returning parameters. The importing, exporting, and changing parameters behave similarly to the parameters we define in function modules. For example, data is imported through importing parameters and exported through exporting parameters, while changing parameters can be used for both importing and exporting.

The methods that use returning parameters are called *functional methods*. Prior to SAP NetWeaver 7.4, functional methods could only have one returning parameter and no exporting parameters. However, with SAP NetWeaver 7.4, this restriction

is lifted. Functional methods are typically used when a result is returned to the caller program that can be directly assigned to a data object in the program.

For example, with the statement `v_bmi = oref->get_bmi()`, `v_bmi` is the data object, `oref` is the reference object, and `get_bmi` is a method with a returning parameter the value of which will be assigned to the `v_bmi` variable.

Inheritance

Often, many objects are different, but similar enough to be put into a single class. For example, animals and humans have heartbeats, so a class may have a `get_heart_beat` method. However, dogs have tail lengths, so should this class also have a `get_tail_length` method? No, we can define another class as a child class (called a *subclass*) of the parent class. This subclass will have all the features of the parent class plus some additional features. This allows you to extend the functionality of your applications while leveraging the existing code.

Let's look at an example. Objects are designed to closely represent real-life objects—such as a printer. A printer may have an external interface, including a paper tray, keypad, and so on, and the basic functionality of the printer is to create a printout. OOP objects can be designed to represent that setup.

Therefore, let's create a class called `printer` with a method called `print`, as shown in [Listing 8.4](#).

```
CLASS PRINTER DEFINITION.
PUBLIC SECTION.
METHODS print.
ENDCLASS.
CLASS PRINTER IMPLEMENTATION.
METHOD print.
WRITE 'document printed'.
ENDMETHOD.
ENDCLASS.
```

Listing 8.4 Parent Class for Printer

This is a basic printer, but what if we want to extend its functionality? For example, if we have a new printer model that has a counter feature to keep track of the number of copies printed, do we have to build all of the printer logic from scratch? No, we can define a subclass, as shown in [Listing 8.5](#), that inherits all the functionality of the basic printer and extends it.

```

CLASS PRINTER_WITH_COUNTER DEFINITION INHERITING FROM PRINTER.
PUBLIC SECTION.
DATA counter TYPE I.
METHODS print REDEFINITION.
ENDCLASS.
CLASS PRINTER_WITH_COUNTER IMPLEMENTATION.
METHOD print.
Counter = counter + 1.
CALL METHOD SUPER->PRINT.
ENDMETHOD.
ENDCLASS.

```

Listing 8.5 Subclass Implementation

In [Listing 8.5](#), the class `PRINTER_WITH_COUNTER` is inherited from the class `printer`. Here, `printer` is the superclass (parent class) and `PRINTER_WITH_COUNTER` is the subclass (child class). This process of a subclass inheriting the traits of a superclass is called *inheritance* in OOP.

Notice the keyword `INHERITING FROM` used in the definition of the class `PRINTER_WITH_COUNTER`. The subclass inherits all the components of the parent class. It will, by default, have all the attributes and methods defined in the parent class. If required, the methods inherited in the subclass can be redefined to add functionality.

In [Listing 8.5](#), the `print` method in the `PRINTER_WITH_COUNTER` subclass is redefined to increment the `counter` attribute and calls the `print` method of the parent class. This effectively extends the functionality of the method while leveraging the existing functionality provided by the method of the parent class. Because the same `print` method behaves differently in the superclass and subclass, it leads to some interesting behavior called *polymorphism* (meaning *having many forms*), which we'll discuss in [Section 8.4](#).

Now we have a printer with a counter, and we know that the counter increments by one every time a document is printed. However, what will its initial value be? How do we ensure that the counter always starts from one or any other initial value? These details are sorted out by defining a constructor for the class.

There are two types of constructors:

► **Instance constructor**

An *instance constructor* is a special method called when the object is *instantiated*. The instance constructor is called for each object when it is instantiated.

► **Class constructor**

The *class constructor* is special method called just once for the whole class in one internal mode. A class constructor can have a maximum of one instance and one static constructor. A static constructor can't have any parameters, whereas the instance constructor can have only importing parameters.

By defining importing parameters for the instance constructor, we can ensure that the object is always initialized before any of its attributes or methods are accessed.

Listing 8.6 shows the implementation of the constructor.

```

CLASS PRINTER_WITH_COUNTER DEFINITION INHERITING FROM PRINTER.
PUBLIC SECTION.
DATA counter TYPE I.
METHODS constructor IMPORTING count TYPE I.
METHODS print REDEFINITION.
ENDCLASS.

CLASS PRINTER_WITH_COUNTER IMPLEMENTATION.
METHOD constructor.
CALL METHOD super->constructor.
counter = count.
ENDMETHOD.
METHOD print.
counter = counter + 1.
CALL METHOD SUPER->PRINT.
ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
DATA oref TYPE REF TO PRINTER_WITH_COUNTER.
CREATE OBJECT oref
EXPORTING
count = 0.

```

Listing 8.6 Class with Constructor

In Listing 8.6, the constructor method is defined in the definition part of the `PRINTER_WITH_COUNTER` class with one importing parameter, `count`. Because the `PRINTER_WITH_COUNTER` class is a subclass of the `printer` class, the superclass constructor must be called first in the implementation of the constructor method, even if there's no constructor defined in the superclass. This ensures that all of the superclass components in the inheritance tree are initialized properly. If a constructor is defined in the superclass, it won't be inherited by the subclass. This

allows each class in the inheritance tree to have its own constructor in order to initialize its attributes per requirements.

In the example in [Listing 8.6](#), the `count` parameter of the constructor is assigned a value when instantiating the reference object `oref`. The syntax checker will give an error if the constructor's mandatory parameters aren't passed while instantiating the reference object. This ensures that the object is always instantiated properly.

Now, say we have a more advanced printer model that can print multiple copies along with the basic features provided so far. To create this printer, we'll inherit the `PRINTER_WITH_COUNTER` class and add additional functionality, as shown in [Listing 8.7](#).

```

CLASS MULTI_COPY_PRINTER DEFINITION INHERITING FROM PRINTER_WITH_
COUNTER.
PUBLIC SECTION.
DATA copies TYPE I.
METHODS set_copies IMPORTING copies TYPE I.
METHODS print REDEFINITION.
ENDCLASS.

CLASS MULTI_COPY_PRINTER IMPLEMENTATION.
METHOD set_copies.
Me->copies = copies.
ENDMETHOD.
METHOD print.
Do copies TIMES.
CALL METHOD SUPER->Print.
ENDDO.
ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
DATA oref TYPE REF TO MULTI_COPY_PRINTER.
CREATE OBJECT oref
EXPORTING
count = 0.
oref->set_copies( 5 ).
oref->print( ).

```

Listing 8.7 Class Extended to Print Multiple Copies

In [Listing 8.7](#), we defined the `MULTI_COPY_PRINTER` class, inheriting from `PRINTER_WITH_COUNTER`. In the `MULTI_COPY_PRINTER` class, we defined a method called

`set_copies` that receives the `copies` importing parameter and assigns the value to the `copies` attribute. We redefined the `print` method to print multiple copies by calling the method in the superclass in a `DO` loop.

It's good practice not to allow direct access to the class attributes and to use methods to set or get attribute values. This gives us more control while manipulating attributes. The methods that set values are called *setter methods* and the methods that are used to get values are called *getter methods*. Setter methods are prefixed with `set_` and getter methods are prefixed with `get_`.

Encapsulation

Now, say our printer is an industrial printer and we want to set a restriction for the minimum number of prints. For example, the printer should only take a request if a minimum of fifty copies is requested. We implement this restriction by writing an `IF` condition in the `set_copies` method, as shown in [Listing 8.8](#).

```
METHOD set_copies.
  IF copies GE 50.
    Me->copies = copies.
  ELSE.
    *Raise exception
  ENDIF.
ENDMETHOD.
```

Listing 8.8 Restricting Copies in `set_copies` Method

The code in [Listing 8.8](#) adds the restriction to set the `copies` attribute only if the value is greater than or equal to 50. However, there's one loophole in this implementation: Because the `copies` attribute is defined with its visibility as `public`, anyone can directly access this attribute from outside the class and set a value, thus bypassing the check in the `set_copies` method, as shown in [Listing 8.9](#).

```
DATA oref TYPE REF TO MULTI_COPY_PRINTER.
CREATE OBJECT oref
EXPORTING
count = 0.
oref->copies = 5.
oref->print( ).
```

Listing 8.9 Bypassing `set_copies` Method to Set Number of Copies

As you can see in [Listing 8.9](#), we can directly access the `copies` attribute to set the number of copies. This is where *encapsulation* and *implementation hiding* plays an

important role. By defining clear boundaries on what can be accessed from outside the class and what should be restricted, we not only make it easy for developers to use our classes, because they only need to know the public interface (components that are publicly accessible) of our class, but also ensure that inadvertent bugs don't creep into our software implementation.

Encapsulation also gives you the freedom to change the private sections of the class according to changing requirements without having to worry about dependencies. For example, if you change the interface of a method in the public visibility section, all the programs that call this method will be affected; however, because no programs can directly access the private section of a class, you can change private sections freely without having to worry about program dependencies.

In this example, we can make the attribute `copies` private so that it can be accessed only from the methods of the same class, as shown in [Listing 8.10](#).

```
CLASS MULTI_COPY_PRINTER DEFINITION INHERITING FROM PRINTER_WITH_
COUNTER.
PUBLIC SECTION.
METHODS set_copies IMPORTING copies TYPE I.
METHODS print REDEFINITION.
PRIVATE SECTION.
DATA copies TYPE I.
ENDCLASS.
```

Listing 8.10 Making Attribute Private

As you can see, we made our printer more and more complex without having to duplicate any code. Also, as the printer added advanced capabilities, it still performed the basic task of printing a document. Therefore, a user using the advanced model of the printer need not be aware of its advanced features to perform the basic task of printing a document. This is similar to a car with advanced features like cruise control. The driver need not know how to set cruise control to drive the car, but he can learn to set cruise control to take advantage of it.

In addition, if the `print` method in the `printer` superclass is enhanced, then all the enhancements will be available by default to all the subclasses without having to change any of them.

8.1.3 Instance and Static Components

A class can contain both instance and static components. An instance attribute can have multiple instances in the program, whereas a static attribute has only one instance. An instance component is accessed through a reference object using the object component selector `->`, whereas a static component is accessed using the class component selector `=>`. Static components are preceded by the `CLASS` keyword during the definition.

Listing 8.11 shows the definition of both instance and static components. For simplicity, we're accessing the attributes of the class directly in this example, but in real-world code direct access to attributes should always be avoided by using setter and getter methods.

```

CLASS CL_PERSON DEFINITION.
  PUBLIC SECTION.
    CLASS-DATA height TYPE I.           "Static attribute
    DATA weight TYPE I.                "Instance attribute
    METHODS get_bmi EXPORTING bmi TYPE I. "Instance method
    METHODS constructor                 "Instance constructor
      IMPORTING name TYPE CHAR20.
    CLASS-METHODS set_height          "Static Method
      IMPORTING height TYPE I.
    CLASS-METHODS class_constructor.  "Static Constructor
ENDCLASS.

CLASS CL_PERSON IMPLEMENTATION.
  METHOD class_constructor.
  ENDMETHOD.
  METHOD constructor.
  ENDMETHOD.
  METHOD get_bmi.
  ENDMETHOD.
  METHOD set_height.
  ENDMETHOD.
ENDCLASS.
DATA oref TYPE REF TO cl_person. "Defining reference object
DATA v_bmi TYPE I.
START-OF-SELECTION.
*Accessing static attributes and methods using the selector "=>"
*with class reference
cl_person=>height = 165.
CALL METHOD cl_person=>set_height
EXPORTING
  height = 165.
*Instantiating the reference object and accessing the instance

```



```

*attributes and methods using the selector "->" with object
*reference.
CREATE OBJECT oref
EXPORTING
    name = 'JOHN'.
oref->weight = 50.
CALL METHOD oref->get_bmi
IMPORTING
    bmi = v_bmi.

```

Listing 8.11 Instance and Static Class Components

As shown, static components are defined with the preceding keyword `CLASS` (e.g., `CLASS-DATA`, `CLASS-METHODS`) and accessed using the `=>` selector with the class name reference directly (e.g., `class_name=>component`). Static methods are somewhat similar to function modules in that only one instance exists in the program memory and each subsequent call to the method points to the same instance, whereas each reference object works with its own instance components.

A static constructor can't have any parameter interface, whereas an instance constructor can have importing parameters. A static constructor is executed only once on the first access of any of the class components, including instantiation of any reference object. An instance constructor is executed each time a reference object is instantiated.

Table 8.1 compares and contrasts instance and static components.

Instance Component	Static Component
Instance components are accessed through a reference object with the <code>-></code> selector.	Static components can be accessed directly through the class name reference with the <code>=></code> selector even before a reference object is instantiated. They can also be accessed using a reference object.
Multiple instances can exist.	Only single instance exists.
Within an instance method, both static and instance attributes can be accessed.	Within a static method, only static attributes or static methods can be accessed.
To access the instance components of a class, a reference object needs to be defined in the program and instantiated using <code>CREATE OBJECT</code> statement.	Static components can be accessed directly from program using a class reference (similar to calling a function module directly) without the need for a reference object.

Table 8.1 Differences between Instance and Static Components

Instance Component	Static Component
An instance constructor only can have importing parameters and is executed each time a reference object is instantiated.	A static constructor can't have any parameter interface and is executed only once per program session. However, it's executed before the instance constructor, if the class has both static and instance constructors, and when the first reference object is instantiated, provided it isn't loaded previously.

Table 8.1 Differences between Instance and Static Components (Cont.)

8.1.4 Events

Events can be defined in a class to be triggered from a method (called *triggers*) when a specific event takes place during the runtime of the object; specific methods (called *event handlers*) react to these events. For example, we can define an event called `double_click` in the class to be triggered in one of the methods when the user double-clicks the report output. The `on_double_click` method can be defined as an event handler for this event so that it's executed when the `double_click` event is triggered from the trigger method.

To trigger an event, the class must have the events defined in the definition part of the code and triggered in one of its methods. *Instance events* are defined using the `EVENTS` keyword and *static events* are defined using the `CLASS-EVENTS` keyword. A static event can be triggered from both static and instance methods, whereas, an instance event can only be triggered from an instance method.

Events can have exporting parameters that are passed to the event handler method when the event is triggered. To trigger an event, the statement `RAISE EVENT event_name` is used in one of the methods of the class. The `EXPORTING` addition is used with the `RAISE EVENT` keyword to pass the event parameters to the event handler method.

Event handlers are special methods that are defined to react to the event. Event handler methods are defined in the definition part of the class as methods for specific events. The syntax to define an event handler method is `METHODS method_name FOR EVENT event_name OF class_name`, where `method_name` is the name of the event handler method, `event_name` is the name of the event, and `class_name` is the name of the class in which the event is defined. If the event has exporting

parameters defined, then the event handler must import all non-optional parameters. All instance events have an implicit `sender` parameter that can be imported in the event handler. The `sender` parameter contains a reference to the sender object in which the event is defined. To define a static method as an event handler, use the `CLASS-METHODS` keyword.

To allow an event handler method to react to an event, we must determine at runtime the trigger to which it is to react. We can do that by using the `SET HANDLER` statement. The `SET HANDLER` statement registers the event handler methods with the event triggers.

Instance events can be registered to a specific instance or to all instances, whereas static events are registered to the whole class. The syntax to use `SET HANDLER` statement for instance events is as follows:

```
SET HANDLER event_handler_method FOR oref.
```

or:

```
SET HANDLER event_handler_method FOR ALL INSTANCES.
```

The syntax to use the `SET HANDLER` statement for static events is as follows:

```
SET HANDLER event_handler_method.
```

Here, `event_handler_method` is an instance or static method that is defined as an event handler for an event. The event handler method will not be triggered if the handler is not registered using the `SET HANDLER` statement.

[Listing 8.12](#) shows an example to define and register events in a class. This example defines the `cl_events_demo` class with two events, `double_click` and `right_click`. Here, `double_click` is defined as an instance event and `right_click` is defined as a static event. The `double_click` event also has a couple of mandatory exporting parameters (`column` and `row`) defined. We've also defined a `trigger_event` method to act as a trigger method. This is defined similarly to any regular method. The `on_double_click` and `on_right_click` methods are defined as event handler methods for the `double_click` and `right_click` events, respectively.

```
CLASS cl_events_demo DEFINITION.
  PUBLIC SECTION.
    EVENTS double_click
    EXPORTING
      VALUE(column) TYPE i
      VALUE(row) TYPE i .
  CLASS-EVENTS right_click .
```

```

METHODS trigger_event.
METHODS on_double_click FOR EVENT double_click OF cl_events_demo
  IMPORTING
    column
    row
    sender.
METHODS on_right_click FOR EVENT right_click OF cl_events_demo.
ENDCLASS.
CLASS cl_events_demo IMPLEMENTATION.
METHOD trigger_event.
  RAISE EVENT double_click
  EXPORTING
    column = 4
    row    = 5.
  RAISE EVENT right_click.
ENDMETHOD.
METHOD on_double_click.
WRITE: / 'Double click event triggered at column', column,
        'and row', row.
ENDMETHOD.
METHOD on_right_click.
WRITE: / 'Right click event triggered'.
ENDMETHOD.
ENDCLASS.
DATA oref TYPE REF TO cl_events_demo.
START-OF-SELECTION.
  CREATE OBJECT oref.
  SET HANDLER oref->on_double_click FOR oref. "Instance event
  SET HANDLER oref->on_right_click. "Handler for Static event
  CALL METHOD oref->trigger_event( ).

```

Listing 8.12 Defining Events

In the implementation part of the class, the `trigger_event` method uses the `RAISE EVENT` statement to trigger the `double_click` and `right_click` events. Because the `double_click` event contains defined exporting parameters, this method exports certain values to the event handler method. The `on_double_click` method imports the values that were exported by the trigger method. In the program code, the events are registered using the `SET HANDLER` statement, and the code in the event handler methods is executed when the `trigger_event` method is called.

In this section, we introduced the basics of OOP, including classes, methods, instance and static components, and event. Working with objects simplifies the implementation of functionality. OOP concepts provide many advantages compared to procedural programming techniques. Programs developed using OOP techniques are easy to maintain and, if designed well, should be easy to enhance.

Now that you've been introduced to the basics of OOP, in the next few sections we'll discuss some of the characteristics of this kind of programming, beginning with the process of encapsulation.

8.2 Encapsulation

In procedural programming, developers generally use a technique called *functional decomposition* for application development design. In functional decomposition, the series of steps that the application needs to perform are identified from the functional requirements, and procedures are developed for each step. In other words, complex business requirements are broken down into smaller, easy to understand steps. Once all the procedures are in place, they are composed into the main program, which calls the procedures in a fixed order and passes the relevant data to each procedure.

For example, if you want to develop software for an ATM machine, then the requirements can be broken down into smaller steps, such as reading a card, verifying its PIN, dispensing cash, and so on. Modules can then be developed separately for each of these steps, such as one module to read the card, another module to validate the PIN, another module to verify the account balance and dispense cash, and so on. These modules can then be combined to form the overall application. By breaking down the requirements into smaller steps that are easy to understand and maintain, this approach makes it easy to develop small- to medium-sized applications.

However, because procedures (function modules or subroutines) don't have their own data environment, they're dependent on the main program to supply the data each time a procedure is called, or they must work with the global data of the program. This makes the procedures tightly coupled with the main program. As the application branches out and becomes more and more complex, it may lead to maintenance headaches.

For example, if many procedures are manipulating the global data, we can't be sure if the application will behave in a predictable way if the sequence of procedure calls is changed for any future enhancements. By depending entirely on the main program to supply the required data each time the procedure is called, the interface will soon be cluttered, and it makes the procedure too restrictive to work in other environments. The developer of the procedure depends too much

on the external application, over which he has no control. It also becomes difficult to develop loosely coupled procedures that can work seamlessly for future enhancements. In addition, once the procedure is developed, the developer can't change the parameter interface for future enhancements without breaking existing applications that call the procedure.

In OOP, because objects have their own data environment, they can be made smart enough to make their own decisions without too much dependency on external applications.

Encapsulation in OOP allows you to define boundaries and hide implementation from the external world. The term *encapsulation* means that the attributes (data) and the methods (functions) that manipulate the data are enclosed in a capsule (object). This means that we can set a boundary between what can be accessed from within the object and from the outside world. This boundary helps to address many of the issues mentioned previously regarding procedures.

In this section, we'll discuss some of the characteristics of encapsulation in OOP. We'll look at component visibility, the different visibility sections, friendship between classes, and implementation hiding.

8.2.1 Component Visibility

As shown back in [Listing 8.7](#), the component of an object can be accessed from an *external program* or from a *subclass* or from within the class. The access restriction on a component of a class is what defines its visibility. In other words, we can decide if the component of a class can be accessed only from within the class or its subclasses or from external programs.

Back in [Listing 8.10](#), we defined sections (PUBLIC SECTION, PRIVATE SECTION) under which components are declared. These sections are called *visibility sections*. There are three visibility sections: *public*, *protected*, and *private*.

The visibility section identifies how the component of a class can be accessed. The following list defines the different visibility sections:

- ▶ **Public section**

The components in the *public section* can be accessed from within the class and outside of the class. These components form the public interface of the class.

► **Protected section**

The components in the *protected section* can be accessed only from within the class and its subclasses (child classes). These components can't be accessed from external programs.

► **Private section**

The components in the *private section* can be accessed only from within the class and its friend classes (we'll discuss friend classes in [Section 8.2.2](#)).

[Listing 8.13](#) shows an example of the visibility sections.

```

CLASS c1_encapsulation_demo DEFINITION.
  PUBLIC SECTION.
    METHODS print.
    METHODS set_copies IMPORTING copies TYPE i.
    METHODS get_counter EXPORTING counter TYPE i.
  PROTECTED SECTION.
    METHODS reset_counter.
  PRIVATE SECTION.
    DATA copies TYPE i.
    DATA counter TYPE i.
    METHODS reset_copies.
ENDCLASS.
CLASS c1_encapsulation_demo IMPLEMENTATION.
  METHOD print.
    "Business logic goes here
  ENDMETHOD.
  METHOD set_copies.
    "Business logic goes here
    me->copies = copies.
  ENDMETHOD.
  METHOD get_counter.
    "Business logic goes here
    counter = me->counter.
  ENDMETHOD.
  METHOD reset_counter.
    "Business logic goes here
    CLEAR counter.
  ENDMETHOD.
  METHOD reset_copies.
    "Business logic goes here
    CLEAR copies.
  ENDMETHOD.
ENDCLASS.
CLASS c1_encapsulation_sub_demo DEFINITION INHERITING FROM c1_
encapsulation_demo.
  PROTECTED SECTION.
    METHODS reset_counter REDEFINITION.

```

```

ENDCLASS.
CLASS cl_encapsulation_sub_demo IMPLEMENTATION.
  METHOD reset_counter.
    super->reset_counter( ).
  *   super->reset_copies( ). "gives syntax error
  ENDMETHOD.
ENDCLASS.

```

Listing 8.13 Implementing Visibility Sections

The code in [Listing 8.13](#) implements the `cl_encapsulation_demo` class, and the `cl_encapsulation_sub_demo` class is defined as a subclass of `cl_encapsulation_demo`. The code shows the syntax to implement visibility sections. Notice the order of implementation of the visibility sections: First, public components are defined, then protected components, and finally private components. This declaration order can't be changed.

In [Listing 8.13](#), the `cl_encapsulation_sub_demo` subclass can access only the public and protected components of its superclass, `cl_encapsulation_demo`. If the `cl_encapsulation_sub_demo` class tries to access the private components of its superclass, the attempt will result in a syntax error. Note that when redefining a method of the superclass in a subclass, the visibility section of the method can't be changed. For example, if the method is defined in the protected section of the superclass, then the method should be redefined in the protected section of the subclass.

[Figure 8.1](#) shows the visibility set in Class Builder for global classes.

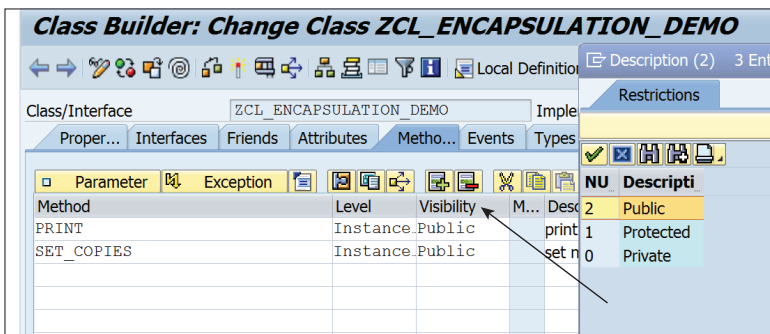


Figure 8.1 Visibility Using Class Builder

With visibility sections, clear boundaries can be defined so that only the required components of an object are exposed to the outside world. This arrangement

provides greater flexibility in designing the objects because the private and protected components of a class can be changed per changing business requirements without affecting the external programs, which only need to worry about the public interface.

By keeping object attributes private and implementing setter and getter methods in the class, we have more control and security over the object's data; any change to the attributes should be requested through the setter methods, and any request for information should be requested through getter methods. This arrangement allows for better abstraction and prevents undesired data corruption scenarios, which in turn helps objects perform in a predictable way.

For example, by implementing a setter method to set the value of an attribute, we can ensure that all the business logic is applied before the value is changed. This level of certainty isn't possible if the attribute is directly accessed by an external program to assign a value over which we have no control.

8.2.2 Friends

A class can declare another class as its friend to allow access to all of its components, including its private and protected components. A *friend class* is any class that is explicitly defined as a friend in the class definition. The friendship relation is not reciprocal; for example, if class `c1` declares class `c2` as a friend, then the methods of the class `c2` can access the private components of class `c1`, but the methods of class `c1` can't access the private components of class `c2`.

Listing 8.14 expands the code from Listing 8.13 to show the implementation of friend classes.

```

CLASS c1 DEFINITION DEFERRED.
CLASS c2 DEFINITION DEFERRED.
CLASS c1_encapsulation_demo DEFINITION FRIENDS c1 c2.
  PUBLIC SECTION.
    METHODS print.
    METHODS set_copies IMPORTING copies TYPE i.
    METHODS get_counter EXPORTING counter TYPE i.
  PROTECTED SECTION.
    METHODS reset_counter.
  PRIVATE SECTION.
    DATA copies TYPE i.
    DATA counter TYPE i.
    METHODS reset_copies.
ENDCLASS.

```

```

CLASS c1_encapsulation_demo IMPLEMENTATION.
  METHOD print.
    "Business logic goes here
  ENDMETHOD.
  METHOD set_copies.
    "Business logic goes here
    me->copies = copies.
  ENDMETHOD.
  METHOD get_counter.
    "Business logic goes here
    counter = me->counter.
  ENDMETHOD.
  METHOD reset_counter.
    "Business logic goes here
    CLEAR counter.
  ENDMETHOD.
  METHOD reset_copies.
    "Business logic goes here
    CLEAR copies.
  ENDMETHOD.
ENDCLASS.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    METHODS get_counter IMPORTING counter TYPE i.
  PROTECTED SECTION.
    DATA counter TYPE i.
ENDCLASS.
CLASS c1 IMPLEMENTATION.
  METHOD get_counter.
    DATA oref TYPE REF TO c1_encapsulation_demo.
    CREATE OBJECT oref.
    oref->reset_counter( ).
    oref->counter = counter.
  ENDMETHOD.
ENDCLASS.
CLASS c2 DEFINITION.
  PUBLIC SECTION.
    METHODS set_copies IMPORTING copies TYPE i.
  PRIVATE SECTION.
    DATA copies TYPE i.
ENDCLASS.
CLASS c2 IMPLEMENTATION.
  METHOD set_copies.
    DATA oref TYPE REF TO c1_encapsulation_demo.
    CREATE OBJECT oref.
    oref->reset_copies( ).
    oref->copies = copies.
  ENDMETHOD.
ENDCLASS.

```

```

CLASS c1_encapsulation_sub_demo DEFINITION INHERITING FROM c1_
encapsulation_demo.
    PROTECTED SECTION.
        METHODS reset_counter REDEFINITION.
    ENDClass.
CLASS c1_encapsulation_sub_demo IMPLEMENTATION.
    METHOD reset_counter.
        super->reset_counter( ).
        super->reset_copies( ). "Gives syntax error
    ENDMETHOD.
ENDCLASS.

```

Listing 8.14 Visibility with Friends

The code in [Listing 8.14](#) implements the `c1_encapsulation_demo` class and defines the `c1` and `c2` classes as friends via the `FRIENDS` addition in the `CLASS DEFINITION` statement. Any number of classes can be listed with the `FRIENDS` addition, separated by spaces. Also notice the `CLASS c1 DEFINITION DEFERRED` and `CLASS c2 DEFINITION DEFERRED` statements at the beginning of the code. These two statements tell the syntax checker that the classes `c1` and `c2` are defined later in the code; otherwise, the syntax checker will give an error when the `c1_encapsulation_demo` class refers to the `c1` and `c2` classes as friends in its definition.

In [Listing 8.14](#), the `c1` and `c2` classes can access the public, private, and protected components of `c1_encapsulation_demo` because they're defined as friends for that class. However, the `c1_encapsulation_demo` class can't access the protected or private sections of the `c1` or `c2` classes because the relationship isn't reciprocal. Without this restriction, any class can declare itself as a friend of another class to access its private or protected components, which defeats the purpose of having visibility sections.

[Figure 8.2](#) shows the friend classes defined under the `FRIENDS` tab in Class Builder, where you can maintain friend classes for a global class.

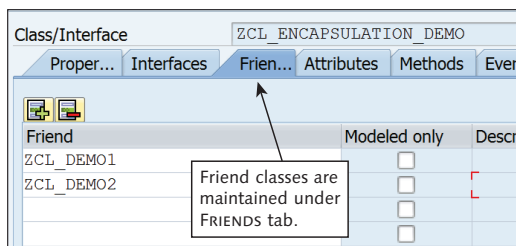


Figure 8.2 Friends Definition in Class Builder

8.2.3 Implementation Hiding

Hiding implementation from the outside world allows for better implementation control while providing flexibility for future enhancements due to ever-changing business requirements. Keeping the public interface to a minimum allows you to change the private and protected sections of the class per changing requirements without having to worry about existing applications that call the class. This saves a lot of headaches during enhancements and maintenance.

Let's consider a use case to understand how implementation hiding is useful. For this example, assume you're developing an app for a mobile operating system such as iOS or Android. One of the features in this software is to show notifications to the user from different apps. To facilitate this, define a public attribute in the application programming interface (API) class that can be accessed by app developers to send notifications. However, this way, you can't have control over the number of notifications displayed by the third-party app or control any spam activity. [Listing 8.15](#) shows the sample API code.

```

CLASS c1_notification_api DEFINITION.
PUBLIC SECTION.
DATA message TYPE string.
METHODS set_message IMPORTING im_message TYPE string.
METHODS display_notification.
ENDCLASS.
CLASS c1_notification_api IMPLEMENTATION.
METHOD set_message.
message = im_message.
ENDMETHOD.
METHOD display_notification.
WRITE message.
ENDMETHOD.
ENDCLASS.
*Code in the calling program.
DATA notify TYPE REF TO c1_notification_api.
CREATE OBJECT notify.
notify->set_message( im_message = 'My App notification' ).
Notify->display_notification( ).

```

Listing 8.15 Making All Implementation Public

In [Listing 8.15](#), the `c1_notification_api` class is defined with a `message` attribute and the `set_message` and `display_notification` methods. Let's assume `c1_notification_api` works as an API that can be called from third-party apps to display notifications to the user.

The attribute `message` holds the message that can be set by calling the `set_message` method. The notification is displayed by calling the `display_notification` method. In this class, all of the components have public visibility.

Because all of the components are visible outside the class, any change to them will break the implementation in all the apps that use this API. For example, implementing additional features such as filtering a message or restricting the number of messages per app may be difficult with this design.

The `set_message` setter method is provided to set the message after screening it (such as screening for inappropriate words). However, this restriction can be bypassed easily by third-party app developers, because the `message` attribute is also accessible from outside the class. Therefore, outside developers can directly access the attribute to set the message instead of calling the `set_message` method.

The API in [Listing 8.15](#) can be enhanced by hiding the implementation, as shown in [Listing 8.16](#).

```

CLASS cl_notification_api DEFINITION.
  PUBLIC SECTION.
    METHODS set_message IMPORTING im_message TYPE string.
    METHODS display_notification.
  PRIVATE SECTION.
    DATA MESSAGE TYPE string.
    METHODS filter_message RETURNING VALUE(boolean) TYPE boolean.
    METHODS check_count RETURNING VALUE(boolean) TYPE boolean.
    METHODS check_status RETURNING VALUE(boolean) TYPE boolean.
ENDCLASS.
CLASS cl_notification_api IMPLEMENTATION.
  METHOD set_message.
    MESSAGE = im_message.
  ENDMETHOD.
  METHOD display_notification.
    IF me->filter_message( ) EQ abap_true OR
       me->check_count( ) EQ abap_true OR
       me->check_status( ) EQ abap_true.
      WRITE MESSAGE.
    ELSE.
      CLEAR message.
    ENDIF.
  ENDMETHOD.
  METHOD filter_message.
*Filtering logic goes here and the parameter "Boolean" is set to
*abap_true or abap_false accordingly.
  ENDMETHOD.
  METHOD check_count.

```

```

*Logic to check number of messages goes here and the parameter
*"Boolean" is set to abap_true or abap_false accordingly.
    ENDMETHOD.
    METHOD check_status.
*Logic to check user personal setting goes here and the parameter
*"Boolean" is set to abap_true or abap_false accordingly.
    ENDMETHOD.
ENDCLASS.
*Code in the calling program.
DATA notify TYPE REF TO cl_notification_api.
CREATE OBJECT notify.
notify->set_message( im_message = 'My App notification' ).
Notify->display_notification( ).

```

Listing 8.16 Hiding Implementation

The code in [Listing 8.16](#) now moves the `message` attribute to the private section so that it can't be accessed outside of the class, meaning that its implementation is hidden from the outside world. In addition, three new methods are defined in the private section to perform various checks on the message before it's displayed to the user; for example, if the user has turned off notifications for the app, the program can check for that information in the `check_status` method, and the message is not displayed if the `check_status` method returns the Boolean value `abap_false`. Similarly, if the user has set a restriction not to display more than five notifications per day for an app, the `check_count` method can handle that validation. The validating methods used in the private section can be enhanced or changed without having to worry about any dependencies.

External developers need not know about the intricacies of the validations performed to display their notifications. All they need to know is how to set the message and call the notification API. With the implementation in [Listing 8.16](#), you have total control over the displayed notifications, and there's no way for any checks to be bypassed. This also makes the implementation future-proof, because you can add as many additional checks as you'd like without external developers changing anything about the way the API is called in their applications.

When used the right way, implementation hiding makes software development less prone to errors and easier to enhance or maintain.

Now that we've looked at the primary characteristics of encapsulation, in the next section we'll move onto another OOP concept: inheritance.

8.3 Inheritance

Inheritance is an important concept in OOP that helps us avoid repetition of code and leverage existing code libraries. In [Section 8.1](#), we described how inheritance makes it easy to expand functionality without having to rewrite existing code. With procedural programming techniques, the closest you can get to something similar to inheritance is either editing the existing code to expand the functionality or copying the existing code and then adding the additional functionality. Both approaches are undesirable, because editing existing code may break the original functionality, and it requires a lot of testing to check that the original and the new functionality both work correctly, which can itself be a time-consuming process to go through each time the functionality is enhanced. If we copy the existing code, not only is the code now redundant, but also it leads to a maintenance nightmare, because each update to the original code needs to be replicated in all of the copies.

Inheritance allows you to expand a class to meet more specific requirements while reusing what's already been developed without having to repeat the existing code. For example, say that during the initial stage of development you defined a class to process information about students. Because your requirement didn't differentiate between types of students, you developed a generic student class.

However, later in the development process, the requirement has been enhanced to branch students into various categories—commerce students, IT students, science students, and so on—with each student category required to perform specific tasks. As the requirement becomes more specific, you can simply inherit the original student class to add more specific functionality for each student category, as shown in [Listing 8.17](#). This allows you to move from generic functionality to specific functionality without breaking the existing implementation while reusing the effort that already went into the original development.

```

CLASS cl_student DEFINITION.
    PUBLIC SECTION.
        METHODS tuition_fee.
ENDCLASS.
CLASS cl_commerce_student DEFINITION INHERITING FROM cl_student.
    PUBLIC SECTION.
    METHODS tuition_fee REDEFINITION.
    METHODS subject. "New method for additional functionality
ENDCLASS.

```

```

CLASS c1_commerce_student IMPLEMENTATION.
  METHOD tuition_fee.
    "Logic to calculate tuition fee for commerce students goes here
  ENDMETHOD.
METHOD subject.
ENDMETHOD.
ENDCLASS.

```

Listing 8.17 Inheritance

In [Listing 8.17](#), the new `c1_commerce_student` class is a subclass of the original student superclass from which it's inheriting, `c1_student`.

In this section, we'll look at the different components of the inheritance concept. We'll then look at both abstract and final classes and methods before moving on to composition relationships. In the final subsection, we'll look at how to use the refactoring assistant in Class Builder to move class components in the inheritance tree.

8.3.1 Inheriting Components

In [Listing 8.16](#), we were only concerned with what can be accessed from within and outside of a class. The public section allowed access from external applications, while the private section restricted access to components within the class.

Inheritance brings a new dimension to implementation hiding. With the protected section, you can restrict access to the components from external applications but allow access to the subclasses. Therefore, for external programs, the components defined in the protected section are similar to the components in the private section. This allows you to provide access to child classes without exposing the components of a parent class to the public.

[Listing 8.18](#) provides some sample code to demonstrate inheritance. In this example, we're demonstrating how all the components of the parent class are inherited by the child class.

```

CLASS c1_parent DEFINITION.
  PUBLIC SECTION.
  METHODS set_value IMPORTING value TYPE string.
  PROTECTED SECTION.
  DATA value TYPE string.
  METHODS check_value.
  PRIVATE SECTION.
  METHODS reset_value.

```



```

ENDCLASS.
CLASS cl_parent IMPLEMENTATION.
METHOD set_value.
ENDMETHOD.
METHOD check_value.
ENDMETHOD.
METHOD reset_value.
ENDMETHOD.
ENDCLASS.
CLASS cl_child DEFINITION INHERITING FROM cl_parent.
ENDCLASS.
CLASS cl_child IMPLEMENTATION.
ENDCLASS.
DATA child TYPE REF TO cl_child.
START-OF-SELECTION.
CREATE OBJECT child.
Child->set_value( value = 'child' ).

```

Listing 8.18 Inheritance Example

In [Listing 8.18](#), the `cl_child` class is defined as a subclass of `cl_parent`. Here, `cl_parent` is the superclass, and it implements the `set_value` method in the public section, the `check_value` method and the `value` attribute in the protected section, and the `reset_value` method in the private section.

The `cl_child` class is inherited from `cl_parent` using the inheriting from addition of the `class` definition statement. This sets the parent-child relationship between the two classes. The `cl_child` class doesn't list any components on its own. All the components in the public and protected sections of the `cl_parent` superclass are, by default, available to the `cl_child` subclass. This is shown in the program code under `start-of-selection` (see [Listing 8.18](#)), where a reference object `child` is defined by referring to `cl_child` and the `set_value` method of `cl_parent` is accessed using this `child` class reference with the statement `Child->set_value(value = 'child')`.

Keep the following important points about inheritance in mind:

- ▶ The subclass can access the components defined under public and protected visibility sections in the superclass.
- ▶ You can't define a component in a subclass with the same name as the component in the public or protected sections of the superclass.
- ▶ Because the private section of the superclass is invisible to the subclass, you can define a component in the subclass with the same name as the component in the superclass.

- ▶ The visible instance attributes in the superclass can be accessed from the subclass directly or by using the self-reference variable `me`.
- ▶ The visible instance methods of the superclass can be accessed from the subclass using the pseudo-reference variable `super`.
- ▶ The static attributes of a class are associated with the complete inheritance tree (all the classes in the hierarchy) and not just with a single class, so they can be accessed using any class reference in the hierarchy. For example, if three classes A, B, and C are in inheritance relationship (B inherits A and C inherits B) and class A contains a static attribute `counter`, then this attribute can be accessed as `A=>counter` or `B=>counter` or `C=>counter`.
- ▶ The constructor of the superclass is not inherited by the subclass. This allows the subclass to define its own constructor, but to ensure the components of the superclass are instantiated properly, it's mandatory to call the constructor of the superclass in the constructor of the subclass, for instance constructors. If the subclass implements a static constructor, the runtime environment automatically calls the static constructor of the superclass when the subclass is instantiated if the static constructor of the superclass isn't yet called.
- ▶ The constructor of the class will have visibility to its own components. For example, if a method is redefined in the subclass and this method is accessed in the constructor of both the superclass and subclass, then the constructor of the subclass will execute the redefined method in the subclass, while the constructor of the superclass will execute the original method in the superclass, as shown in [Listing 8.19](#).
- ▶ A method of the superclass can be redefined in the subclass using the `REDEFINITION` addition to the `METHODS` statement in the definition part of the subclass, as shown in [Listing 8.19](#). This allows you to enhance the functionality of the superclass method in the subclass.
- ▶ Any changes in the superclass will be automatically available to the subclass, whereas changes to subclasses don't affect the superclass. For example, if a method of the superclass is enhanced, then the enhancements will be automatically available to the subclass. However, if the same method is changed (redefined) in the subclass, then it won't impact the superclass.
- ▶ The definition of a component of superclass can't be changed in the subclass; in other words, you can redefine a superclass method in a subclass, but you can't change its parameter interface.

```

CLASS cl_parent DEFINITION.
  PUBLIC SECTION.
    METHODS constructor.
  PROTECTED SECTION.
    METHODS meth.
ENDCLASS.
CLASS cl_parent IMPLEMENTATION.
  METHOD constructor.
    Me->meth( ).
  ENDMETHOD.
  METHOD meth.
    WRITE 'I'm in parent class'.
  ENDMETHOD.
ENDCLASS.
CLASS cl_child DEFINITION INHERITING FROM cl_parent.
  PUBLIC SECTION.
    METHODS constructor.
  PROTECTED SECTION.
    METHODS meth REDEFINITION.
ENDCLASS.
CLASS cl_child IMPLEMENTATION.
  METHOD constructor.
    super->constructor( ).
    Me->meth( ).
  ENDMETHOD.
  METHOD meth.
    WRITE 'I'm in child class'.
  ENDMETHOD.
ENDCLASS.
DATA child TYPE REF TO cl_child.
START-OF-SELECTION.
CREATE OBJECT child.

```

Listing 8.19 Constructor Access in Superclass and Subclass

In [Listing 8.19](#), `cl_parent` is defined with an instance constructor and the `meth` method. `cl_child` inherits the `cl_parent` class, and the `meth` method is redefined in the `cl_child` subclass. A reference object, `child`, is defined in the program referring to `cl_child`. When the reference object `child` is instantiated under start-of-selection using the `CREATE OBJECT` statement, it will call the constructor of the `cl_child` class. However, because it's mandatory to call the parent constructor before any instance components can be accessed, the constructor in `cl_child` calls the parent constructor using the `super->constructor()` statement.

The `meth` method is called in the respective constructors of the superclass and subclass. The constructor of the `cl_child` subclass calls the redefined `meth`

method in `cl_child`, whereas the constructor of the `cl_parent` superclass calls the `meth` method defined in `cl_parent`. You can keep a breakpoint on the `CREATE OBJECT` statement and execute the code to see the sequence of the code execution in debug mode for easy understanding.

For global classes in Class Builder, the inheritance relation can be maintained either while creating the class, as shown in [Figure 8.3](#), or under the PROPERTIES tab, as shown in [Figure 8.4](#).

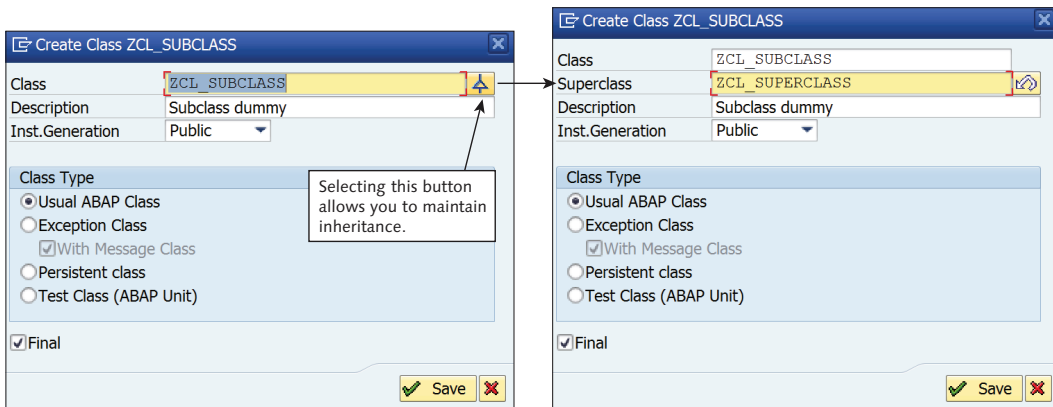


Figure 8.3 Maintaining Superclass Details in Class Builder

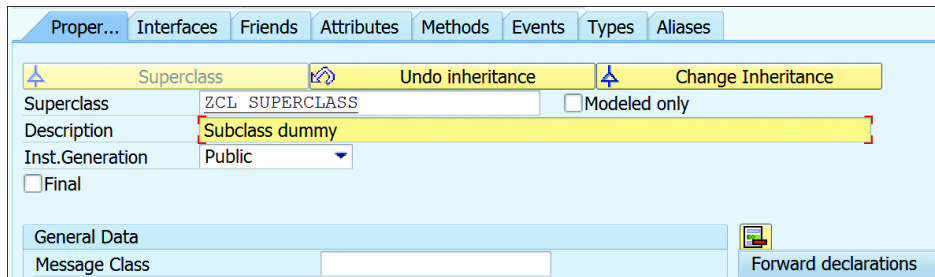


Figure 8.4 Maintaining Inheritance under Class Properties

To redefine a method in the subclass, you can use the `REDEFINE` button in Class Builder, as shown in [Figure 8.5](#).

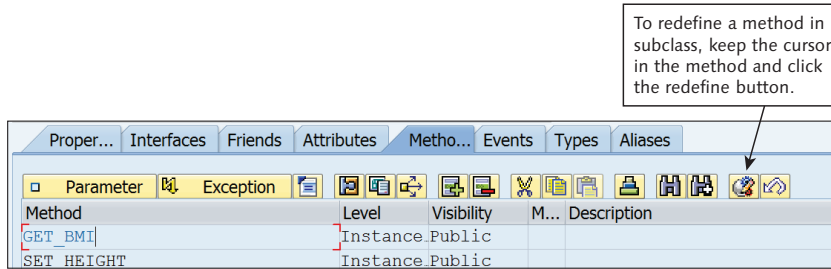


Figure 8.5 Redefine Button in Class Builder

8.3.2 Abstract Classes and Methods

Sometimes, you may want to define a generic class that can be used as a template that can be implemented in subclasses. Defining a dummy superclass with dummy methods may not be a good way of defining this template. For example, if you have students and various course categories and each student course category has a different process to determine its tuition, then you can define a student class with a `tuition_fee` method that can be used as a template, and its implementation can be maintained more specifically in the specific subclasses that inherit this class. For example, a `commerce_student` class can inherit the student class and implement the `tuition_fee` method specific to commerce students.

If you define this student class as a regular class, then you also need to maintain its implementation. However, because this is a generic class, maintaining the implementation doesn't make sense. In such scenarios, we can define this student class as an abstract class and define the `tuition_fee` method as an abstract method.

An *abstract class* only maintains a definition; no implementation is required for such a class if it contains all abstract components, which are inherited in a subclass in which the specific implementation can be maintained. [Listing 8.20](#) provides an example of using an abstract class.

```
CLASS c1_student DEFINITION ABSTRACT.
  PUBLIC SECTION.
    METHODS tuition_fee ABSTRACT.
ENDCLASS.
```

```
CLASS c1_commerce_student DEFINITION INHERITING FROM c1_student.
  PUBLIC SECTION.
```

```

METHODS tuition_fee REDEFINITION.
ENDCLASS.
CLASS c1_commerce_student IMPLEMENTATION.
  METHOD tuition_fee.
    "Logic to calculate tuition fee for commerce students goes here
  ENDMETHOD.
ENDCLASS.

CLASS c1_science_student DEFINITION INHERITING FROM c1_student.
  PUBLIC SECTION.
  METHODS tuition_fee REDEFINITION.
ENDCLASS.
CLASS c1_science_student IMPLEMENTATION.
  METHOD tuition_fee.
    "Logic to calculate tuition fee for science students goes here
  ENDMETHOD.
ENDCLASS.

```

Listing 8.20 Using Abstract Class

In [Listing 8.20](#), `c1_student` is defined as an abstract class by using the `ABSTRACT` addition with the `CLASS DEFINITION` statement.

An abstract method is defined in class `c1_student` using the `ABSTRACT` addition with the `METHODS` statement. Because this class contains an abstract method, it can only be implemented in a subclass by using the `redefinition` addition. If the class contains a regular method (not an abstract method), then we need to maintain the implementation of that method in the implementation part of the class `c1_student`. This implementation then would be part of the template that the subclasses inherit. Because there are no regular methods in [Listing 8.20](#), we've ignored the implementation of the `c1_student` class completely.

The code in [Listing 8.20](#) defines two subclasses—`c1_commerce_student` and `c1_science_student`—which are inherited from the `c1_student` abstract class. The `tuition_fee` method is redefined in the respective subclasses to implement specific functionality.

Note

It isn't possible to create an instance of an abstract class, because an abstract class is only used as a template.

For global classes, the abstract property for a class is set under the `PROPERTIES` tab in the `INST.GENERATION` field, as shown in [Figure 8.6](#).

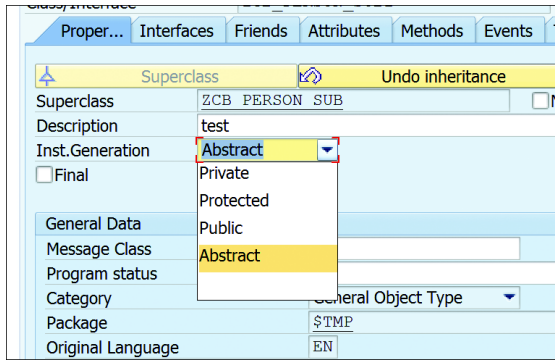


Figure 8.6 Abstract Class in Class Builder

For methods in Class Builder, the abstract property is set by keeping the cursor on the method, selecting the **DETAIL VIEW** button, and then selecting the **ABSTRACT** checkbox, as shown in Figure 8.7.

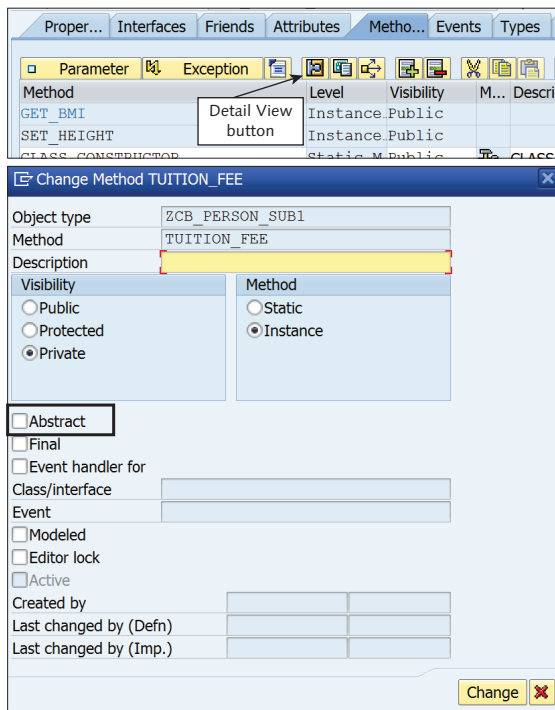


Figure 8.7 Defining Abstract Method in Class Builder

8.3.3 Final Classes and Methods

Sometimes, in the inheritance tree, it may not make sense to expand a class further. In such a scenario, you can make the class *final*, which means it can't be inherited further. A local class can be defined as a final class by adding `FINAL` to the `CLASS DEFINITION` statement, as shown in [Listing 8.21](#).

```
CLASS c1_final DEFINITION FINAL.
.....
ENDCLASS.
```

Listing 8.21 Defining Final Class

For global classes, to define a class as final in Class Builder, the `FINAL` checkbox needs to be selected in the `PROPERTIES` tab, as shown in [Figure 8.8](#).

Methods can also be defined as final so that they can't be redefined further in the subclass. For example, it may make sense to further inherit a class, but a method in the class may not need to be extended further because its purpose is complete. You can make a method final by adding `FINAL` to the `METHODS` statement, as shown in [Listing 8.22](#).

```
CLASS c1_parent DEFINITION.
PUBLIC SECTION.
METHODS student FINAL.
ENDCLASS.
```

Listing 8.22 Final Method

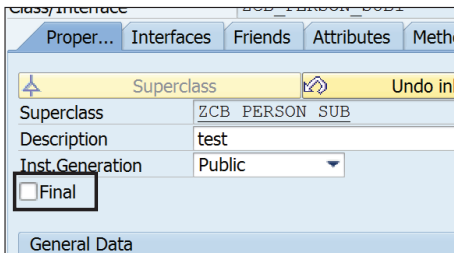


Figure 8.8 Final Class

To make the method of a global class final, keep the cursor on the method in the Class Builder and select the `DETAIL VIEW` button. In the `CHANGE METHOD` window, select the `FINAL` checkbox, as shown in [Figure 8.9](#).

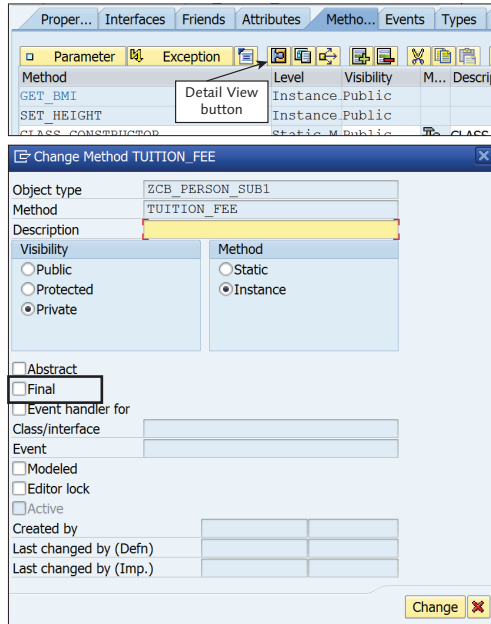


Figure 8.9 Final Method in Class Builder

8.3.4 Composition

Using inheritance, we design classes that fit an *is a* relationship. For example, a commerce student *is a* type of student, so we define the class `commerce_student` as a subclass of `student` in the example in [Listing 8.20](#). Sometimes, in an effort to reuse the existing code, developers create an inheritance tree that doesn't fit an *is a* relationship. For example, if we have an existing `orders` class, it makes sense to create a `sales_order` class inheriting from the `orders` class, because a sales order *is an* order.

However, if you want to define a class for `delivery`, it may not make sense to inherit the `orders` class, because a `delivery` *is not* an order. However, each order *has* one or more deliveries associated with it. If the objects fit the *has a* relationship, we call it a *composition relationship*.

Composition allows you to reuse existing functionality by maintaining the existing object as an attribute in the class. Therefore, the `orders` class can have `delivery` as an attribute in the class. This arrangement allows you to take advantage of the existing functionality in the system, as shown in [Listing 8.23](#).

```

CLASS c1_delivery DEFINITION.
PUBLIC SECTION.
METHODS get_delivery.
ENDCLASS.
CLASS c1_delivery IMPLEMENTATION.
METHOD get_delivery.
ENDMETHOD.
ENDCLASS.
CLASS c1_orders DEFINITION.
PUBLIC SECTION.
METHODS track_order.
PRIVATE SECTION.
DATA delivery TYPE REF TO c1_delivery.
ENDCLASS.
CLASS c1_orders IMPLEMENTATION.
METHOD track_order.
CREATE OBJECT delivery.
delivery->get_delivery( ).
ENDMETHOD.
ENDCLASS.

```

Listing 8.23 Composition Example

In [Listing 8.23](#), the `c1_delivery` class is maintained as an attribute in the `c1_orders` class. The `delivery` object is instantiated in the `track` method to access the delivery information about the order.

Tip

As a rule of thumb, use an inheritance relationship between objects if they fit the *is a* relationship and use composition if the objects fit the *has a* relationship. This tip should help you make better design decisions.

8.3.5 Refactoring Assistant

When you're designing an application, sometimes you may miss defining the class components at the right level in the inheritance hierarchy. For example, you may have defined a method in the subclass that may make more sense in the superclass due to a change in the requirement.

In such a scenario, you can use the refactoring assistant tool in Class Builder to move the class components up or down the inheritance tree. This saves a lot of effort and any chance of missing steps when manually moving the components.

To use the refactoring assistant, select the menu path UTILITIES • REFACTORING • REFACTORING in Class Builder, as shown in [Figure 8.10](#).

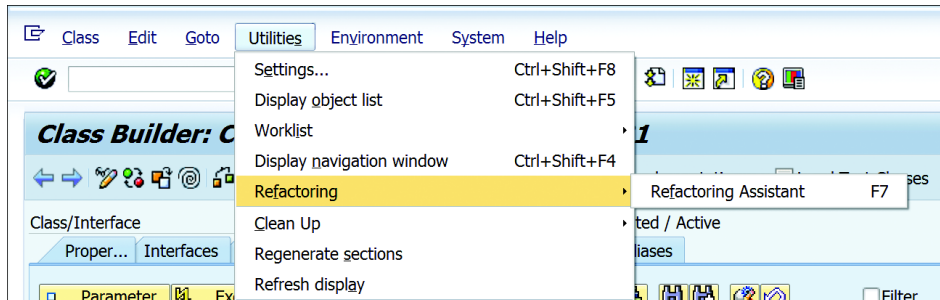


Figure 8.10 Refactoring Assistant Menu Path

In the REFACTORING ASSISTANT window (see [Figure 8.11](#)), you can drag and drop components to the right hierarchy level.

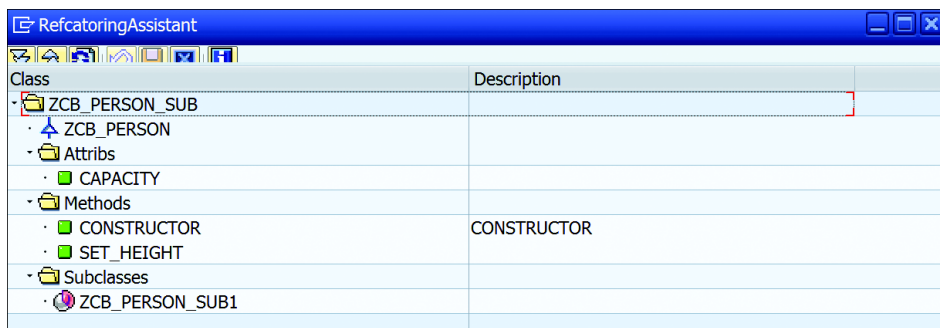


Figure 8.11 Refactoring Assistant Window

Now that we've walked through the inheritance principles, let's move on to using polymorphism in OOP.

8.4 Polymorphism

Polymorphism means *having many forms*. The inheritance concept leads to many interesting scenarios in which objects can take different forms. For example, it's possible for a subclass to respond to a method call of the superclass.

To understand polymorphism, you first need to understand static and dynamic types. We'll then look at how to use casting when a static type source reference object isn't the same as the target reference object. From there, we'll discuss how to use dynamic binding with the `CALL` method before finally looking at implementing multiple inheritances using interfaces.

8.4.1 Static and Dynamic Types

The *static type* of an object (reference variable) is the class type that is used to define an object. For example, in [Listing 8.24](#), the `oref` reference is defined by referring to the `c1_class` class. In this case, the static type of the `oref` object is the `c1_class` class as it's statically defined in the code.

```
CLASS c1_class DEFINITION.
PUBLIC SECTION.
METHODS meth.
ENDCLASS.
CLASS c1_class IMPLEMENTATION.
METHOD meth.
ENDMETHOD.
ENDCLASS.
DATA oref TYPE REF TO c1_class.
```

Listing 8.24 Code Defining Reference Object

Sometimes, you may want to assign one reference object to another reference object that doesn't share the same static type. For example, because instances of classes in an inheritance tree are interchangeable, you can assign the instance of the subclass to the instance of the superclass, as shown in [Listing 8.25](#), by using the `parent = child` statement.

```
CLASS c1_parent DEFINITION.
PUBLIC SECTION.
METHODS meth1.
METHODS meth2.
ENDCLASS.
CLASS c1_parent IMPLEMENTATION.
METHOD meth1.
WRITE 'In method1 of parent'.
ENDMETHOD.
METHOD meth2.
WRITE 'In method2 of parent'.
ENDMETHOD.
ENDCLASS.
```

```

CLASS cl_child DEFINITION INHERITING FROM cl_parent.
  PUBLIC SECTION.
    METHODS meth2 REDEFINITION.
    METHODS meth3.
ENDCLASS.
CLASS cl_child IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child'.
  ENDMETHOD.
ENDCLASS.
DATA parent TYPE REF TO cl_parent.
DATA child TYPE REF TO cl_child.
START-OF-SELECTION.
CREATE OBJECT parent.
CREATE OBJECT child.
parent->meth2( ).
parent = child.
parent->meth2( ).

```

Listing 8.25 Example of Casting

In [Listing 8.25](#), once the reference object `parent` is instantiated using the `CREATE OBJECT` statement, it sets the pointer (in memory) of the `parent` reference object to the `cl_parent` class. The pointer to which the reference object points at runtime is called the *dynamic type* of the object.

At this point, both the static type and dynamic type of the `parent` object are the same. The `parent = child` statement assigns the instance of the `child` object to the `parent` object. When one reference object is assigned to another reference object, the object itself is not assigned; only the pointer of the object is moved. In other words, after the assignment `parent = child`, the `parent` object points to the memory location of the `child` object. After this assignment, the dynamic type of the `parent` object is changed to the `cl_child` class and the static type is `cl_parent`.

At this point in the code, both the `parent` and `child` reference objects are pointing to the same `cl_child` object. Because there's no reference to the `parent` object after this assignment, the `parent` object will be removed from the memory automatically by the garbage collector. The ABAP runtime environment calls the *garbage collector* periodically to remove objects from memory that are no longer referenced by object reference variables.

In [Listing 8.25](#), when the `parent->meth2()` method is first called before the `parent = child` assignment (that is, before its dynamic type is changed), it will execute the code in the `meth2` method of the `cl_parent` superclass. However, the same statement after the `parent = child` assignment will execute the code in the `meth2` method of the `cl_child` subclass, effectively displaying polymorphism because the same object behaves differently during the runtime of the program.

8.4.2 Casting

Similar to the conversion rules applied when different types of data objects are assigned to each other, the assignment of objects of different static types are governed by certain rules. If the static type of the source reference object is not the same as the target reference object, then a special operation called a *cast* must occur. This process is known as *casting*. For the cast operation to work, the static type of the target reference object should be same or more general than the dynamic type of the source reference object.

In the example in [Listing 8.25](#), the `cl_parent` parent object is more generic than the `cl_child` child object (a superclass will always be the same as or more generic than a subclass). In other words, the parent class has implemented two methods, whereas the subclass has become more specific by implementing a third method not available in the parent class. Therefore, the parent class is more generic than the child class, and for this reason, we could assign the child object to the parent object.

There are two different types of cast operations possible: a *narrowing cast* (or *up cast*) and a *widening cast* (or *down cast*). The following subsections look at these different casting operations.

Narrowing Cast

A narrowing cast occurs when the static type of the target reference object is more generic than the static type of the source reference object. The casting operation we performed in [Listing 8.25](#) is a *narrowing cast*. It's called a narrowing cast because the parent class is more generic, and after casting the parent class can only access the components of the child class that are defined in the static type of the parent class, effectively reducing the scope of (narrowing) the components that can be accessed.

For example, say a parent class has two methods, `meth1` and `meth2`, and the child class has an additional method, `meth3`, which isn't defined in the parent class. After the casting operation, you can only access the `meth1` and `meth2` methods of the child class by using the parent object reference. If you try to access the `meth3` method, it will result in a syntax error, as shown in [Listing 8.26](#). You can, however, access the `meth3` method using the child object reference, because it still exists in the program and isn't truncated.

```
Parent = child. "Assigns child reference to parent.
Parent->meth1( )."Calls the method meth1 in child if redefined.
Parent->meth2( )."Calls the method meth2 in child if redefined.
Parent->meth3( )."Results in syntax error.
Child->meth3( )."Works as expected.
```

Listing 8.26 Narrow Cast

The narrow cast can also be performed during instantiation by adding `TYPE` to the `CREATE OBJECT` statement as shown:

```
DATA parent TYPE REF TO cl_parent.
CREATE OBJECT parent TYPE cl_child.
```

Widening Cast

A *widening cast* occurs when the static type of the target reference object is more specific than the static type of the source reference object. Because this contravenes the rule for casting, which specifies that the target object should always be more generic than the source object, the syntax checker will complain that the source object can't be converted to the target object if you try to assign the objects using the `=` operator or use the `MOVE` statement. To bypass this restriction and tell the compiler that we know what we're doing, we use the `?=` operator for a widening cast, as shown:

```
Child ?= parent.
```

Using the `?=` operator will only bypass the check statically during compile time and postpones it until runtime. It will throw a runtime error if the target object is not more generic than the source object.

For example, the code in [Listing 8.27](#) will pass the syntax checker (static check) but result in a runtime error because the child object is more specific than the parent object.

```

CLASS c1_parent DEFINITION.
  PUBLIC SECTION.
    METHODS meth1.
    METHODS meth2.
ENDCLASS.
CLASS c1_parent IMPLEMENTATION.
  METHOD meth1.
    WRITE 'In method1 of parent'.
  ENDMETHOD.
  METHOD meth2.
    WRITE 'In method2 of parent'.
  ENDMETHOD.
ENDCLASS.
CLASS c1_child DEFINITION INHERITING FROM c1_parent.
  PUBLIC SECTION.
    METHODS meth2 REDEFINITION.
    METHODS meth3.
ENDCLASS.
CLASS c1_child IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child'.
  ENDMETHOD.
ENDCLASS.
DATA parent TYPE REF TO c1_parent.
DATA child TYPE REF TO c1_child.
START-OF-SELECTION.
CREATE OBJECT parent.
CREATE OBJECT child.
Child ?= parent. "Results in runtime error

```

Listing 8.27 Widening Cast Resulting in Runtime Error

A widening cast should only be used when the dynamic type of the target object will be generic than the source object. This effectively means that a narrowing cast must be performed before a widening cast, as shown in [Listing 8.28](#). Widening casts can be confusing and dangerous, so they should be used with care.

```

CLASS c1_parent DEFINITION.
  PUBLIC SECTION.
    METHODS meth1.
    METHODS meth2.
ENDCLASS.
CLASS c1_parent IMPLEMENTATION.
  METHOD meth1.
    WRITE 'In method1 of parent'.
  ENDMETHOD.

```



```

METHOD meth2.
  WRITE 'In method2 of parent'.
ENDMETHOD.
ENDCLASS.
CLASS c1_child1 DEFINITION INHERITING FROM c1_parent.
  PUBLIC SECTION.
  METHODS meth2 REDEFINITION.
  METHODS meth3.
ENDCLASS.
CLASS c1_child1 IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child1'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child1'.
  ENDMETHOD.
ENDCLASS.
CLASS c1_child2 DEFINITION INHERITING FROM c1_child1.
  PUBLIC SECTION.
  METHODS meth2 REDEFINITION.
  METHODS meth3 REDEFINITION.
  METHODS meth4.
ENDCLASS.
CLASS c1_child2 IMPLEMENTATION.
  METHOD meth2.
    WRITE 'In method2 of child2'.
  ENDMETHOD.
  METHOD meth3.
    WRITE 'In method3 of child2'.
  ENDMETHOD.
  METHOD meth4.
    WRITE 'In method4 of child2'.
  ENDMETHOD.
ENDCLASS.
DATA parent TYPE REF TO c1_parent.
DATA child1 TYPE REF TO c1_child1.
DATA child2 TYPE REF TO c1_child2.
START-OF-SELECTION.
CREATE OBJECT parent.
CREATE OBJECT child1.
CREATE OBJECT child2.
parent = child2.
child1 ?= parent.

child1->meth2( ).

```

Listing 8.28 Widening Cast Example

In [Listing 8.28](#), three classes are defined: `cl_parent`, `cl_child1`, and `cl_child2`. `cl_child1` inherits from `cl_parent`, and `cl_child2` inherits from `cl_child1`.

A narrow cast is first performed by assigning the `child2` object to the `parent` object, and then a widening cast is performed by assigning the `parent` object to `child1`. When the `meth2` method is called using the `child1` object reference after the widening cast, it will execute the code in the `meth2` method of the `cl_child2` class.

8.4.3 Dynamic Binding with the CALL Method

When an object is passed to the method as a parameter, the system automatically binds the object dynamically. This allows us to design objects that are easily extendable.

To understand dynamic binding with the `CALL` method, let's look at the code snippet in [Listing 8.29](#). Here, we've defined `cl_student` as an abstract class with `tuition_fee` and `get_fee` as abstract methods with public visibility. A `fee_paid` attribute that can be set if the fee is paid by the student is defined in the protected section.

Two classes, `cl_commerce_student` and `cl_science_student`, inherit the `cl_student` class and maintain their own implementation of the `tuition_fee` and `get_fee` abstract methods. In the `tuition_fee` method, the private `fee_paid` attribute is set to `abap_true` if the student has paid the fee. The `get_fee` method returns the value of the private `fee_paid` attribute.

We've defined one `cl_admission` class that can be used to enroll a student in a course. This class has two methods, `set_student` and `enroll`, which can be used to complete the admission of the student in a course. Because this `cl_admission` class should be able to enroll any student from any student category (science student, commerce student, etc.), we need to design this class to import information about any student category.

The `set_student` method of the `cl_admission` class is defined with an importing parameter to import the `student` object. If we maintain the static type of this importing parameter as `cl_commerce_student`, it can only import the commerce student object and our class can only process the admission for commerce students. Similarly, if we maintain the static type of the importing parameter as `cl_science_student`, it can only import the science student object and our class can

only process the admission for science students. In order to make it work for any student category, we've maintained the importing parameter of this method as a static type of the `cl_student` abstract class, from which the `cl_science_student` and `cl_commerce_student` classes inherit.

```

CLASS cl_student DEFINITION ABSTRACT.
  PUBLIC SECTION.
    METHODS tuition_fee ABSTRACT.
    METHODS get_fee ABSTRACT RETURNING VALUE(fee_paid) TYPE boolean.
  PROTECTED SECTION.
    DATA fee_paid TYPE boolean.
ENDCLASS.
CLASS cl_commerce_student DEFINITION INHERITING FROM cl_student.
  PUBLIC SECTION.
  METHODS tuition_fee REDEFINITION.
  METHODS get_fee REDEFINITION.
ENDCLASS.
CLASS cl_commerce_student IMPLEMENTATION.
  METHOD tuition_fee.
    "logic to calculate tuition fee for commerce students goes here
    "IF fee paid.
      fee_paid = abap_true.
  ENDMETHOD.
  METHOD get_fee.
    fee_paid = me->fee_paid.
  ENDMETHOD.
ENDCLASS.
CLASS cl_science_student DEFINITION INHERITING FROM cl_student.
  PUBLIC SECTION.
  METHODS tuition_fee REDEFINITION.
  METHODS get_fee REDEFINITION.
ENDCLASS.
CLASS cl_science_student IMPLEMENTATION.
  METHOD tuition_fee.
    "logic to calculate tuition fee for science students goes here
    "IF fee paid.
      fee_paid = abap_true.
  ENDMETHOD.
  METHOD get_fee.
    fee_paid = me->fee_paid.
  ENDMETHOD.
ENDCLASS.
CLASS cl_admission DEFINITION.
  PUBLIC SECTION.
  METHODS set_student IMPORTING im_student TYPE REF TO cl_student.
  METHODS enroll.
  PRIVATE SECTION.
  DATA admit TYPE boolean.
ENDCLASS.

```

```

CLASS c1_admission IMPLEMENTATION.
  METHOD set_student.
    IF im_student->get_fee( ) EQ abap_true.
      admit = abap_true.
    ENDIF.
  ENDMETHOD.
  METHOD enroll.
    IF admit EQ abap_true.
*Perform the steps to enroll
    ENDIF.
  ENDMETHOD.
ENDCLASS.

DATA : commerce_student TYPE REF TO c1_commerce_student,
      science_student TYPE REF TO c1_science_student,
      admission TYPE REF TO c1_admission.
START-OF-SELECTION.
CREATE OBJECT: commerce_student,
              science_student,
              admission.
CALL METHOD commerce_student->tuition_fee.
CALL METHOD admission->set_student( EXPORTING im_student = commerce_
student ).
CALL METHOD admission->enroll.

CALL METHOD science_student->tuition_fee.
CALL METHOD admission->set_student( EXPORTING im_student = science_
student ).
CALL METHOD admission->enroll.

```

Listing 8.29 Code for Dynamic Call Method Binding

When the `set_student` method of the `c1_admission` class is called, it allows us to pass any student object that is inherited from `c1_student`. Within this method, we're checking if the student has paid the fee by calling the `get_fee` method of the imported object and accordingly processing the admission of the student. Here, even though the static type of the imported object is different from the static type of the importing parameter, the system automatically binds the object to the correct instance.

With this design, we can easily process admissions for any future student category via the `c1_admission` class. All the student category class needs to do is to inherit the abstract class `c1_student`.

8.4.4 Interfaces

Similar to Java, ABAP Objects only supports single inheritance and don't support multiple inheritance as in C++. *Single inheritance* means a class can have multiple subclasses but only one superclass. Many subclasses can use the same class as their superclass, but each subclass can only have a single superclass.

Modern programming languages didn't add support for multiple inheritance (having multiple superclasses) in order to avoid ambiguity. One of the common problems with multiple inheritance is called the *diamond problem*. For example, as depicted in [Figure 8.12](#), assume you have two subclasses B and C inheriting from the same superclass A and that both classes B and C have redefined a method XYZ of class A in their implementation. Now, if a new subclass D is defined as having both class B and class C as its superclasses then which superclass method will it use?

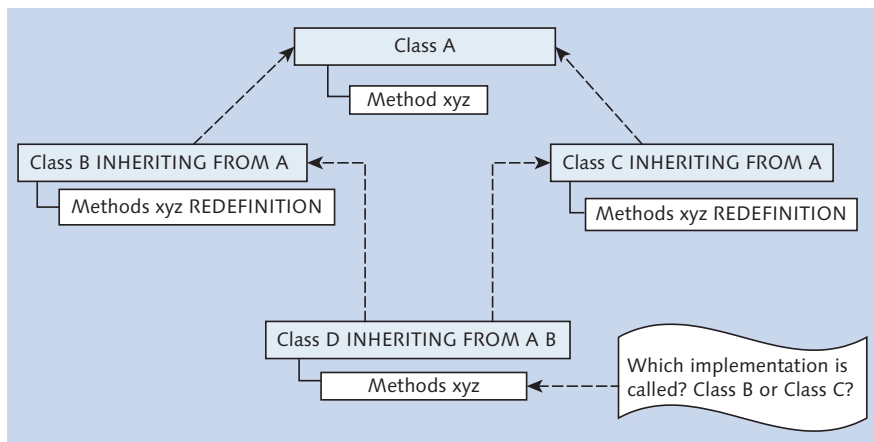


Figure 8.12 Diamond Problem with Multiple Inheritance

One way of implementing multiple inheritance is to use interfaces. Interfaces provide all the advantages of multiple inheritance while avoiding all the problems.

Interfaces are defined similarly to classes via the `INTERFACE` statement. An interface only contains a definition for which the implementation is maintained in the class that uses the interface. If you need a common definition in multiple classes, then you can create an interface and use it in multiple classes.

The syntax to define an interface is shown in [Listing 8.30](#), in which interface `if_student` is defined using the `INTERFACE` statement and ended with the `ENDINTERFACE` statement.

```
INTERFACE if_student.
  DATA course TYPE char10.
  METHODS meth1.
  EVENTS enrolled.
ENDINTERFACE.

CLASS c1_science DEFINITION.
  PUBLIC SECTION.
  INTERFACES if_student.
ENDCLASS.
CLASS c1_science IMPLEMENTATION.
  METHOD if_student~meth1.
  ENDMETHOD.
ENDCLASS.
```

Listing 8.30 Interfaces

An interface can have all the components that a class supports, such as methods, attributes, events, and so on. All of the components are in `PUBLIC SECTION` by default, and you can't use any visibility section in the interface definition. Because an interface is generally created to be used in multiple classes, it makes sense to use the interface to define the public interface of the class, with each class that uses the interface implementing its own private or protected components.

In [Listing 8.30](#), a `c1_science` class implements the `if_student` interface via the `INTERFACES` statement. In the implementation of the `c1_science` class, the `meth1` method defined in the `if_student` interface is implemented. Because the `meth1` method is defined in the interface and not in the class itself, the method is accessed using the `interface_name~component_name` syntax. Here, the tilde symbol (`~`) is the interface component selector that should be used to access the component of an interface.

Unlike classes, interfaces do not have any implementation section. Interfaces can be defined only in the global declaration area of a program and can't be defined within a class or a procedure or an event block.

Sometimes, the interface name can be too long. To avoid addressing the interface component using the long interface name, we can use aliases to make it easy while coding, as shown in [Listing 8.31](#).

```

INTERFACE if_student.
  DATA course TYPE char10.
  METHODS meth1.
  EVENTS enrolled.
ENDINTERFACE.
CLASS c1_science DEFINITION.
  PUBLIC SECTION.
  INTERFACES if_student.
  ALIASES m1 FOR if_student~meth1.
ENDCLASS.
CLASS c1_science IMPLEMENTATION.
  METHOD m1.
  ENDMETHOD.
ENDCLASS.
DATA science TYPE REF TO c1_science.
START-OF-SELECTION.
CREATE OBJECT science.
science->m1( ).

```

Listing 8.31 Using Aliases

Interfaces can be nested by using the `INTERFACES` statement to include an interface within an interface. In [Listing 8.32](#), the `if_student` interface is nested inside `if_college`. An alias is defined for the `meth1` method of `if_student` within `if_college`. The `c1_science` class implements the `if_college` interface. The `meth1` method of the `if_student` interface is accessed within the class using the alias defined for it in `if_college`.

```

INTERFACE if_student.
  DATA course TYPE char10.
  METHODS meth1.
  EVENTS enrolled.
ENDINTERFACE.
INTERFACE if_college.
  INTERFACES if_student.
  ALIASES m1 FOR if_student~meth1.
  METHODS meth1.
ENDINTERFACE.
CLASS c1_science DEFINITION.
  PUBLIC SECTION.
  INTERFACES if_college.
  ALIASES m1 FOR if_college~m1.
  ALIASES m2 FOR if_college~meth1.
ENDCLASS.
CLASS c1_science IMPLEMENTATION.
  METHOD m1.
  ENDMETHOD.

```

```

METHOD m2.
ENDMETHOD.
ENDCLASS.
DATA science TYPE REF TO cl_science.
START-OF-SELECTION.
CREATE OBJECT science.
science->m1( ).

```

Listing 8.32 Nesting Interfaces

The process of defining a global interface in Class Builder is similar to creating a class. Follow these steps to create an interface in Class Builder:

1. On the initial Class Builder screen (Transaction SE24; see [Figure 8.13](#)), enter the name of the interface and click the CREATE button. If the name starts with *ZIF*, Class Builder will automatically create an interface; otherwise, you'll see a dialog box (see [Figure 8.14](#)) to choose whether to create a class or interface.

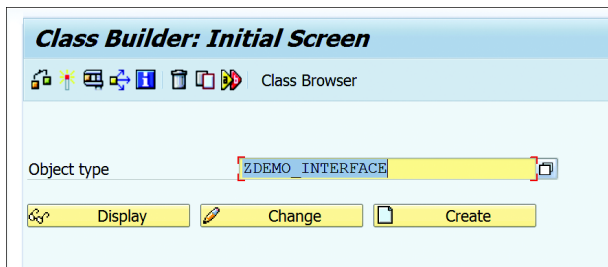


Figure 8.13 Class Builder: Initial Screen

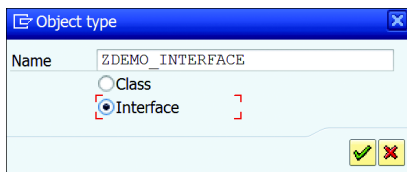


Figure 8.14 Object Type Selection Dialog Box

2. On the CLASS BUILDER: CHANGE INTERFACE... screen (see [Figure 8.15](#)), maintain the components of the interface and click the ACTIVATE button in the application toolbar to activate.

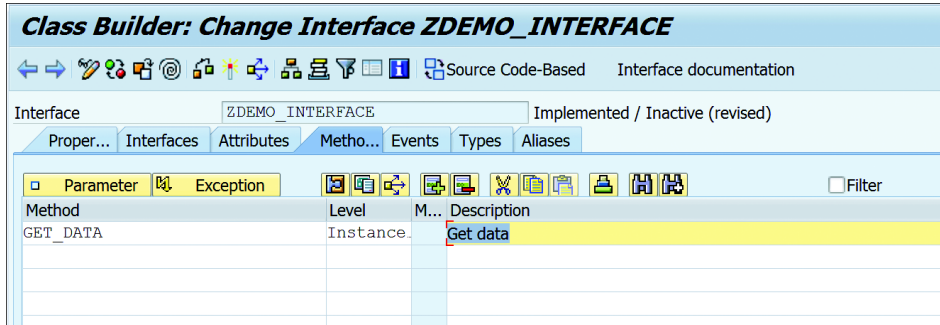


Figure 8.15 Class Builder: Change Interface Screen

3. You can include the interface in a global class under the INTERFACES tab, as shown in [Figure 8.16](#).

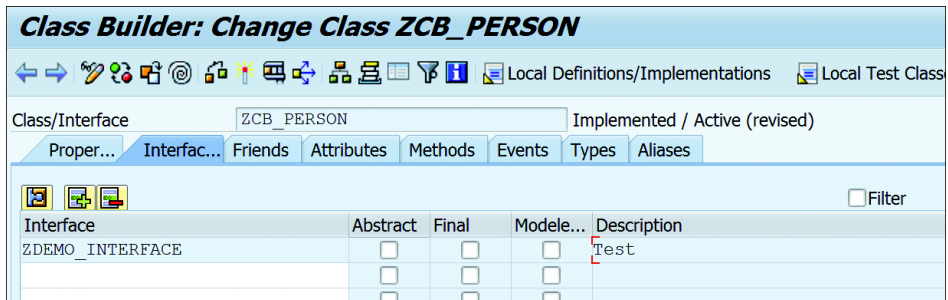


Figure 8.16 Including Interface in Global Class

4. The alias for an interface component can be maintained under the ALIASES tab, as shown in [Figure 8.17](#). The name of the alias is maintained under the ALIAS column, and the visibility for the alias can be set under the VISIBLE column.

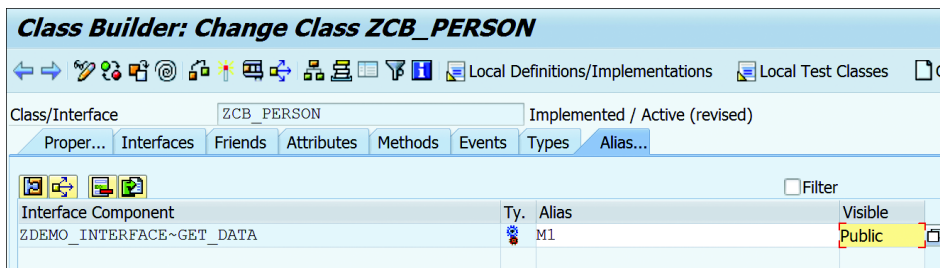


Figure 8.17 Aliases for Interface Component

In this section, we discussed the concept of polymorphism in OOP. In the next section, we'll take a bit of a departure from object-oriented concepts and look at how to work with XML.

8.5 Working with XML

Extensible Markup Language (XML) is a meta markup language that is used to define structured documents that can be easily exchanged between heterogeneous systems. There are many ways to exchange data between systems, but with the growth of web services, XML has gained popularity among developers. The beauty of XML lies in its flexibility and simplicity.

XML defines a standard that can be used to define the format of documents. Using XML, we can structure and organize various kind of data. A markup language (e.g., HTML) consists of *tags*, which are predefined. Tags allow you to format and organize the data. Tags are maintained between less than (<) and greater than (>) symbols. For example, in an HTML document, we can open the paragraph tag with <p> and close it with </p>. The content maintained between the tags is interpreted accordingly. Here, the content maintained between <p> and </p> is formatted as a paragraph; the content maintained between <h1> and </h1> is interpreted as a header.

However, XML is a *meta* markup language, which defines a markup language. It's designed to be flexible and easy to understand for both humans and machines. In XML, there are no predefined tags; any tag can be used to describe content.

Even though an in-depth understanding of XML documents and their processing is beyond the scope of this book, in this section we'll discuss the basic XML syntax and look at the iXML library provided by SAP to create XML documents in ABAP. Understanding XML is helpful when exposing or consuming data with external applications. This section should help you appreciate how OOP aids in designing libraries that are easy for developers to use in their programs.

8.5.1 XML Overview

XML files provide great flexibility to share structured documents. Because there are no predefined tags, we can use tags that suit our domain and that are self-

explanatory. Of course, developing a standard for document markup makes it easy for third-party vendors to develop software that can process XML files.

For example, say we have a website that can be extended by installing custom extensions. The extensions are of two types: components and plugins (i.e., the website can be extended by developing a component or a plugin). We've developed a component and want to upload the files of the custom component as a package to the webserver. To provide the information about the component to the installer script, we can supply an XML file with the package that contains information about the file structure of the package, as shown in [Listing 8.33](#). The installer can read this XML file and upload the files in the package to the designated folders in the web server.

```
<?xml version="1.0" encoding="utf-8"?>
<!--This is comment-->
<extension type="component">
  <files folder="site">
    <filename>index.html</filename>
    <filename>site.php</filename>
  </files>
  <media folder="media">
    <folder>css</folder>
    <folder>images</folder>
    <folder>js</folder>
  </media>
</extension>
```

Listing 8.33 Sample XML Document

As shown in [Listing 8.33](#), XML documents are organized into a series of elements. The first line in the document specifies the XML version and the encoding used. The syntax for this statement is `<?xml version="1.0" encoding=""?>` and it's optional.

The basic syntax to define an element in XML is as follows:

```
<element_name attribute_name=attribute_value>
  <!-- Element Content -->
</element_name>
```

In the example XML file provided, for the `<extension type="component">` tag, the element name is `extension`, the attribute name is `type`, and the attribute value is `component`. This tells the upload script that we're trying to upload an extension that's a component. The tags `<files>` and `<media>` are the elements under the

parent node `<extension>`. The tags under `<files>` and `<media>` are the respective elements' content, which specifies the folders and files that we are uploading. The document is closed with the closing tag `</extension>`. Comments can be maintained between `<!--` and `-->`.

XML markup is case-sensitive; for example, the element name `<ELEMENT>` is different than `<element>`.

8.5.2 XML Processing Concepts

SAP NetWeaver AS ABAP provides an iXML library that can be used to process XML files. This library implements various interfaces that allow us to work with XML files by calling various methods.

[Listing 8.34](#) shows the code to create an XML file using the iXML library. The XML file we're creating contains the data from [Listing 8.33](#). In [Listing 8.34](#), we're defining reference objects for the XML document from the iXML library. The reference object for the XML document is referenced to the `if_ixml_document` interface, and a reference object is defined for each element in the XML file by referencing `if_ixml_element`. The XML file is generated by calling the respective methods from the library, as shown in [Listing 8.34](#).

```
REPORT ZDEMO_XML.
*Declarations to create XML document
DATA: lr_ixml TYPE REF TO if_ixml. "Reference for iXML object
      "Reference for XML document
DATA: lr_document TYPE REF TO if_ixml_document.
      ."Reference for "extension" element in document
DATA: lr_extension TYPE REF TO if_ixml_element

      ."Reference for "files" element in document
DATA: lr_files TYPE REF TO if_ixml_element
      ."Reference for "media" element in document
DATA: lr_media TYPE REF TO if_ixml_element
      ."Reference to set encoding
DATA: lr_encoding TYPE REF TO if_ixml_encoding

*Declarations to create output stream and render the file to
*application server directory
DATA: lr_streamfactory TYPE REF TO if_ixml_stream_factory,
      lr_ostream TYPE REF TO if_ixml_ostream,
      lr_renderer TYPE REF TO if_ixml_renderer.
DATA file_path TYPE string VALUE 'D:\USR\SAP\PUT\MANIFEST.XML'.
```

```

* Create iXML object
lr_ixml = cl_ixml=>create( ).
*Create Document
lr_document = lr_ixml->create_document( ).
*Create encoding
lr_encoding = lr_ixml->create_encoding( BYTE_ORDER = 0 CHARACTER_SET =
'UTF-8').
*Set encoding
lr_document->set_encoding( lr_encoding ).
*Create element "extension" as root
lr_extension = lr_document->create_simple_element(
name = 'extension'
parent = lr_document ).
*Set attribute for the "extension" element
lr_extension->set_attribute( name = 'Type'
VALUE = 'Component' ).
*Create "files" element with "extension" element as parent
lr_files = lr_document->create_simple_element(
name = 'files'
parent = lr_extension ).
*Set attribute for "files" element
lr_files->set_attribute( name = 'Folder'
VALUE = 'site' ).
*Create element content
lr_document->create_simple_element( name = 'filename'
parent = lr_files
VALUE = 'index.html' ).
lr_document->create_simple_element( name = 'filename'
parent = lr_files
VALUE = 'site.php' ).

*Create "media" element with "extension" element as parent
lr_media = lr_document->create_simple_element(
name = 'media'
parent = lr_extension ).
**Set attribute for "media" element
lr_media->set_attribute( name = 'Folder'
VALUE = 'media' ).

*Create element content
lr_document->create_simple_element( name = 'folder'
parent = lr_media
VALUE = 'css' ).
lr_document->create_simple_element( name = 'folder'
parent = lr_media
VALUE = 'images' ).
lr_document->create_simple_element( name = 'folder'
parent = lr_media
VALUE = 'js' ).

```

```

* Create stream factory
lr_streamfactory = lr_ixml->create_stream_factory( ).
* Create output stream
lr_ostream = lr_streamfactory->create_ostream_uri( system_id = file_
path ).
* Create renderer
lr_renderer = lr_ixml->create_renderer( ostream = lr_ostream
document = lr_document ).
* Set pretty print
lr_ostream->set_pretty_print( abap_true ).
* Renders the attached document into output stream
lr_renderer->render( ).

```

Listing 8.34 Using iXML library to Create XML File

The code in [Listing 8.34](#) saves the XML file in the application server directory, as shown in [Figure 8.18](#).

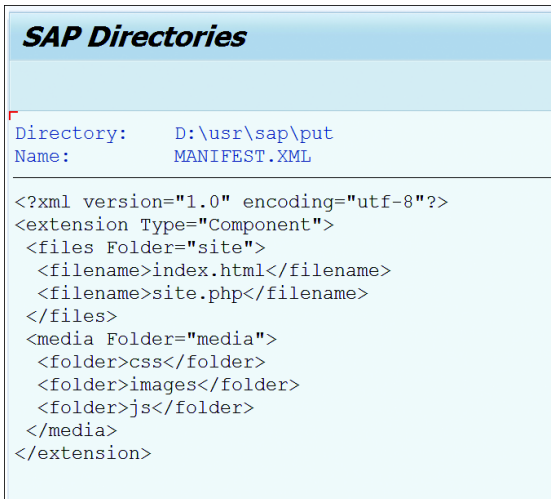


Figure 8.18 XML File

8.6 Summary

In this chapter, we explored various OOP concepts. The four pillars of OOP are encapsulation, inheritance, polymorphism, and abstraction. We discussed all of these concepts and looked at the various ways they're implemented in ABAP.

By now, you should have a good overview of objects and should feel comfortable working with ABAP Objects. Designing software using ABAP Objects takes some experience, but this chapter should serve as a starting point for you to start looking at designing applications by using OOP concepts.

In the next chapter, we'll use some of the concepts from this chapter to shed light on how to handle error situations in applications.

Exception handling is what happens when a system throws an error. In this chapter, we'll discuss how to handle exceptions for both procedural and object-oriented programming approaches.

9 Exception Handling

During the runtime of a program, a program may encounter error situations. Sometimes, errors are introduced by programmers accidentally, but most errors are due to incorrect handling of the application. Error situations may be anticipated during the development of an application, but there is simply no way to avoid every type of error.

In this chapter, we'll discuss how to handle exceptions and look at the various ways in which errors can be captured. In [Section 9.1](#), we provide an overview of what exceptions are before turning our attention to how exceptions are handled in procedures in [Section 9.2](#). Then, we look at how exception handling can be separated from the main program logic by using class-based exceptions and OOP in [Section 9.3](#). Finally, in [Section 9.4](#), we'll look at using messages within exception classes.

9.1 Exceptions Overview

In [Chapter 6, Section 6.2](#), you learned how to show various messages to the user when an error situation occurs while the user is executing a program. However, procedures are always called in programs and never executed by the user directly. Showing a message in a procedure doesn't really make sense, because messages are meant to be displayed to the user. With procedures, the recommended way to handle an error situation is to raise an exception and let the calling program decide what to do with the exception.

For example, say we've developed a function module to take the material number as an importing parameter and export various details of the material. If the material number imported into the function module is invalid, then instead of showing a message that the material number is invalid, we can raise an exception in the function module, which the calling program can process per its functional requirement. That is, the calling program can either show an error message or perform other steps. In addition, because the procedures are designed to be reusable, by raising exceptions in a procedure different programs can handle them differently per their particular requirements.

9.2 Procedural Exception Handling

Prior to class-based exceptions, only function modules and methods were allowed to raise self-defined treatable exceptions, and exception handling logic was intermixed with actual program logic.

In this section, we'll look at using function modules and methods for maintaining exceptions. We'll then look at how to maintain exceptions for local classes.

9.2.1 Maintaining Exceptions Using Function Modules

Exceptions are statically maintained in a function module with explanatory text explaining the exception situation. When this exception is raised, the SY-SUBRC system field is set with the exception number, which the calling program can check to implement custom error handling logic.

In [Listing 9.1](#), the ZEXCEPTION_EXAMPLE function module has two exceptions maintained: NO_MATERIAL_DATA and NO_PLANT_DATA. When this function module is called, a suitable exception is raised in the function module if an error situation occurs. The calling program checks the value of SY-SUBRC to identify the exception and maintain suitable error handling logic in the program.

```
CALL FUNCTION 'ZEXCEPTION_EXAMPLE'
EXPORTING
  Im_matnr = v_matnr
IMPORTING
  t_marc   = it_marc
EXCEPTIONS
```

```

NO_MATERIAL_DATA = 1
NO_PLANT_DATA    = 2
OTHERS           = 3.
IF sy-subrc <> 0.
CASE sy-subrc.
WHEN 1.
"NO_MATERIAL_DATA exception is raised by Function Module
"Suitable exception handling code goes here
WHEN 2.
"NO_PLANT_DATA exception is raised by Function Module
"Suitable exception handling code goes here
WHEN 3.
"Unknown exception is raised by Function Module
ENDCASE.
ENDIF.

```

Listing 9.1 Exception Handling with Function Modules

These self-defined exceptions are manually maintained in function modules and methods and are raised using the `RAISE` statement within a function module or method.

Figure 9.1 shows the exception maintained under the `EXCEPTIONS` tab in a function module.

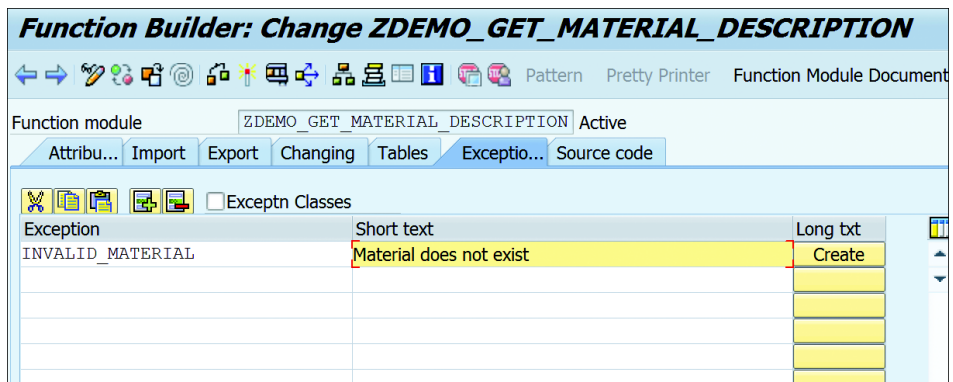


Figure 9.1 Maintaining Exceptions

Figure 9.2 shows the code to raise the exception using the `RAISE` statement.

```

1 FUNCTION ZDEMO_GET_MATERIAL_DESCRIPTION.
2 -----
3 *** Local Interface:
4 ***
5 *** IMPORTING
6 *** REFERENCE (IM_MATNR) TYPE MATNR
7 *** REFERENCE (IM_SPRAS) TYPE SPRAS
8 *** EXPORTING
9 *** REFERENCE (EX_MAKTX) TYPE MAKTX
10 *** EXCEPTIONS
11 *** INVALID_MATERIAL
12 ***
13 -----
14 SELECT SINGLE maktx FROM makt INTO ex_maktx WHERE matnr EQ im_matnr AND spras EQ im_spras.
15 IF sy-subrc IS NOT INITIAL.
16 RAISE invalid_material.
17 ENDIF.
18
19
20 ENDFUNCTION.

```

Figure 9.2 Raising Exception

When the first exception is raised, the function module is exited, and control is passed back to the calling program without executing further code in the function module. If the calling program doesn't catch the exception, then a runtime error (short dump) is generated. For this reason, it's always recommended to use the EXCEPTIONS keyword in the function module call (CALL FUNCTION) to catch exceptions (see [Figure 9.3](#)), even if you don't plan to implement any custom logic for the exception.

```

CALL FUNCTION 'ZDEMO_GET_MATERIAL_DESCRIPTION'
  EXPORTING
    im_matnr      = v_matnr
    im_spras     = sy-langu
  IMPORTING
    EX_MAKTX     = v_maktx
  EXCEPTIONS
    INVALID_MATERIAL = 1
    OTHERS        = 2
.

IF sy-subrc <> 0.
  * Implement suitable error handling here
ENDIF.

```

Figure 9.3 Catching Exception in Calling Program

9.2.2 Maintaining Exceptions Using Methods

Similar to function modules, exceptions can be maintained in methods via Class Builder by keeping the cursor on the method and selecting the EXCEPTION button, as shown in [Figure 9.4](#).

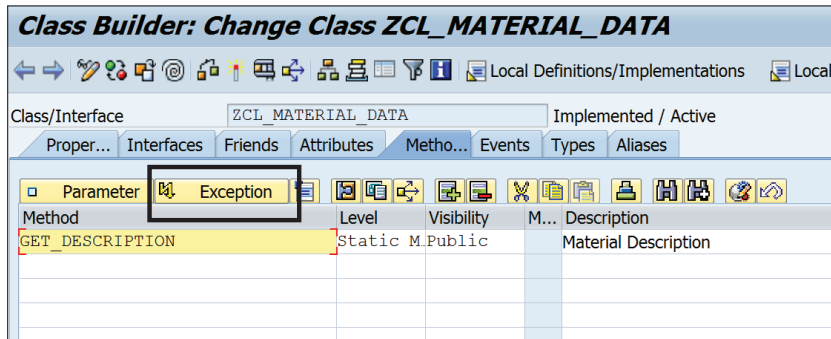


Figure 9.4 Exception Button

Under the EXCEPTION column in the METHOD tab ([Figure 9.5](#)), give a name for the exception and maintain a description to explain the exception so that developers calling the method can understand why that exception is raised.

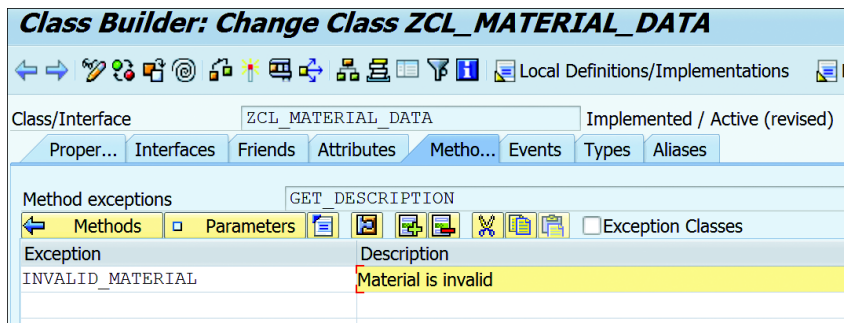


Figure 9.5 Exceptions Window

As shown in [Figure 9.6](#), an exception can be raised in the method using the RAISE statement.

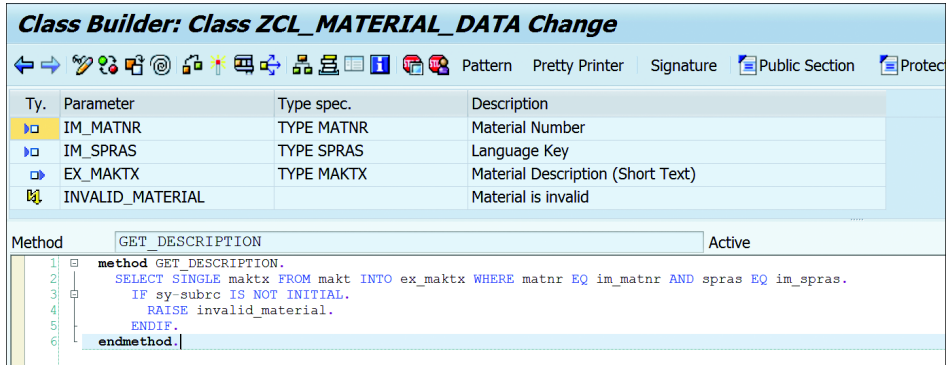


Figure 9.6 Raising Exception for Class ZCL_MATERIAL_DATA

Note that exceptions are specific to a procedure in the sense that if an exception is maintained for a function module, it can be raised only within that function module and can't be raised from other function modules of the function group. Similarly, an exception maintained for a method can be raised only from within that method and can't be raised from other methods of the class.

9.2.3 Maintaining Exceptions for Local Classes

For local classes, exceptions can be defined using the `EXCEPTIONS` statement in the definition part of the class, as shown in [Listing 9.2](#). The exceptions that are defined in the definition part of the class can be raised by the methods implemented in the implementation part of the same class.

In [Listing 9.2](#), we defined the `INVALID_MATERIAL` exception in the definition part of the `LCL_MATERIAL_DATA` class. Exceptions are defined using the `EXCEPTIONS` keyword. It's a good practice to maintain a comment with a short description for the exception.

```

CLASS lcl_material_data DEFINITION
  PUBLIC
  PUBLIC SECTION.
    CLASS-METHODS get_description
    IMPORTING
      im_matnr TYPE matnr
      im_spras TYPE spras
    EXPORTING
      ex_maktx TYPE maktx

```

EXCEPTIONS

```

        invalid_material. "Raised when material is invalid
ENDCLASS.

```

Listing 9.2 Maintaining Exception in Local Class

Having gone through how to handle procedural-based exceptions, next we'll shift our focus to class-based exception handling.

9.3 Class-Based Exception Handling

Class-based exceptions (object-oriented) help separate the exception handling logic from the main programming logic by handling the exceptions in a separate block. In the previous section on procedural methods for handling exceptions, you saw that beyond checking the system field `SY-SUBRC` to identify the exception, there isn't much information available about the exception.

In addition, because self-defined exceptions are supported only in function modules and methods, there's no proper way to handle error situations in regular programs. For example, if there's an error situation within a subroutine in which self-defined exceptions aren't supported, how do we handle it?

One way is to pass the error structure or some kind of variable in the parameter interface and check against it after the execution of subroutine, as shown in [Listing 9.3](#).

```

PERFORM get_data USING v_matnr
                    CHANGING it_makt
                            retcode.

IF retcode IS INITIAL.
*Program logic
ELSE.
*Error handling logic
ENDIF.
FORM get_data USING p_matnr
                CHANGING pt_makt
                        p_retcode.
SELECT * FROM makt INTO TABLE pt_makt WHERE matnr EQ p_matnr.
IF sy-subrc IS NOT INITIAL.
p_retcode = 1.
ENDIF.
ENDFORM.

```

Listing 9.3 Error Handling in Subroutine

As you can see, the code in [Listing 9.3](#) uses the `RETCODE` parameter to capture the error situation and build the program logic accordingly to handle the exception. However, this isn't an elegant way to handle exceptions, and the interface can soon become cluttered if many exception situations need to be handled. This also creates problems when forwarding the exception if the exception needs to be handled in another processing block. For example, most BAPIs implement an error structure in the interface to capture the error. In this error structure, most of the information about the error is captured, such as error type, error ID, message, and so on.

Now, we want to pass these error details to another function module called within the BAPI, one in which the exception can be handled. To do this, we need to ensure that the BAPI and the function module called within the BAPI have similar error structures; otherwise, we need to manually convert the data and assign it to the new error structure of the function module.

Contrary to what many developers assume, class-based exceptions are not limited to ABAP Objects. They can be used in any processing block. Class-based exceptions are handled using a special block called the `TRY` block. Using the `CATCH` and `CLEANUP` statements, we can handle exceptions within a `TRY` block. The `CATCH` statement can catch multiple exceptions within a block, and the `CLEANUP` statement allows you to perform cleanup tasks while exiting the exception. For example, if you open a file for output and come across an exception while transferring one of the records, then the cleanup block allows you to close the file.

In this section, we'll discuss the use of class-based exception handling, including steps for raising exceptions, using catchable and non-catchable exceptions, and how to define exception classes both globally and locally.

9.3.1 Raising Exceptions

Exceptions are raised in a couple of situations—either programmatically using the `RAISE EXCEPTION` statement or automatically by the ABAP runtime environment. For example, if we execute the code in [Listing 9.4](#), it will result in a runtime error, as shown in [Figure 9.7](#).

```
DATA : v_unit_value TYPE I,  
      v_total_value TYPE I VALUE 20,
```



```
v_quantity TYPE I.
v_unit_value = v_total_value / v_quantity.
```

Listing 9.4 Illegal Arithmetic Operation

The screenshot shows an SAP error message dialog box with the following content:

Category	ABAP Programming Error
Runtime Errors	COMPUTE_INT_ZERODIVIDE
Except.	CX_SY_ZERODIVIDE
ABAP Program	ZDEMO_ERROR
Application Component	Not assigned
Date and Time	12.04.2016 14:51:10

Short Text
Division by 0 (type I)

What happened?
Error in the ABAP Application Program

The current ABAP program "ZDEMO_ERROR" had to be terminated because it has come across a statement that unfortunately cannot be executed.

Error analysis
An exception has occurred which is explained in more detail below. The exception, which is assigned to class 'CX_SY_ZERODIVIDE', was not caught and therefore caused a runtime error. The reason for the exception is:
In the current program "ZDEMO_ERROR", an arithmetic operation ('DIVIDE', '/', 'DIV', or 'MOD') attempted to use operands of type I to divide by 0.

An arrow points from the text "Exception class" to the boxed text 'CX_SY_ZERODIVIDE' in the error analysis section.

Figure 9.7 Exception Raised by Runtime System

Figure 9.7 highlights the exception class that is raised by the system when the operand is divided by zero. We can avoid the short dump by catching the exception in our program and exiting the exception situation.

Listing 9.5 shows the syntax to implement the TRY block to handle exceptions.

```
DATA : v_unit_value TYPE I,
       v_total_value TYPE I VALUE 20,
       v_quantity TYPE I.
DATA r_ex TYPE REF TO cx_sy_zerodivide.
TRY. "Begin of TRY block
v_unit_value = v_total_value / v_quantity.
* Other programming logic goes here
CATCH cx_sy_zerodivide INTO r_ex.
WRITE : / 'Error Short Text:', r_ex->get_text( ).
WRITE : / 'Error long Text:', r_ex->get_longtext( ).
CLEANUP.
```

```
* Any cleanup logic goes here
ENDTRY. "End of try block
```

Listing 9.5 Catching Exceptions

The code in [Listing 9.5](#) results in the output shown in [Figure 9.8](#). Here, the system didn't generate a short dump because we caught the exception in the program and handled the exception situation programmatically. A good programming practice is to avoid such exception situations by making sure a check is performed on the denominator before using it in division (e.g., IF v_quantity IS NOT INITIAL).

<i>Demo program to handle exception</i>
Demo program to handle exception
Error Short Text: Division by zero. Error long Text: You tried to divide by zero during the operation '/'.

Figure 9.8 Output for Code in Listing 9.5

9.3.2 Catchable and Non-Catchable Exceptions

There are two types of exceptions generated by the system: *catchable* and *non-catchable*. For *catchable exceptions*, we can use the TRY control structure to catch the exception in the program, *non-catchable exceptions* result in a runtime error.

We can find information about various possible exceptions raised by the system by checking the keyword documentation of any particular keyword and implement suitable exception handling in our program. For example, [Figure 9.9](#) shows the possible catchable and non-catchable exceptions raised by the system for ASSIGN statement. All the system exception classes start with CX_SY_.

Going back to the code in [Listing 9.5](#), we have identified the CX_SY_ZERODIVIDE catchable exception, which is raised by the system when it encounters a value being divided by zero. The only situation in which a system exception is not raised under such circumstances is when the numerator is also zero. Here, CX_SY_ZERODIVIDE is a global class with certain components, as shown in [Figure 9.10](#).

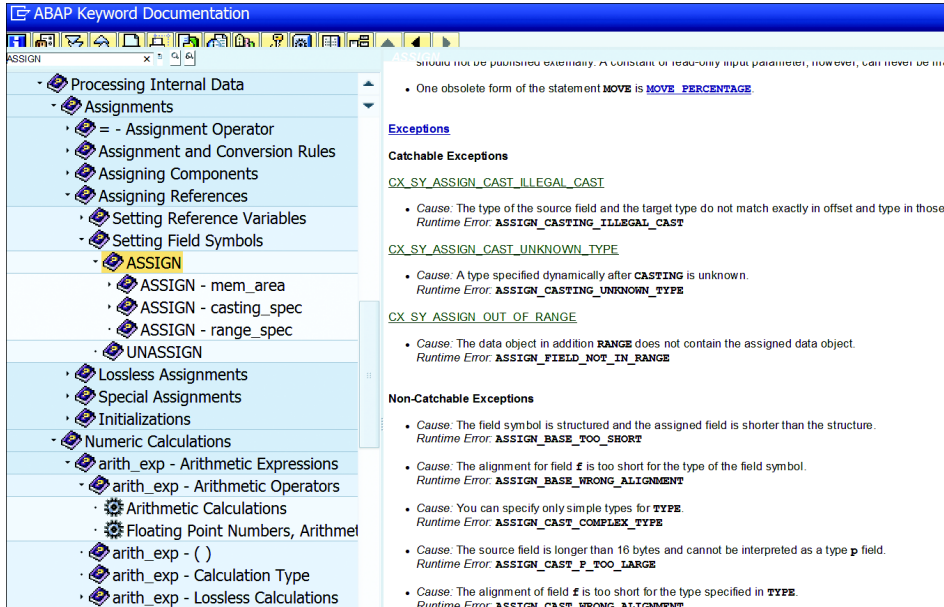


Figure 9.9 Possible Exceptions for ASSIGN Statement

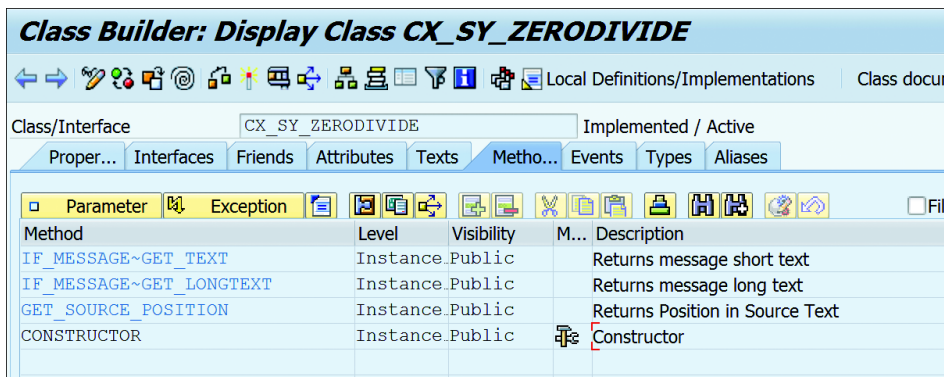


Figure 9.10 CX_SY_ZERODIVIDE Class

As you can see in [Figure 9.10](#), the `CX_SY_ZERODIVIDE` class implements an `IF_MESSAGE` interface that has two methods: `GET_TEXT` and `GET_LONGTEXT`. Aliases for these methods are maintained in the `ALIASES` tab, as shown in [Figure 9.11](#).

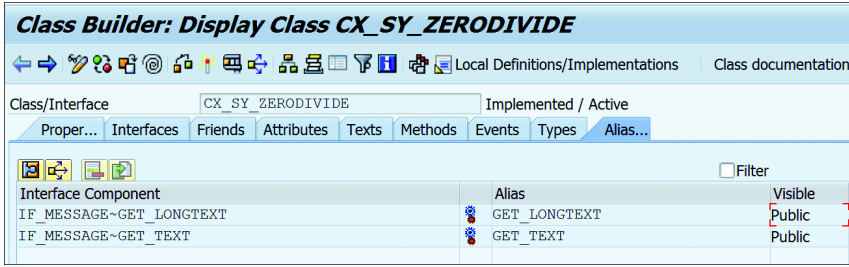


Figure 9.11 Aliases for Interface Methods

In Listing 9.5, we defined a `R_EX` reference object referring to the `CX_SY_ZERODIVIDE` static type. Within the `TRY` control structure, the statement `CATCH cx_sy_zerodivide INTO r_ex` instantiates the `R_EX` object. We accessed the short text and long text of the error by calling the methods `GET_TEXT` and `GET_LONGTEXT`, respectively.

Optionally, we also maintained the `CLEANUP` block, in which suitable exit code to gracefully exit the procedure can be maintained. For example, the `CLEANUP` block can be used to close any open files or rollback any database updates.

If you look carefully at the class `CX_SY_ZERODIVIDE`, you'll see that it inherits from a `CX_SY_ARITHMETIC_ERROR` superclass. The `CX_SY_ARITHMETIC_ERROR` class in turn inherits from a `CX_DYNAMIC_CHECK` superclass, which inherits from `CX_ROOT` (see Figure 9.12).

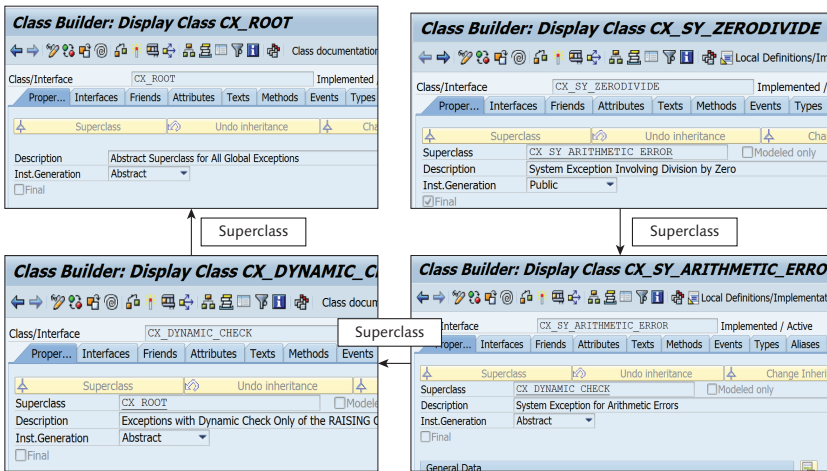


Figure 9.12 Inheritance Tree for Exception Class

The base class for all exception classes is `CX_ROOT`, which implements the `IF_MESSAGE` and `IF_SERIALIZABLE_OBJECT` interfaces. The `IF_MESSAGE` interface contains the `GET_TEXT` and `GET_LONGTEXT` methods, which allow you to extract the error message. The `IF_SERIALIZABLE_OBJECT` interface is used for *object serialization*, a process in which the object is converted into a serialization format (e.g., XML, CSV) that can be transmitted to create a clone of the object in the same system or another system. The `IF_SERIALIZABLE_OBJECT` interface should be implemented if the exception object needs to be serialized. The `CX_ROOT` class also implements another method, `GET_SOURCE_position`, which returns the position in the source code that caused the error.

When catching class-based exceptions, the exception information is captured by an object that is an instance of a class that inherits from one of three global abstract classes: `CX_STATIC_CHECK`, `CX_DYNAMIC_CHECK`, or `CX_NO_CHECK`. These three global abstract classes are in turn subclasses of `CX_ROOT`. For example, in [Listing 9.5](#), we can define the `R_ER` reference object as an instance of the `CX_ROOT` class instead of the `CX_SY_ZERODIVIDE` class and still be able to catch the exception as shown in [Listing 9.6](#). However, doing so will make the error logic generic and miss any specific exception handling details maintained in the specific exception class.

```
DATA r_ex TYPE REF TO cx_root.
TRY.
v_unit_value = v_total_value / v_quantity.
CATCH cx_sy_zerodivide INTO r_ex.
  WRITE : / 'Error Short Text:', r_ex->get_text( ).
  WRITE : / 'Error long Text:', r_ex->get_longtext( ).
ENDTRY.
```

Listing 9.6 Using `CX_ROOT` Instance

As noted, when defining a custom exception class, it should inherit from one of three classes:

► **`CX_STATIC_CHECK`**

If an exception class that's a subclass of `CX_STATIC_CHECK` is raised in a function module or method, it should be either defined in the signature (parameter interface) or caught in the procedure within a `TRY` block. Otherwise, a warning message is shown by the syntax checker.

► **CX_DYNAMIC_CHECK**

Similar to `CX_STATIC_CHECK`, if an exception class that's a subclass of `CX_DYNAMIC_CHECK` is raised in a function module or method, it should either be defined in the signature (parameter interface) or caught in the procedure within a `TRY` block. Otherwise, the compiler will not give a static warning message statically but will produce a runtime error.

► **CX_NO_CHECK**

The exception class that is a subclass of `CX_NO_CHECK` can't be defined in the signature of a function module or method. It's implicitly declared by default and always propagated to the caller.

Because `CX_STATIC_CHECK` is checked by the compiler during design time, it's recommended to use this class to define exception classes. Most system exceptions are derived from `CX_DYNAMIC_CHECK`.

SAP has implemented many exception classes that can be used out of the box. The keyword documentation lists all the available system exception classes, as shown in [Figure 9.13](#). Keyword documentation can be accessed using Transaction ABAP-DOCU.

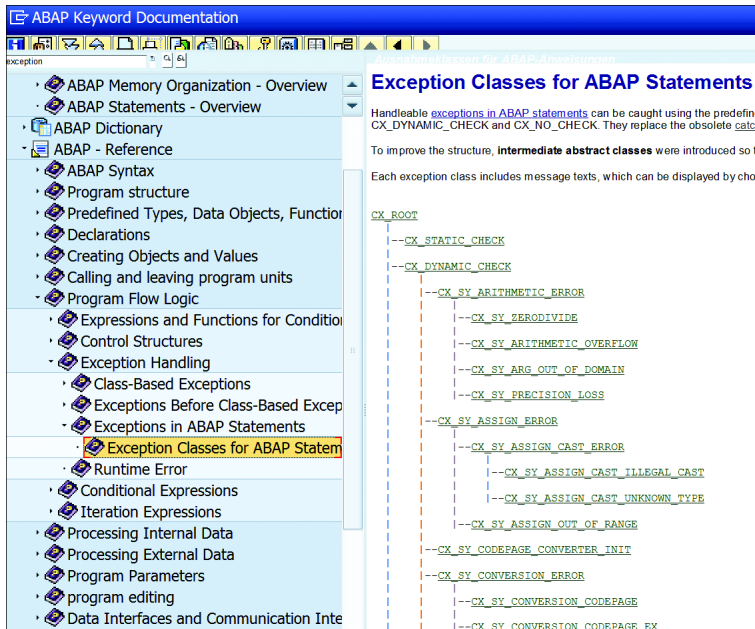


Figure 9.13 System Exception Classes in Keyword Documentation

However, if required, we can create custom exception classes either locally in the program or globally in Class Builder. We'll look at how to implement both options next.

9.3.3 Defining Exception Classes Globally

Most often, we define exception classes globally so that they can be used in multiple programs. The steps to create an exception class in Class Builder are similar to those to create any regular class, except that the exception class must inherit `CX_STATIC_CHECK`, `CX_DYNAMIC_CHECK`, or `CX_NO_CHECK`, or any class that inherits these classes.

In addition, exception classes should follow the naming convention of `CX_` for the SAP namespace and `ZCX_` for customer namespace. The `EXCEPTION CLASS` checkbox is also checked for the exception class.

Follow these steps to define an exception class in Class Builder:

1. Enter the name of the exception class on the Class Builder initial screen (Transaction SE24) and click the `CREATE` button. The name of the exception class should start with `ZCX_`, as shown in [Figure 9.14](#).

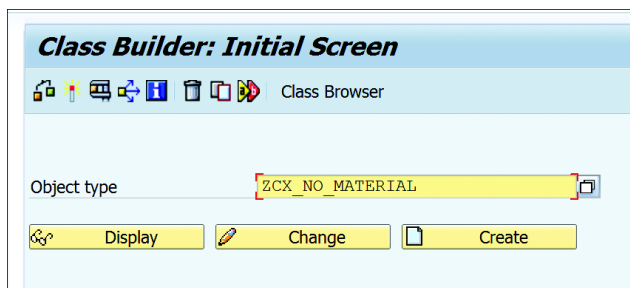


Figure 9.14 Class Builder: Initial Screen

2. The system will present the properties dialog box. Because you used the naming convention `ZCX_`, the system automatically assigns `CX_STATIC_CHECK` under `SUPERCLASS` and selects the `EXCEPTION CLASS` radio button under `CLASS TYPE`, as shown in [Figure 9.15](#). Optionally, you can change the superclass to `CX_DYNAMIC_CHECK` or `CX_NO_CHECK` or any class that inherits these abstract classes. Maintain a description for the class and set the preferred instantiability for the class, as shown in [Figure 9.15](#).

The WITH MESSAGE CLASS checkbox will allow you to use a message class for storing messages. We'll discuss using messages in [Section 9.4](#).

3. Click SAVE and save the class to a suitable package.

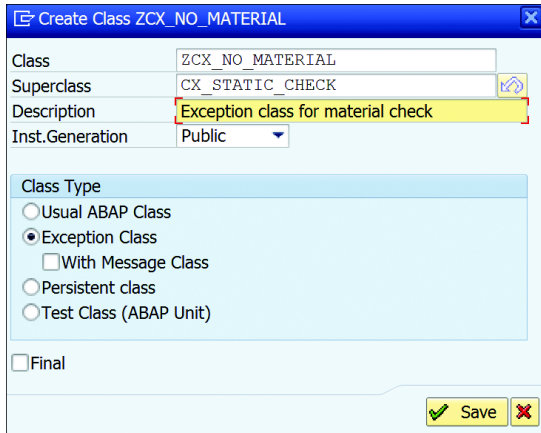


Figure 9.15 Class Properties Dialog Box

4. You'll notice that Class Builder has opened the class in the exception class perspective, which is similar to the class perspective except for minor additional options (see [Figure 9.16](#)). For example, you'll see a TEXTS tab that isn't available when creating regular classes.

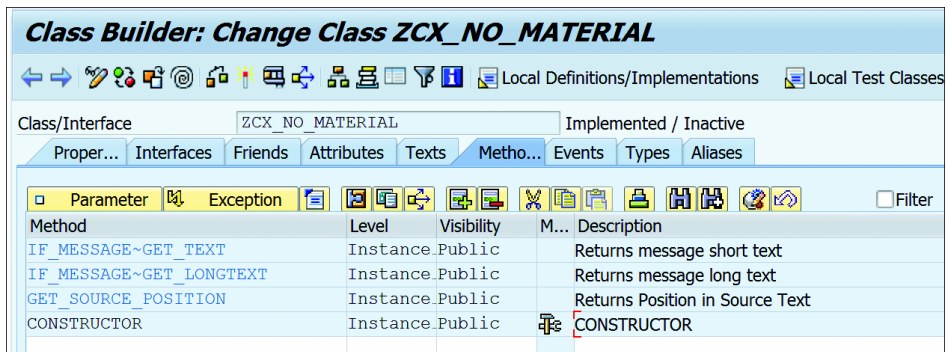


Figure 9.16 Class Builder in Exception Class Perspective

The class inherits the interfaces and methods from the `CX_ROOT` superclass. A constructor is also generated for the class, and it can't be edited. The constructor initializes the attributes defined in the class. These attributes can be passed when the exception is raised using the `RAISE EXCEPTION` statement. We'll explain how to use these attributes in the next section.

SAP recommends that you do not define new methods or redefine inherited methods.

5. Click the `ACTIVATE` button to activate the class.
6. The exception class now can be used in your programs to raise an exception. In function modules and methods of a global class, you need to select the `EXCEPTION CLASSES` checkbox to maintain the exception class, as shown in [Figure 9.17](#).

Note

Note that in function modules and methods, you can't maintain the static exceptions discussed in [Section 9.2](#) when using exception classes.

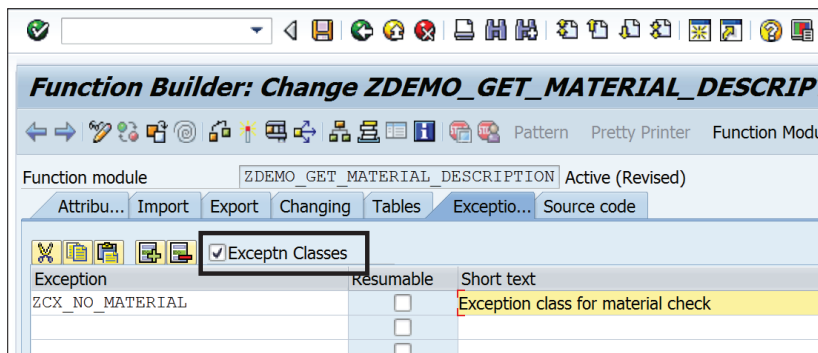


Figure 9.17 Maintaining Exception Class in Function Module

7. The exception can be raised within the function module using the syntax `RAISE EXCEPTION TYPE ZCX_NO_MATERIAL`, as shown in [Figure 9.18](#).

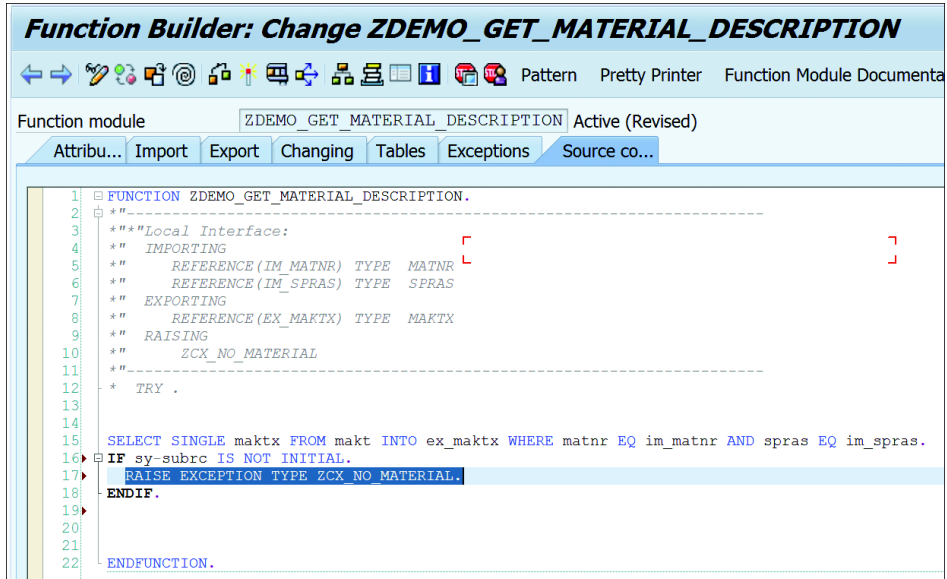


Figure 9.18 Raising Exception for ZDEMO_GET_MATERIAL_DESCRIPTION

Now that you know how to define exception classes globally, let's look at local exception classes.

9.3.4 Defining Exception Classes Locally

Local exception classes can be defined to work with exceptions within the program locally. The syntax to define a local exception class is similar to that for a regular class, except that it should inherit `CX_STATIC_CHECK`, `CX_DYNAMIC_CHECK`, or `CX_NO_CHECK`, as shown in [Listing 9.7](#). The naming convention `LCX_` is used for local exception classes.

In [Listing 9.7](#), we define a `LCX_NO_MATERIAL` local exception class, which inherits from the `CX_STATIC_CHECK` class. Apart from their visibility, working with local exception classes is similar to working with global exception classes.

```

CLASS lcx_no_material DEFINITION INHERITING FROM cx_static_check.
ENDCLASS.
CLASS lcx_no_material IMPLEMENTATION.
ENDCLASS.

```

Listing 9.7 Local Exception Class

9.4 Messages in Exception Classes

Exception classes allow us to exit an exception situation gracefully. However, sometimes an exception is caused by an error that may need human intervention to fix. In such situations, we can capture a message when the exception is raised so that the error can be identified. For this, we can pass a text message with the exception. Texts can also have dynamic field values to supplement the message.

Texts can be maintained directly in the exception class, which uses the Online Text Repository (OTR), or we can also use messages from a message class (Transaction SE91). The following subsections will look at each of these cases.

9.4.1 Using the Online Text Repository

The following steps walk you through maintaining messages in exception classes:

1. In the ATTRIBUTES tab of the exception class, define an attribute with a value you want to pass in the message (see [Figure 9.19](#)). In this example, we're maintaining MAT_NO as the attribute name of type MATNR because we plan to pass the material number in the exception message. Once the attribute is defined, click ACTIVATE.

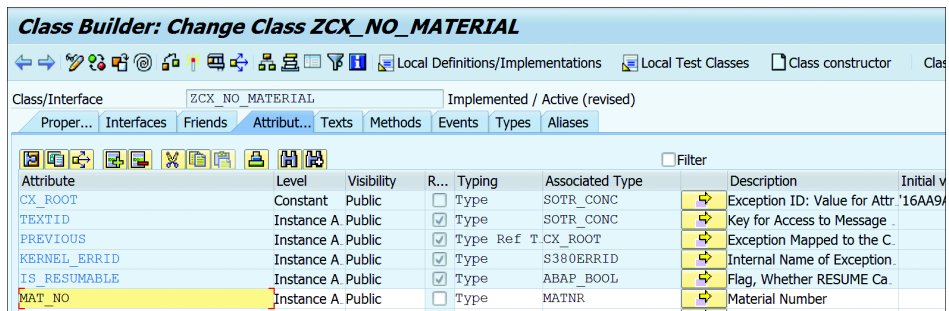


Figure 9.19 Maintaining Attribute in Exception Class

2. Under the TEXTS tab, maintain an EXCEPTION ID and a TEXT. You can pass the attribute value in the message by keeping the attribute name between ampersands (&), as shown in [Figure 9.20](#). Here, we've maintained the EXCEPTION ID as INVALID_MATERIAL and the text as The material &mat_no& is invalid by enclosing the attribute within ampersands. You can also maintain long text for the message by clicking the LONG TEXT button. When finished, click ACTIVATE.

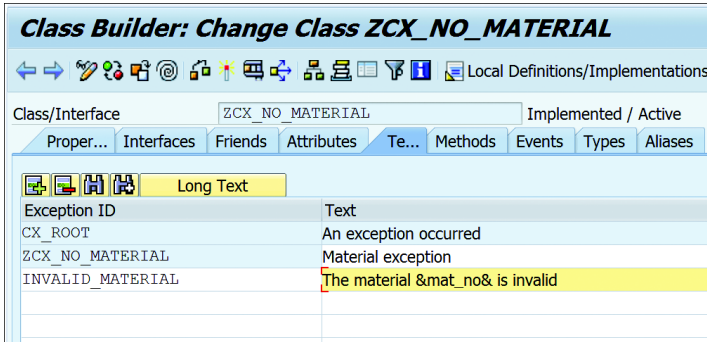


Figure 9.20 Maintaining Exception ID and Text

- At this point, under the ATTRIBUTES tab, you'll see that the EXCEPTION ID defined is added as a constant of type SOTR_CONC, as shown in Figure 9.21. The maintained text is saved in the OTR and supports internationalization (translations).

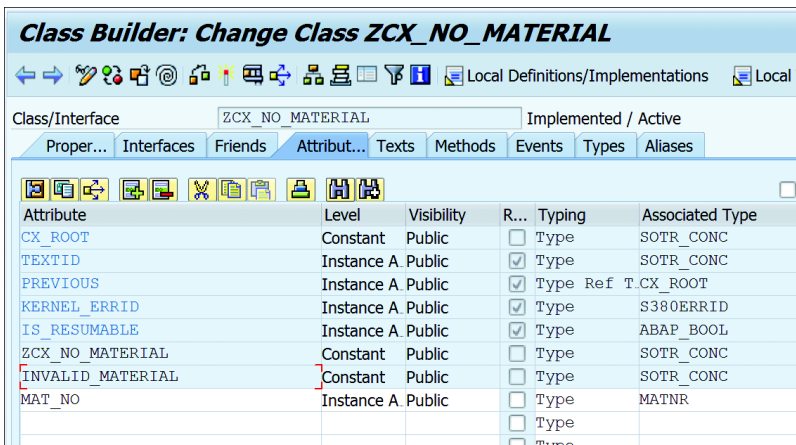


Figure 9.21 Exception ID Added as Attribute

- The class constructor automatically adds the attributes that you've added as importing parameters to its signature, which can be passed while raising the exception using the EXPORTING keyword. The constructor ensures that all the attributes are initialized properly. The textid attribute stores the EXCEPTION ID to access the corresponding text. The previous attribute stores the previous exceptions in the context. Figure 9.22 shows the signature and initialization code of the constructor method after adding the mat_no attribute.

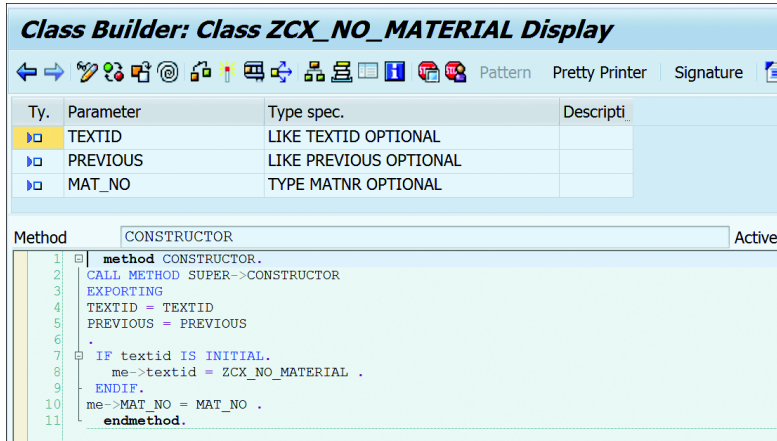


Figure 9.22 Signature of Constructor

5. To raise the exception and pass the message with the material number, you can enhance the RAISE EXCEPTION statement used earlier, as shown in Figure 9.23. Here, you call the constructor and pass TEXTID by accessing the INVALID_MATERIAL constant defined under the attributes of the class. This constant refers to the corresponding EXCEPTION ID. You also pass the material number to the mat_no attribute so that the &mat_no& field in the text will be replaced with the value.

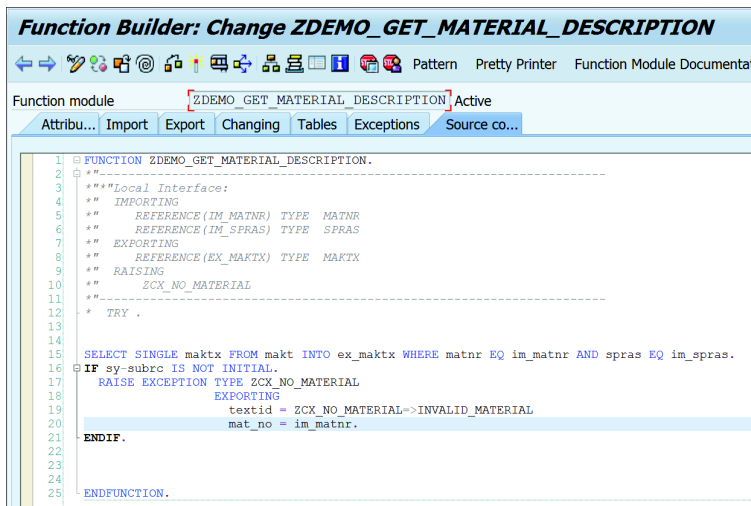


Figure 9.23 Passing Message in Exception

When this exception is caught in the program using the code listed in [Listing 9.8](#), it will result in the output shown in [Figure 9.24](#).

```
DATA v_matnr TYPE matnr VALUE '100'.
DATA v_maktx TYPE maktx.
DATA lr_err TYPE REF TO ZCX_NO_MATERIAL.
TRY.
    CALL FUNCTION 'ZDEMO_GET_MATERIAL_DESCRIPTION'
        EXPORTING
            im_matnr = v_matnr
            im_spras = sy-langu
        IMPORTING
            ex_maktx = v_maktx.

CATCH ZCX_NO_MATERIAL INTO lr_err.
    WRITE lr_err->get_text( ).
ENDTRY.
```

Listing 9.8 Catching Exception to Print Text

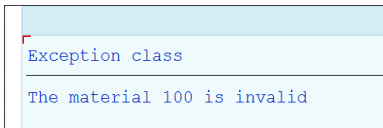


Figure 9.24 Output of Code in [Listing 9.8](#)

9.4.2 Using Messages from a Message Class

To use messages from a message class, proceed with the following steps:

1. Select the WITH MESSAGE CLASS checkbox, as shown in [Figure 9.25](#).

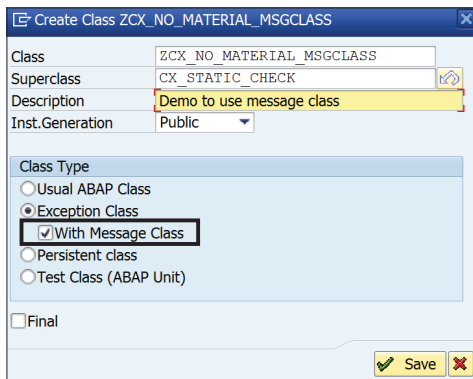


Figure 9.25 Class Properties Window

2. Once this checkbox is selected, the system automatically adds an IF_T100_MESSAGE interface to the message class, which is used to access the messages stored in message class (see [Figure 9.26](#)).

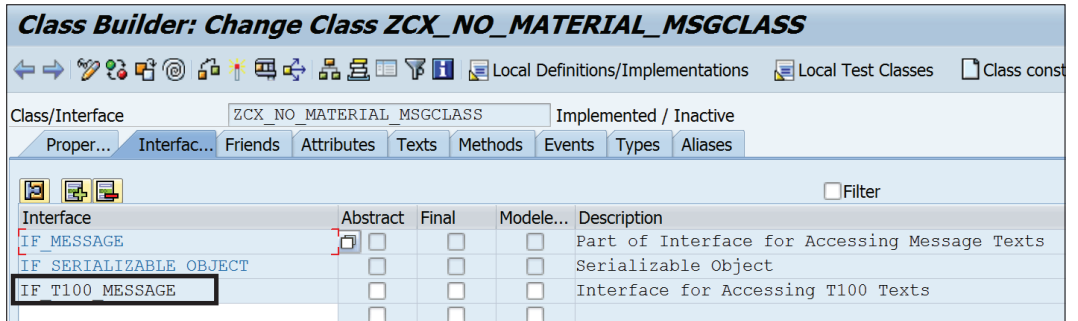


Figure 9.26 IF_T100_MESSAGE Interface Added to Message Class

3. To use the message from the message class, go to the TEXTS tab, maintain the exception ID, and click the MESSAGE TEXT button, as shown in [Figure 9.27](#).

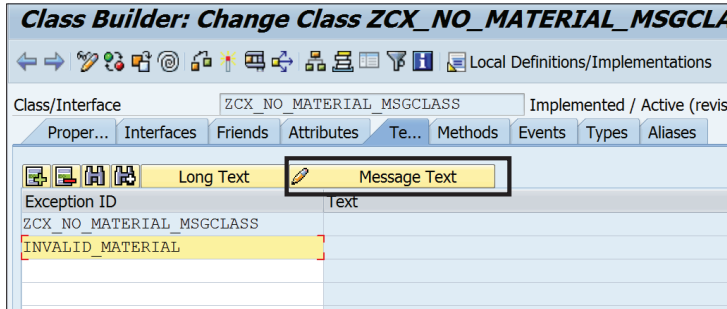


Figure 9.27 Maintaining Message from Message Class

4. In the ASSIGN ATTRIBUTES OF AN EXCEPTION CLASS TO A MESSAGE window, provide the MESSAGE CLASS name and message ID (MESSAGE NUMBER). If you want to pass the attribute value to the message, you can select the attribute from the dropdown next to each attribute field. You can pass up to four attributes, as shown in [Figure 9.28](#).
5. For the attribute value to be shown in the message, the message in the message class should use & as a placeholder, as shown in [Figure 9.29](#). When the message is called, the placeholder will be replaced with the actual value.

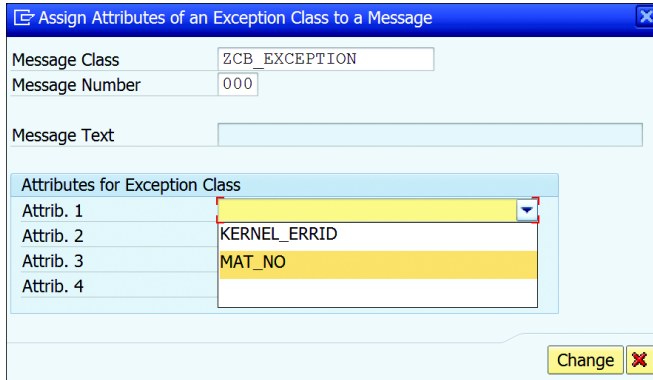


Figure 9.28 Assigning Attributes to Message

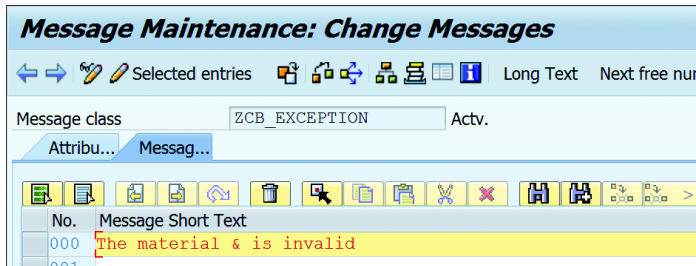


Figure 9.29 Using Placeholder in Message Maintained in Message Class

With SAP NetWeaver 7.5, SAP introduced a new interface, `IF_T100_DYN_MSG`, which extends the functionality of the `IF_T100_MESSAGE` interface, which we discussed in the preceding steps. The `IF_T100_DYN_MSG` interface contains the `IF_T100_MESSAGE` interface as a component interface and contains the `MSGTY` attribute for the message type and attributes from `MSGV1` to `MSGV4` for the placeholders of the message.

Unlike using the `IF_T100_MESSAGE` interface, for which there is no built-in support for message attributes, with the `IF_T100_DYN_MSG` interface you don't need to manually define attributes for dynamic messages, because they're predefined in the interface. The `IF_T100_DYN_MSG` interface also provides built-in support to set the message type for the message using the `MSGTY` attribute.

9.4.3 Using the MESSAGE Addition to Raise an Exception

SAP NetWeaver 7.5 introduced a new addition, `MESSAGE`, which can be used when raising exceptions. Using the `MESSAGE` addition, you can pass the message specifications, such as the message type, message ID, and message number, as well as the placeholders for the message.

If the message class implements the `IF_T100_DYN_MSG` interface, you can pass the message type using the `TYPE` addition, and the placeholders for the message can be passed using the `WITH` addition (see [Listing 9.9](#)). However, if the message class implements the `IF_T100_MESSAGE` interface, then you can't pass the message type and the message placeholders using the `WITH` addition.

[Listing 9.9](#) shows example code to raise an exception when the `IF_T100_DYN_MSG` interface is implemented in the message class. In this example, we're using the `MESSAGE` addition to pass the message specifications. The attributes and the message type can be passed directly without having to define them separately in the exception class, because the `IF_T100_DYN_MSG` interface provides built-in support.

```
RAISE EXCEPTION TYPE zcx_no_material
    MESSAGE ID sy-msgid
            TYPE sy-msgty
            NUMBER sy-msgno
            WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
```

Listing 9.9 Raising Exception with `IF_T100_DYN_MSG` Interface Implemented

9.5 Summary

In this chapter, you learned how errors can be handled in your programs to exit applications gracefully. We also looked at both procedural exceptions and class-based (object-oriented) exceptions. In the final section of this chapter, we discussed the use of messages in exception classes and explained how to use both the `OTR` and messages from message classes. We also identified some new concepts introduced in SAP NetWeaver 7.5.

At this point, you should be comfortable raising exceptions from your procedures and catching exceptions in your programs that are raised by other procedures.

In the next chapter, we'll discuss ABAP Data Dictionary in greater detail and go over how to create various dictionary objects.

ABAP Data Dictionary has been discussed in previous chapters on an as-needed basis. This chapter reviews the concepts already mentioned and offers a complete discussion of those topics that haven't yet been covered.

10 ABAP Data Dictionary

ABAP Data Dictionary is the central store used to create and manage *data definitions (metadata)*. Maintaining data definitions centrally in ABAP Data Dictionary helps avoid redundancies. In addition, because APAP Data Dictionary is a central repository, any modifications made to its data definitions will be reflected in programs automatically. In other words, ABAP Data Dictionary ensures data integrity, data consistency, and data security.

Using ABAP Data Dictionary, we can maintain a variety of objects, such as tables, views, data types, domains, type groups, search helps, and lock objects. [Figure 10.1](#) shows the initial ABAP Data Dictionary screen (Transaction SE11). Using ABAP Data Dictionary objects, we can add additional functionality to screen fields automatically, such as input help (**F1** help) and search help (**F4** help).

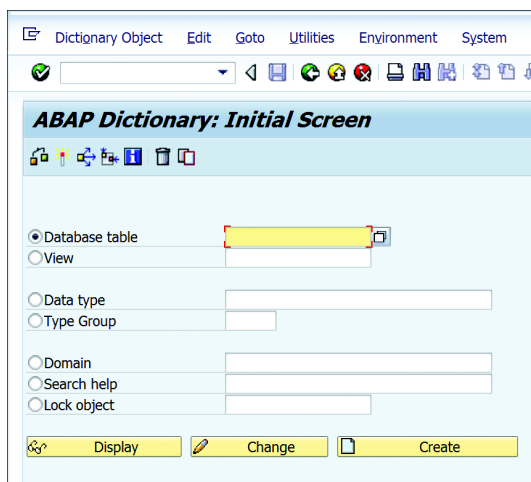


Figure 10.1 ABAP Data Dictionary: Initial Screen

ABAP Data Dictionary is completely integrated with ABAP Workbench, so definitions maintained in ABAP Data Dictionary can be used seamlessly in all ABAP Workbench tools. As for all repository objects, to allow for continuous changes to ABAP Data Dictionary objects during the development phase, such objects are stored in active and inactive versions. Any changes that are made are saved as an inactive version, allowing all the related objects to be activated together and transported. The runtime system uses only the active version of the object and ignores the inactive version, allowing us to change objects without affecting dependent applications.

We've discussed some of the ABAP Data Dictionary objects in bits and pieces so far in this book on an as-needed basis. In this chapter, we'll discuss all of the ABAP Data Dictionary objects in greater detail, including the database tables, views, data types, type groups, domains, search helps, and lock objects. This chapter will also serve as a step-by-step guide to create all of these dictionary objects.

In [Section 10.1](#), we'll discuss database tables, including how to create them and how to use primary and secondary indexes. We'll also touch on using the table maintenance generator, foreign keys, and include and append structures. Then, in [Section 10.2](#), we'll discuss creating, using, and maintaining database views, projection views, maintenance views, help views, and the new ABAP Core Data Services (CDS) views.

Next, in [Section 10.3](#), you'll learn how to create global data types, such as screen elements, structures, and table types. We'll then discuss how to use type groups to maintain global data type declarations in [Section 10.4](#), before shifting our attention to domains in [Section 10.5](#).

[Section 10.6](#) will examine elementary and collective search helps. Finally, in [Section 10.7](#), we'll provide information on using logical lock functionality in applications using lock objects. By the end of the chapter, you'll have a thorough understanding of the ABAP Data Dictionary objects and how to use them in your own projects.

10.1 Database Tables

Database tables are one of the most important objects in ABAP Data Dictionary. As you know, every SAP system consists of an underlying relational database or

in-memory database, such as SAP HANA. However, we can define tables independently of databases in ABAP Data Dictionary. We don't work at the database level to define database tables; we always use ABAP Data Dictionary to maintain database table definitions.

Once the corresponding object (the table) is defined in ABAP Data Dictionary and activated, the system creates the database table physically in the underlying database with the same structure as the table definition defined in ABAP Data Dictionary. The system automatically translates the definitions of the table in ABAP Data Dictionary to the definitions native to the underlying database. This allows us to not worry about the underlying database specifics; ABAP Data Dictionary acts as middleware between the database and the SAP system.

Figure 10.2 shows the table definition of the transparent table `MAKT` maintained in ABAP Data Dictionary. Figure 10.3 shows the corresponding database table created in the underlying database.

The screenshot shows the 'Dictionary: Display Table' window for table MAKT. The table is a transparent table, active, with a short description 'Material Descriptions'. The fields are listed in a table below.

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3		Client
MATNR	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MATNR	CHAR	18		Material Number
SPRAS	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	SPRAS	LANG	1		Language Key
MAKTX	<input type="checkbox"/>	<input type="checkbox"/>	MAKTX	CHAR	40		Material Description (Short Text)
MAKTG	<input type="checkbox"/>	<input type="checkbox"/>	MAKTG	CHAR	40		Material description in upper case for matchcodes

Figure 10.2 Table Definition in ABAP Data Dictionary

A *transparent table* in ABAP Data Dictionary has a one-to-one projection of the underlying database table. In other words, when we define a transparent table in ABAP Data Dictionary, exactly one database table with identical names and fields is created in the underlying database.

In addition to transparent tables, we can define other categories of tables in ABAP Data Dictionary, such as cluster tables, pool tables, and global temporary tables. You'll learn more about these table categories in [Section 10.1.1](#).

Table MAKT: Display Database Object

Header von MAKT

Client-Specific	Storage Type
Ja	Undefiniert

Fields of MAKT

Fld Name	Position	Data Type	Length	Decimals	Not null	Default
MANDT	1	nvarchar	3		X	N'000'
MATNR	2	nvarchar	18		X	N''
SPRAS	3	nvarchar	1		X	N''
MAKTX	4	nvarchar	40		X	N''
MAKTG	5	nvarchar	40		X	N''

Indexes for MAKT

Unique Index MAKT~0
MANDT
MATNR
SPRAS

Index MAKT~M
MANDT
SPRAS
MAKTG

Figure 10.3 Database Object Created in Underlying Database

A database table in ABAP Data Dictionary consists of the following components:

► **Table fields**

Each database table consists of fields. Table fields define the structure of the database table. Each database table field is fully defined by assigning a data type. The fields represent table columns, and the primary key of the table exists as a subset of table fields.

► **Foreign keys**

Foreign keys define the relationship between the current database table and other database tables in the database.

► **Technical settings**

Technical settings control the creation of the database table in the underlying database. We can maintain different settings, such as initial database table size, tablespace to create the database table, and so on.

► **Indexes**

The key fields of the database table form the primary index of the database table. Additional secondary indexes can be created to speed up data selection.

In this section, we'll look at these database table components, as well as inputting data into a database table using a maintenance view. We'll also discuss including structures in tables and using append structures to enhance standard SAP database tables.

But first, let's walk through the steps for creating a database table in the ABAP Data Dictionary.

10.1.1 Creating a Database Table

The following is a step-by-step guide to create a database table in the ABAP Data Dictionary:

1. On the initial screen of the ABAP Data Dictionary (refer back to [Figure 10.1](#)), select the DATABASE TABLE radio button, enter the name of the database table you wish to create, and click the CREATE button. The name of the database table should be in the customer namespaces (starting with Y or Z).
2. The table will open in the DICTIONARY: CHANGE TABLE view. On this screen, provide a meaningful description for the table. By default, the system will create a transparent table. If you wish to create other types of tables, such as cluster, pool, or global temporary tables, use the menu path EXTRAS • CHANGE/DISPLAY TABLE TYPE to change the table category.

The following table categories can be defined in ABAP Data Dictionary:

- **Transparent table:** Each transparent table defined in ABAP Data Dictionary creates exactly one database table in the underlying database, with identical names and fields. All customer tables, including tables that store business data, should be defined as transparent tables.

- ▶ **Pool tables and cluster tables:** Pooled tables and cluster tables are an SAP-specific category of database tables that has one to many relationships with underlying database tables. In other words, when you define a pooled table or a cluster table in ABAP Data Dictionary, it can create multiple database tables in the underlying database. These tables are usually used by SAP to store internal control information, documentation, and so on.

Transparent, pool, and cluster tables

These tables shouldn't be used to store business data. SAP recommends transforming all existing pooled tables and clustered tables to transparent tables. Similar to transparent tables, you can use Open SQL statements to access pooled and cluster tables; however, you can't use Native SQL statements for pooled and cluster tables.

You can read more about these tables at http://help.sap.com/abapdocu_750/en/index.htm?file=abenddic_database_tables_poolclu.htm.

- ▶ **Global temporary tables (GTTs):** With the release of SAP NetWeaver 7.5, a new table category, *global temporary tables*, has become available. GTTs are special transparent tables used to store temporary data within a database logical unit of work (LUW). A database LUW exists between each dialog step. Each dialog step comprises programming logic between two successive screens. Individual dialog steps of a program can be executed by different work processes in the application server, and a program consists of multiple dialog steps. GTTs allow you split complicated database processes into multiple steps, such as storing temporary subtotals.

GTTs are designed specifically for storing temporary data within a database LUW and thus incur far lower administration costs than regular transparent tables. A GTT is always empty at the start of a database LUW, which implies that it should be cleared at the end of each database LUW. Database systems generally provide native support for GTTs by implicitly defining the data in a GTT database table as transaction-specific data. This ensures that the data in a GTT is implicitly deleted at the end of a database LUW (when a database commit or rollback is performed).

If a GTT defined in ABAP Data Dictionary is filled explicitly using Open SQL statements, then it should be cleared explicitly at the end of a database LUW using the statement `DELETE FROM dbtab`. A GTT should also be explicitly cleared

if ABAP statements such as `COMMIT WORK`, `COMMIT CONNECTION`, `ROLLBACK WORK`, `ROLLBACK CONNECTION`, or associated Native SQL statements are used. The `COMMIT WORK` and `ROLLBACK WORK` statements clear the GTTs of all currently open database connections, but you can delete the GTTs of the specified connection by using the `COMMIT CONNECTION` and `ROLLBACK CONNECTION` statements.

If an ABAP Data Dictionary GTT isn't cleared explicitly via the `DELETE FROM dbtab` statement before an implicit database commit is performed, it will result in the runtime error `COMMIT_GTT_ERROR`. This runtime error is raised even if the table is empty if the statement `DELETE FROM dbtab` is supplied with a `WHERE` clause. The explicit clearing of ABAP Data Dictionary GTTs is forced by SAP to ensure that the code is transparent and easy to understand. For example, the developer may be surprised to find the table empty in between a programming module if the table was cleared implicitly by the database system due to a database LUW performed by an RFC. Explicitly clearing GTTs also helps to avoid platform dependencies, because not all database systems implicitly clear GTTs. GTTs can be accessed just like regular transparent tables via both Open SQL and Native SQL statements. However, SAP recommends using only Open SQL statements when working with GTTs.

3. The `DICTIONARY: CHANGE TABLE` view consists of multiple tabs to maintain various properties of the database table. The `DELIVERY AND MAINTENANCE` tab is selected by default. Under this tab, you can maintain the settings for `DELIVERY CLASS` and `DATA BROWSER/TABLE VIEW MAINTENANCE`:
 - ▶ `DELIVERY CLASS`: This field allows you to select the type of data the database table will contain. Use `F4` help to select from one of the available options, as shown in [Figure 10.4](#). The option selected for `DELIVERY CLASS` will control the transport of table data during installation, upgrades, and client copies. This setting also influences transport between customer systems. For this example, select `A` so that you can store application data in this table.
 - ▶ `DATA BROWSER/TABLE VIEW MAINTENANCE`: This field allows you to restrict whether data can be entered by the user manually or the table can be updated only through programs (i.e., Transaction `SE16`). Select the relevant option from the dropdown list. Use `F1` help on both fields to learn more about these settings.

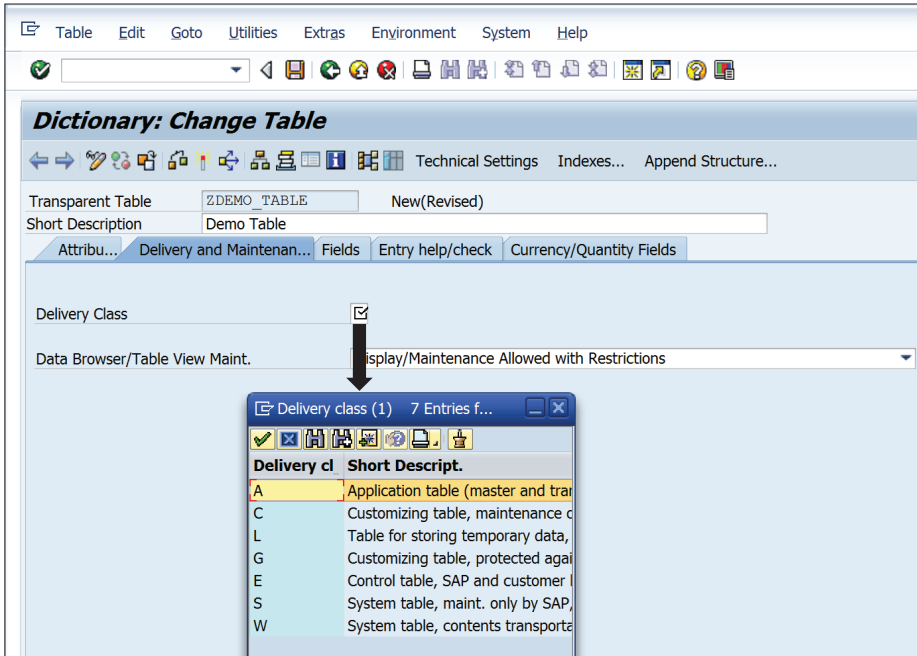


Figure 10.4 Delivery and Maintenance Tab

4. Once the settings are maintained in the `DELIVERY AND MAINTENANCE` tab, click the `FIELDS` tab. This tab allows you to maintain the fields for the database table (see [Figure 10.5](#)). The `FIELDS` tab presents a table control to maintain various attributes for the field.

Under the `FIELDS` tab is a table containing several columns:

- ▶ **FIELD:** Provide a name for the field. The acceptable characters for field names are letters, numbers, and underscore. The field name must begin with a letter. A field name must not be longer than sixteen characters.
- ▶ **KEY:** Select this checkbox if the field is part of the table key. All the key fields should be at the top of the table contiguously.
- ▶ **INITIAL VALUES:** Select this checkbox to set a `NOT NULL` flag in the database. This setting is useful when adding new fields to the existing database table, because the system will automatically fill the field with type-compatible initial values for all the rows of the field. If this flag is not set for new fields in an existing table, the system will assign a `NULL` value for all the rows of the field. Because there's no counterpart for `NULL` in ABAP, this field can then be

queried only using the special WHERE condition IS NULL or IS NOT NULL with Open SQL.

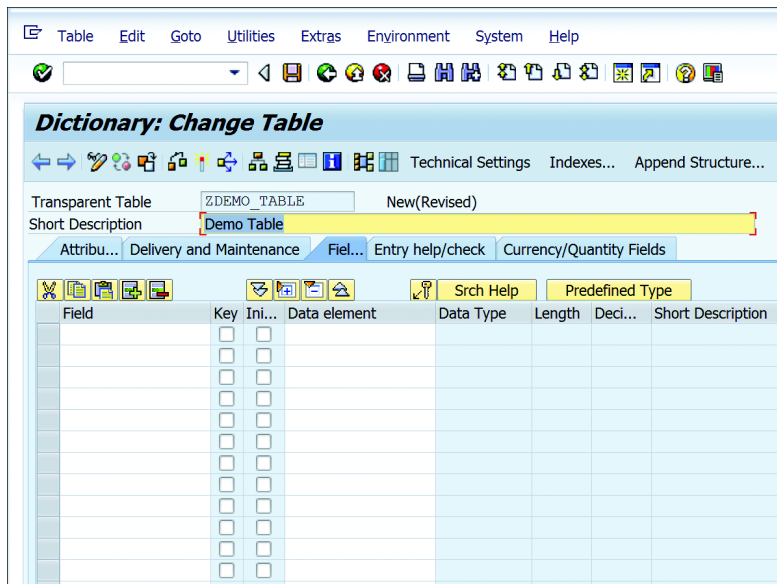


Figure 10.5 Fields Tab

Please keep the following points in mind when using this setting:

- This setting has no effect for new tables, because the NOT NULL flag is set by default for new tables. This flag is always set for key fields.
 - Components of the table can be defined by including components of other flat structures in ABAP Data Dictionary. These structures are known as *include structures*. If this flag is set for an include structure, then it applies to only to the structure components for which the flag is set. If the structure component is not set with this flag, it will be ignored.
 - It's recommended to use this flag only if required or if the table has a few records. In tables with many records, it can take a long time to set the type-compatible initial values.
- ▶ **DATA ELEMENT:** Provide the data element from which the field should inherit semantic and technical properties. If you want to create a new data element, you can enter the name of the data element and double-click it to create a new data element.

- ▶ **DATA TYPE/LENGTH/DECIMAL PLACES/SHORT DESCRIPTION:** These fields are disabled by default and filled automatically based on the data element provided in the previous field. If you wish to maintain the data type, length, decimal places, and short description manually, click the **PREDEFINED TYPE** button (see [Figure 10.6](#)) and enter the details manually. Note that only limited functionality is provided for fields without data elements. These fields can't have foreign keys or fixed values defined. There is no **[F1]** help for these fields.
- ▶ **GROUP:** When including a structure in the table, the structure can be assigned a group name. ABAP programs can access the fields of this include structure together or individually using the group name. Note that the components of the include structure are part of the table, so they can be accessed directly using the table name reference. Providing a group name is optional.

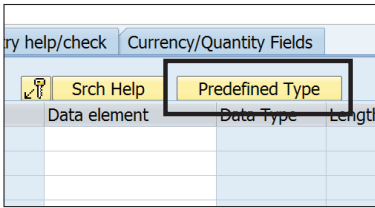


Figure 10.6 Predefined Type

5. Once you've finished with the entries under the **FIELDS** tab, go to the **ENTRY HELP/CHECK** tab. The **ENTRY HELP/CHECK** tab shows information about the associated check table, input help, and search help for the fields (see [Figure 10.7](#)).

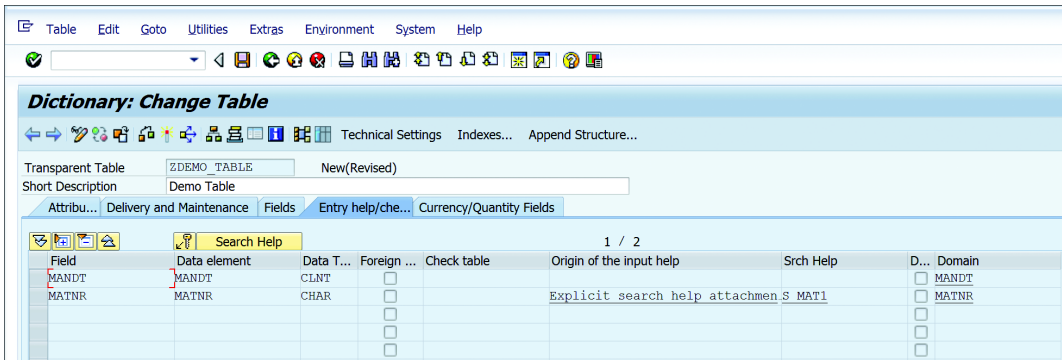
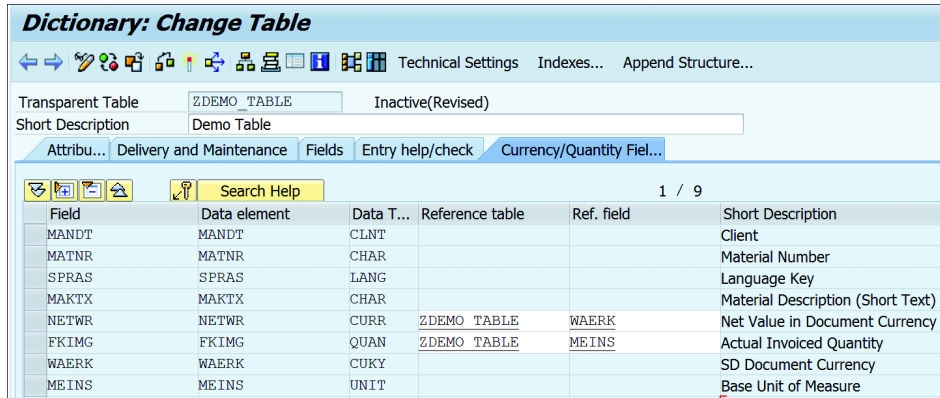


Figure 10.7 Entry Help/Check Tab

6. If the table has fields of type currency (type CURR) or quantity (type QUAN), then the reference fields should be maintained in the CURRENCY/QUANTITY FIELDS tab (see [Figure 10.8](#)). *Currency fields* should be referred to a field of type CUKY and *quantity fields* should be referred to a field of type UNIT. Reference fields can exist in the same table or another table. This tab can be ignored if your table doesn't have any currency or quantity fields.



Field	Data element	Data T...	Reference table	Ref. field	Short Description
MANDT	MANDT	CLNT			Client
MATNR	MATNR	CHAR			Material Number
SPRAS	SPRAS	LANG			Language Key
MAKTX	MAKTX	CHAR			Material Description (Short Text)
NETWR	NETWR	CURR	ZDEMO TABLE	WAERK	Net Value in Document Currency
FKIMG	FKIMG	QUAN	ZDEMO TABLE	MEINS	Actual Invoiced Quantity
WAERK	WAERK	CUKY			SD Document Currency
MEINS	MEINS	UNIT			Base Unit of Measure

Figure 10.8 Currency/Quantity Fields Tab

7. Once all the fields are maintained, you need to maintain the enhancement category for the table. If the enhancement category isn't maintained, the system will produce a warning message during activation. To maintain the enhancement category, follow the menu path EXTRAS • ENHANCEMENT CATEGORY to open the MAINTAIN ENHANCEMENT CATEGORY dialog box, as shown in [Figure 10.9](#). The following options are available:
- ▶ CAN BE ENHANCED (DEEP): Select this option if the table can be enhanced by adding fields of any type.
 - ▶ CAN BE ENHANCED (CHARACTER-TYPE OR NUMERIC): Select this option if the table can be enhanced with character or numeric fields but must not contain any deep data types, such as tables, object references, or strings.
 - ▶ CAN BE ENHANCED (CHARACTER-TYPE): These tables can be enhanced with character-type fields (types C, N, D, T, and X) only.
 - ▶ CANNOT BE ENHANCED: Select this option if the table must not be enhanced in any way.

Click the INFORMATION icon (lowercase i) in the window to learn more about these options.

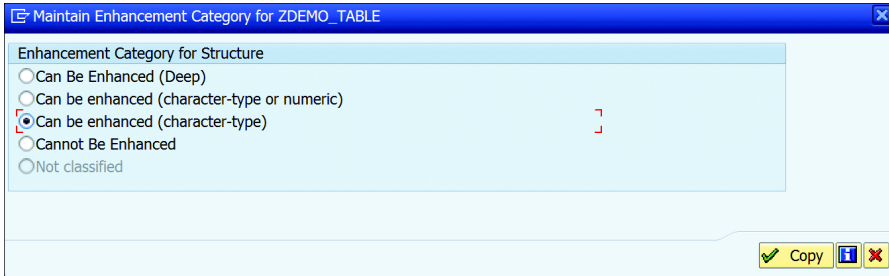


Figure 10.9 Maintain Enhancement Category

8. Once the enhancement category for the table is maintained, you need to maintain the technical settings for the table before it can be activated.

Technical settings define how the table is created in the database. Save your table before calling the technical settings screen. To maintain technical settings for the table, click the TECHNICAL SETTINGS button (see [Figure 10.10](#)) in the application toolbar to call the DICTIONARY: MAINTAIN TECHNICAL SETTINGS screen (see [Figure 10.11](#)). This will open Transaction SE13.

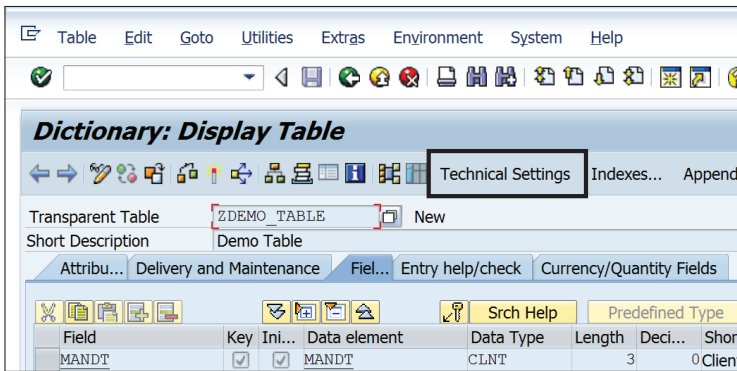


Figure 10.10 Calling Technical Settings Screen

9. You can maintain the following technical settings for the database table on this screen (see [Figure 10.11](#)):

- ▶ DATA CLASS: This setting is significant only for Oracle and Informix database platforms. For all other database platforms, this setting should be ignored.

This setting decides the physical area of the database (known as the *tablespace*) in which the table must be created. Use **[F4]** help to select the type of data that you plan to store in the table. Based on the value, the system will automatically decide the location of the table in the database.

- ▶ **SIZE CATEGORY:** SIZE CATEGORY determines the initial memory space to be allocated for the database table. If the initial memory space is used up, the size of the table will be incremented by the same value, known as *extends*. Provide a good estimate to avoid too many small memory areas or memory areas that are too big. Use the **[F4]** help to select the applicable value. You can ask your functional consultant to provide you an estimate of expected records.

The screenshot displays the 'Dictionary: Maintain Technical Settings' interface. At the top, there is a menu bar with 'Settings', 'Edit', 'Goto', 'System', and 'Help'. Below the menu is a toolbar with various icons. The main content area is titled 'Dictionary: Maintain Technical Settings' and includes a status indicator 'Revised<->Active'. The form contains the following fields:

- Name: ZDEMO_TABLE
- Transparent Table: Transparent Table
- Short Descript.: Demo Table
- Last Changed: E9685324
- Last Changed: 19.04.2016
- Status: New
- Status: Not saved

There are two tabs: 'General Proper...' and 'DB-Specific Properties'. The 'DB-Specific Properties' tab is active and contains the following sections:

- Logical Storage Parameters:**
 - Data Class: [Yellow box]
 - Size Category: [White box]
- Buffering:**
 - Buffering Not Allowed
 - Buffering allowed but switched off
 - Buffering Activated
- Buffering Type:**
 - Single Records Buffered
 - Generic Area Buffered
 - Fully Buffered
 - Number of Key Fields: [White box]
- Log Data Changes
- Write access only with JAVA

Figure 10.11 Technical Settings Screen

- ▶ **BUFFERING:** As discussed in [Chapter 2, Section 2.2.2](#), these settings determine whether the data in the database table should be buffered. If buffering

is switched on, the system will keep a copy of the database content in the application server to speed up data selection and improve performance. Because the data for the buffered tables can be accessed directly from the application server buffer, buffering helps to avoid the time-consuming process of accessing the database and also reduces the network load.

Buffering is recommended for data that seldom changes. When an application changes the data of the buffered table in the application server, the database interface notes this change in a log. A synchronization program checks this log and updates based on the changes at regular intervals (say, every one to two minutes) to keep the application server buffer in sync with the data in the corresponding database table. If the data in the database table changes frequently, then the synchronization program continuously updates the data in the buffer, which may impact system performance. For this reason, we don't recommend buffering tables for which data changes frequently or using buffering when access to real-time data is required.

The following options are available to set buffering permissions:

- ▶ **BUFFERING NOT ALLOWED:** This setting turns off the buffering for the database table and specifies that this table should not be buffered in any system.
- ▶ **BUFFERING ALLOWED BUT SWITCHED OFF:** This option specifies that the buffering is switched off for the database table when delivered but can be activated later in other systems, depending on how the table is used.
- ▶ **BUFFERING ACTIVATED:** This setting activates buffering, and the database table contents are buffered to the application server.
- ▶ **BUFFERING TYPE:** If the table buffer is activated, a buffering type must be defined. **BUFFERING TYPE** defines how many database table records are buffered when a record is read from the database table. The following buffering types are available: **SINGLE RECORDS BUFFERED**, **GENERIC AREA BUFFERED**, **FULLY BUFFERED**. If **SINGLE RECORDS BUFFERED** is selected, only those rows that are accessed by the program are buffered. If **GENERIC AREA BUFFERED** is selected, the number of key fields should be entered in the input field. When a row is read from the database table, the system buffers all the matching rows of the primary key fields up to the number of key fields maintained for this setting. In addition, if **FULLY BUFFERED** is selected, all the rows of the table are buffered when a row of the table is read. With this option, either all the data in the database table exists in the buffer or none of the data exists.

- ▶ **LOG DATA CHANGES:** This setting determines if the data in the table should be logged for any changes. If this option is selected, all the changes to the data in the database table will be logged.
- ▶ **WRITE ACCESS ONLY WITH JAVA:** If this setting is selected, the data in the table can be updated or modified only from Java applications.

Global Temporary Tables

Note that for GTTs the technical settings are predefined and can't be modified. Data type and size category are ignored and undefined. SAP buffering is not allowed, and logging is switched off.

10. Once the required settings are made, click **SAVE** in the standard toolbar. After saving the technical settings, click the **BACK** button to go back to the table and activate it. The physical table in the database is created when the table is activated.

We've now covered the steps for creating a database table. In the following section, we'll look at one of the key components of database tables: indexes.

10.1.2 Indexes

In this section, we'll discuss how to use both primary and secondary indexes for database tables.

Primary Index

The key fields of a database table form the *primary index* of the table. When the database table is created, the system automatically generates the primary index for the key fields. You can think of an index as a copy of the actual table, but with a reduced number of fields (consisting of only index fields) and with pointers to the other fields of the table. The entries in the index are always sorted to provide faster access to the table records. When a search is made for a table record using the index fields, the system checks the index instead of manually scanning the database table in order to retrieve the data faster.

For example, say table `XYZ` has four fields—`A`, `B`, `C`, and `D`—with `A` as the key field. If we write a select query as `SELECT * FROM TABLE xyz WHERE A = '10'`, the system

checks the index to locate the entry where $A=10$ and uses the pointer to retrieve the remaining fields of the row from the table (see [Figure 10.12](#)).

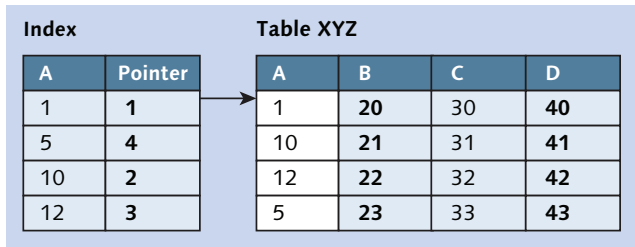


Figure 10.12 Index

Secondary Index

Apart from the primary index created by default, we can create other indexes, called *secondary indexes*. These are required if we're performing a look up on a table in which the complete primary key is not used in the `WHERE` clause. For example, if we're searching table `XYZ` where $D=41$, then the system can't make use of the primary index, because the primary index stores only the key fields.

In such situations in which a suitable index isn't available, the system performs a full table scan, which affects performance. To avoid a full table scan if the complete primary key can't be supplied in the `WHERE` clause, we can create secondary indexes.

The *database kernel* consists of an SQL optimizer, which analyzes the SQL query and optimizes access to the data by performing cost determination and analyzing table statistics. The *SQL optimizer* selects the best search strategy for accessing the data. For example, if the table contains multiple indexes, the SQL optimizer automatically selects the most suitable index for reading the data.

The SQL optimizer can also determine if performing a full table scan results in more efficient access than selecting any of the available indexes. Sometimes, the SQL optimizer may ignore an index and perform a full table scan even when a suitable index is available. This can happen if the SQL optimizer finds the cost of a full table scan to be lower than the cost of using a secondary index due to profile settings that are more favorable to full table scans. Certain databases, such as

Oracle, support *hints*, which allows you as an application developer to make decisions that are usually made by the SQL optimizer.

Sometimes, you may know information about your data that the SQL optimizer may not know; for example, you may know that a certain index is more selective for certain queries, but the SQL optimizer may ignore it. In such cases, you can use hints to force the SQL optimizer to use a specific index. You can supply hints with your query by using the `_%HINTS` parameter, as shown in [Listing 10.1](#).

```
SELECT <fields> INTO TABLE<itab> FROM <database table>
  WHERE <logical expression>
  AND <logical expression>
_%HINTS <database name> 'INDEX("<database table>" "Index name")'.
```

Listing 10.1 Supplying Hints Using `_%HINTS`

In [Listing 10.1](#), the `_%HINTS` parameter is followed by the database name and the index of the table that should be used. If the database is Oracle and we want the SQL optimizer to use the secondary index `SPFLI~001` of table `SPFLI`, we can write the query as shown in [Listing 10.2](#).

```
SELECT * FROM spfli
  INTO TABLE it_spfli
  WHERE cityfrom EQ ' NEW YORK'
  AND cityto EQ ' SAN FRANCISCO'
  %_HINTS ORACLE 'INDEX("SPFLI" "SPFLI~001")'.
```

Listing 10.2 Hints in Oracle

Multiple secondary indexes on the same table are identified by using a three-character index identifier.

The following steps walk through how to define a secondary index for a database table:

1. First, on the `DICTIONARY: CHANGE TABLE` screen, click the `INDEXES` button, as shown in [Figure 10.13](#).
2. The system will open a window displaying all the secondary indexes created for the database table. If no secondary indexes are defined, this window will show an empty list. Click the `CREATE` button and choose `CREATE INDEX` from the drop-down menu, as shown in [Figure 10.14](#).

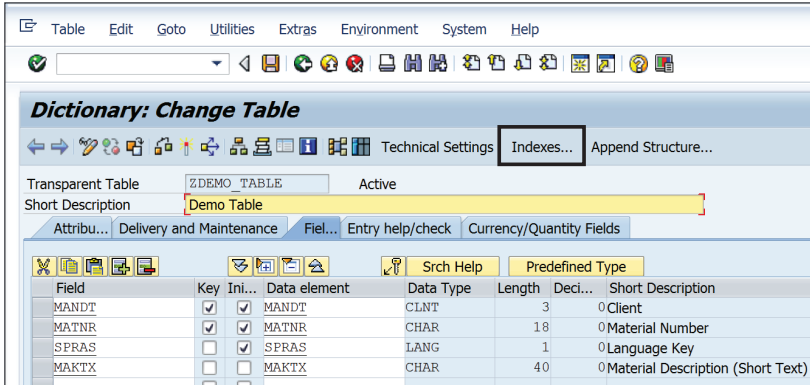


Figure 10.13 Creating Index

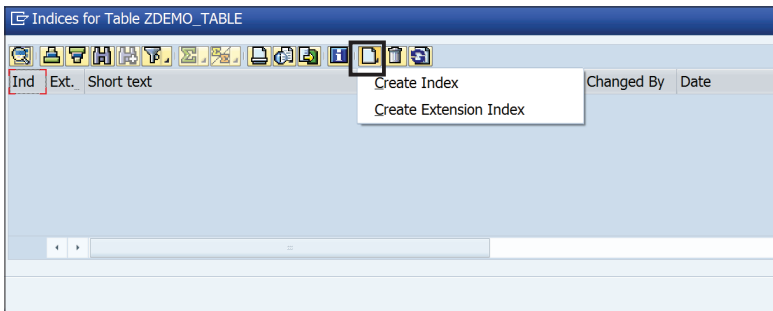


Figure 10.14 Create Secondary Index

3. In the CREATE INDEX dialog box that appears, input the index name and click CONTINUE (see Figure 10.15).

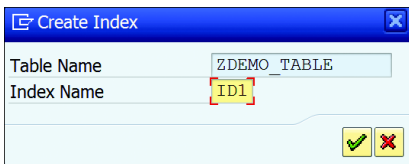


Figure 10.15 Maintain Index Name

4. On the DICTIONARY: CHANGE INDEX screen (see Figure 10.16), provide a meaningful short description. On this screen, you have the option to select to create a *unique index* or a *non-unique index*. If the values in index fields uniquely identify a row (i.e., the combination of the values in the key fields shouldn't exist

more than once in the table), select `UNIQUE INDEX`. Otherwise, select `NON-UNIQUE INDEX`. When you select `NON-UNIQUE INDEX`, you can also select if the index should be created in all database systems (`INDEX ON ALL DATABASES SYSTEMS`), `FOR SELECTED DATABASE SYSTEMS`, or not created at all in the database (`NO DATABASE INDEX`).

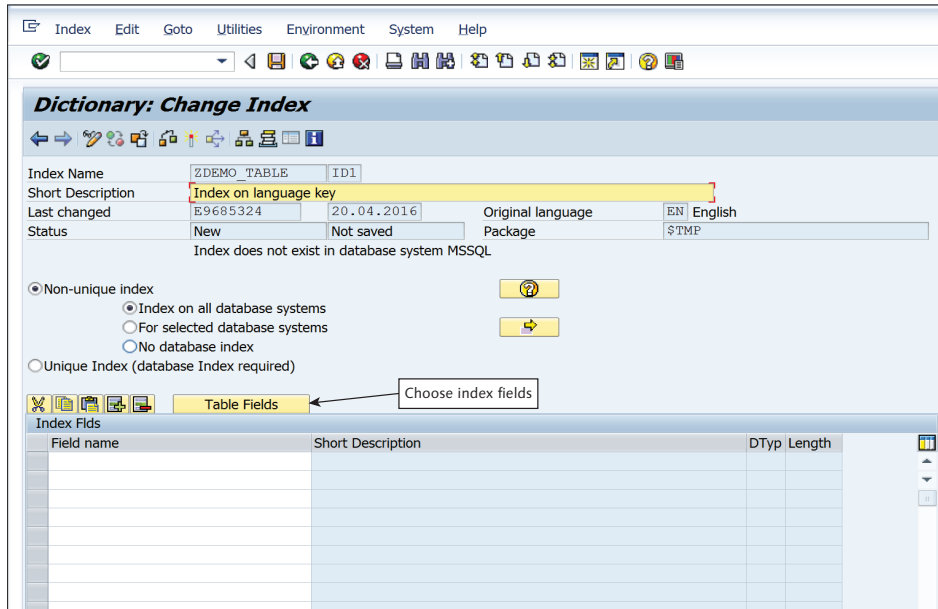


Figure 10.16 Maintaining Index Properties

5. Use the `TABLE FIELDS` button to select the fields of the index. The order of the fields selected is very important. While querying the table, the index is used only if the order of the fields is specified in the same order as in the `WHERE` clause of a `SELECT` statement or if a SQL optimizer hint is supplied (for some database systems, such as Oracle).
6. Once the fields are selected, activate the index. The secondary index is now created for the table.

Creating additional secondary indexes can have side effects for system performance. It isn't a good idea to create too many indexes or indexes that have too many common fields, because doing so might make it more difficult for the optimizer to choose the most selective index.

You can check all the available indexes for the database object by selecting the INDEXES button in the application toolbar, as shown in [Figure 10.17](#).

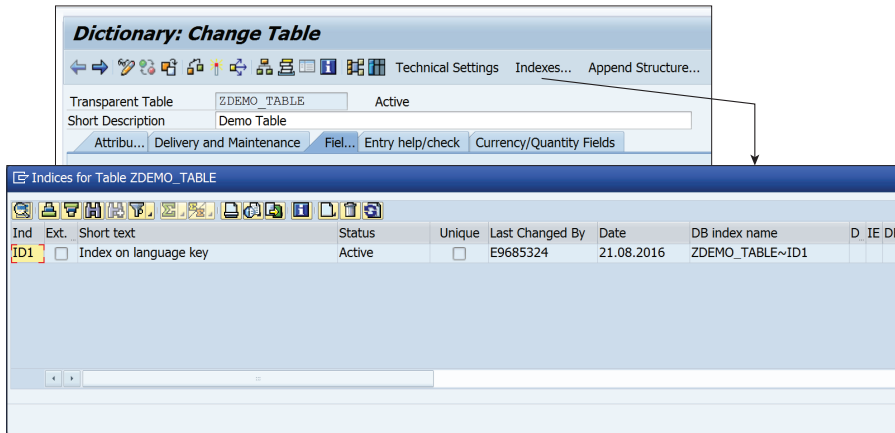


Figure 10.17 Displaying Indexes

10.1.3 Table Maintenance Generator

To make it easy for users to input data into a database table, we can generate a maintenance view for the database table.

Follow these steps to create a maintenance view:

1. From the table maintenance screen, select the DELIVERY AND MAINTENANCE tab. For the TABLE VIEW MAINT. field, select DISPLAY/MAINTENANCE ALLOWED (see [Figure 10.18](#)) from the dropdown and activate the table.

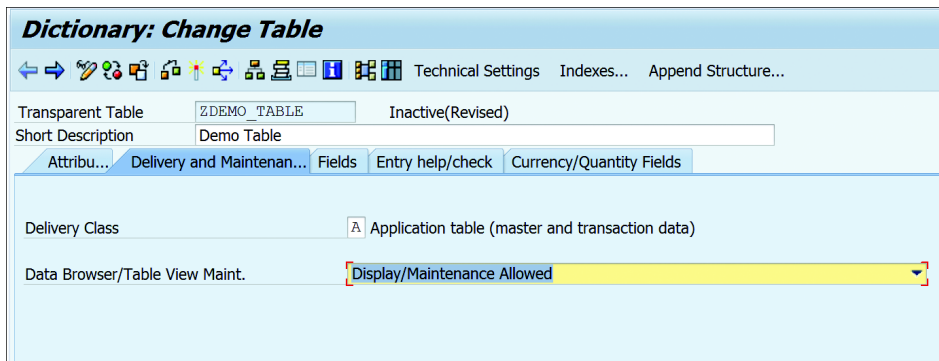


Figure 10.18 Set Display/Maintenance Allowed

- Follow the menu path UTILITIES • TABLE MAINTENANCE GENERATOR. This will open the GENERATE TABLE MAINTENANCE DIALOG: GENERATION ENVIRONMENT window, as shown in [Figure 10.19](#).

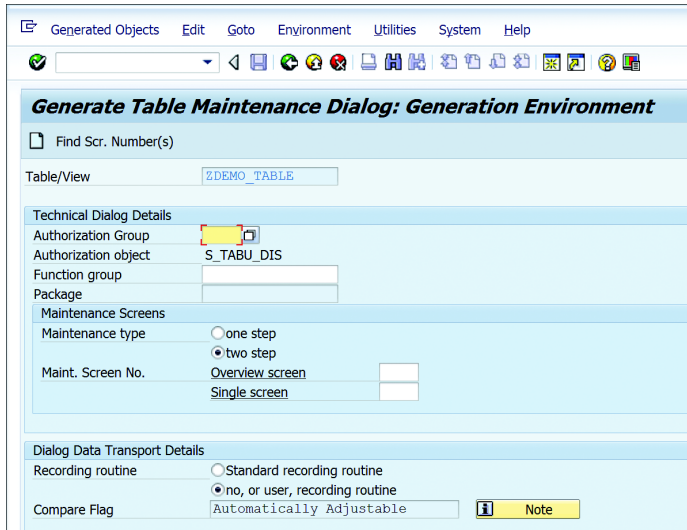


Figure 10.19 Table Maintenance Generator

On this screen, you have the following options:

- ▶ **AUTHORIZATION GROUP:** Provide the AUTHORIZATION GROUP to restrict the maintenance permission to select users. Enter “&NC&” in this field if you don't want to set any restrictions.
- ▶ **FUNCTION GROUP:** Enter the name of the function group in which all the required code will be generated.
- ▶ **PACKAGE:** Displays the package to which the object is assigned.
- ▶ **MAINTENANCE TYPE:** There are two radio button options for this field:
 - **ONE STEP:** This will create a single screen for display and maintenance.
 - **TWO STEP:** This will create two screens, one for display and one for maintenance.
- ▶ **MAINT. SCREEN NO.:** You can enter the screen numbers that should be created or use the FIND SCREEN NUMBERS button in the application toolbar for the system to propose available screen numbers for the chosen function group.

- ▶ **RECORDING ROUTINE:** Allows you to record the changes to the table records, which can then be transported. Select **STANDARD RECORDING ROUTINE** or the **NO, OR USER, RECORDING ROUTINE**. If you select **STANDARD RECORDING ROUTINE**, the system will use the standard routine to record the changes and the data can be transported based on the client settings. Disabling this option will result in changes not recorded and data not transported.
 - ▶ **FIND SCREEN NUMBERS:** Click this button to allow the system to propose the available screen numbers for the function group used.
3. Once the required settings are maintained, click the **CREATE** button to generate the maintenance view for the table.

The maintenance view that is generated allows you to perform various actions using different predefined events.

To execute custom code for the available events, proceed with the following steps:

1. Follow the menu path **ENVIRONMENT • MODIFICATION • EVENTS** on the table maintenance screen.

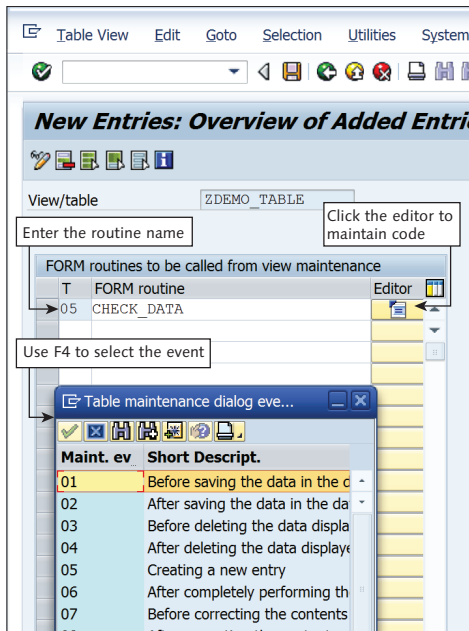


Figure 10.20 Using Events

- In the next window, click the NEW ENTRIES button and select the event using the `F4` help. Enter the form routine name that should be called for the event, as shown in [Figure 10.20](#).

Click the EDITOR icon to create an include program in the function group (see [Figure 10.21](#)) and maintain the subroutine with your custom code (see [Figure 10.22](#)).

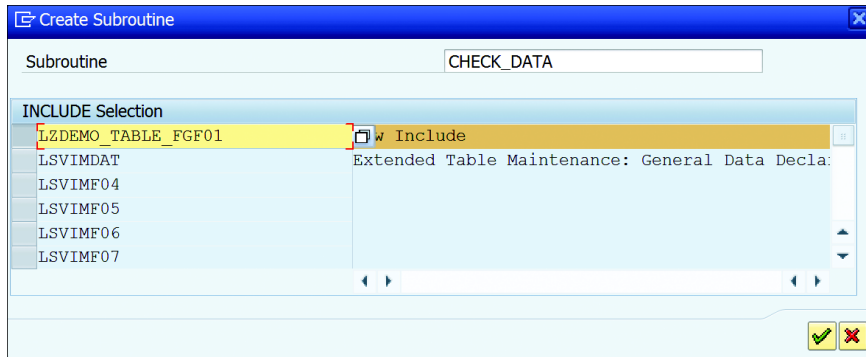


Figure 10.21 Create Include

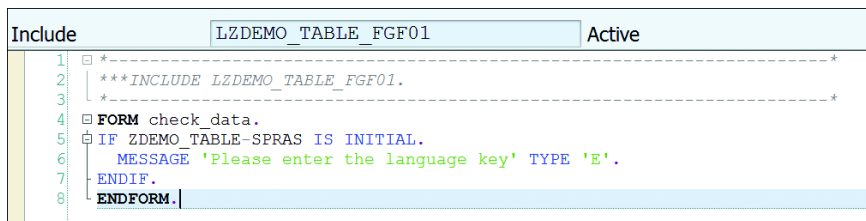


Figure 10.22 Maintain Subroutine

- Here, use event 05, which is triggered when creating a new entry. In the subroutine, you're checking if the `SPRAS` field is empty and throwing an error message accordingly. You can test the implementation by opening the table in Transaction `SM30` or using the menu path `UTILITIES • CREATE ENTRIES` from the table maintenance screen, as shown in [Figure 10.23](#).
- Click the NEW ENTRIES button on to create a record. If the `LANGUAGE` key field is left empty, it will show the error message on the NEW ENTRIES screen (see [Figure 10.24](#)).

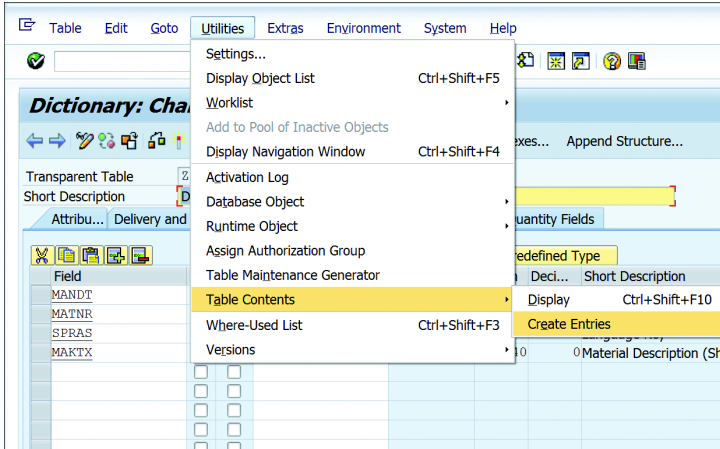


Figure 10.23 Create Entries

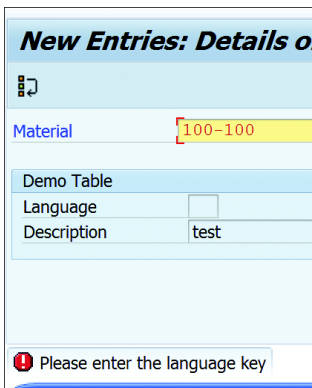


Figure 10.24 Error Message Triggered by Subroutine

In this section, we discussed the steps for creating a maintenance view for a database table. In the next section, we'll look at using foreign keys to define the relationships between database tables.

10.1.4 Foreign Keys

Foreign keys define the relationship between tables. If a field of one table exists as a key field in another table, then the tables are said to be in a *foreign key relationship*. The table in which the fields exist as keys is called the *check table*, and the other table is called the *foreign key table*.

In [Figure 10.25](#), the PLANETYPE field in table SFLIGHT exists as a key field in table SAPLANE. Here, table SFLIGHT is the foreign key table and table SAPLANE is the check table.

Foreign Key Table

Check Table

Dictionary: Display Table			
Transparent Table: SFLIGHT Active			
Short Description: Flight			
Attribu... Delivery and Maintenance Fiel... Entry help			
Field	Key	Ini...	Data element
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S MANDT
CARRID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S CARR_ID
CONNID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S CONN_ID
FLDATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S DATE
PRICE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S PRICE
CURRENCY	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S CURRCODE
PLANETYPE	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S PLANETYPE
SEATSMAX	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SEATSMAX
SEATSOCC	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SEATSOCC
PAYMENTSUM	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SUM
SEATSMAX B	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SMAX B
SEATSOCC B	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SOCC B
SEATSMAX F	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SMAX F
SEATSOCC F	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SOCC F

Dictionary: Display Table			
Transparent Table: SAPLANE Active			
Short Description: Plane			
Attribu... Delivery and Maintenance Fiel... Entry help			
Field	Key	Ini...	Data element
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S MANDT
PLANETYPE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	S PLANETYPE
SEATSMAX	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SEATSMAX
CONSUM	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S CONSUM
CON UNIT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S CONUNIT
TANKCAP	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S CAPACITY
CAP UNIT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S CAPUNIT
WEIGHT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S PLAN WEI
WEI UNIT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S WEI UNIT
SPAN	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S SPAN
SPAN UNIT	<input type="checkbox"/>	<input checked="" type="checkbox"/>	S UNIT

Figure 10.25 Foreign Key Relationship

Using the check table, automatic input validations can be performed for the field in the foreign key table. This ensures data integrity, and no invalid data goes into the system. For example, if you have a custom table with the MATNR field and you want to ensure that the user must input only the valid material numbers maintained in the material master table MARA, you can create a foreign key relationship with MARA as the check table.

When the user inputs the material number in your custom table, the system will automatically check if the record exists in table MARA. Note that the foreign key relationship is created only in ABAP Data Dictionary and not in the database. The check is performed against the check table only when the user inputs the record through a screen. The check is not performed when the records are updated using Open SQL statements. While programming an Open SQL statement, the check table should be considered manually and checked in the program.

The following steps allow you to create a foreign key relationship:

1. On the table maintenance screen, select the table field and click the FOREIGN KEYS icon, as shown in [Figure 10.26](#).

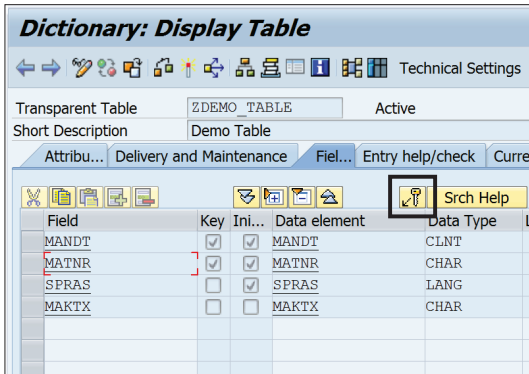


Figure 10.26 Foreign Key

2. If a value table is maintained at the domain of the field, the system will show a prompt asking whether to create a proposal based on the value table. A *value table* is maintained at the domain (see [Figure 10.27](#)) that is attached to the data element; it specifies the possible entries for the field but does not perform any checks.

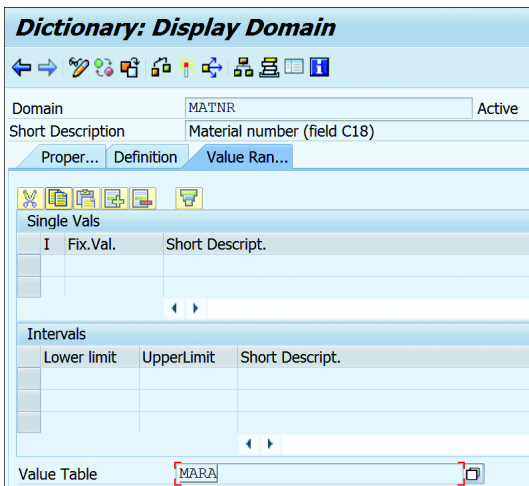


Figure 10.27 Value Table Assigned for Domain

Check tables perform checks on the field input. In most cases, the value table and check table will be same; otherwise, the value table may propose a value as a possible entry and the check table may reject that entry. You can accept the system proposal or maintain any other table as a check table.

3. On the CHANGE FOREIGN KEY... screen (see [Figure 10.28](#)), you have the following options:
 - ▶ CHECK TABLE: Enter the check table name and click GENERATE PROPOSAL to list the fields of the check table that can be mapped to the foreign key table.
 - ▶ GENERIC: If you have multiple fields in the foreign key table that match the key fields in the check table, you can select this checkbox to ignore the input validation for the field.
 - ▶ CONSTANT: You can assign a constant value for other fields so that the system will check if the field in the check table has the assigned constant value for the input validations.
 - ▶ SCREEN CHECK: Select this checkbox if you want the system to perform input validations for the foreign key field against the check table. Optionally, you can also assign a custom error message.
 - ▶ SEMANTIC ATTRIBUTES: Optionally, you can maintain semantic attributes while creating the foreign key relationship. These are primarily for documentation purposes.
 - ▶ FOREIGN KEY FIELD TYPE: Optionally, the type of foreign key fields can be defined to describe the meaning of the foreign key fields in the foreign key table. There are a few options:
 - NON-KEY FIELDS/CANDIDATE: Indicates if the fields of the foreign key table are non-key fields that don't uniquely identify the row.
 - KEY FIELDS/CANDIDATE: Indicates if the fields of the foreign key table are key fields that uniquely identify the row.
 - KEY FIELDS OF A TEXT TABLE: If this option is selected, the foreign key table is handled as a text table of the check table. The primary key of the foreign key table must match the check table, plus a language key field with the type LANG should exist. One check table can have only one text table. The entries in the text table are used for input helps for screen fields and other such purposes.

- ▶ **CARDINALITY:** A n:m cardinality can be defined for the foreign key table. Cardinality specifies, for example, for each record in check table, how many records can exist in the foreign key table. Use **F4** help to select possible options. The short text for each option explains the cardinality sufficiently.

4. Click **COPY** to save the foreign key relationship.

Change Foreign Key ZDEMO_TABLE-MATNR

Short text

Check table: MARA Generate proposal

Check table	ChkTablFld	For.key table	Foreign Key Field	Generic	Constant
MARA	MANDT	ZDEMO_TABL.MANDT		<input type="checkbox"/>	
MARA	MATNR	ZDEMO_TABL.MATNR		<input type="checkbox"/>	

Screen check

Check required Error message MsgNo AArea

Semantic attributes

Foreign key field type

- Not Specified
- Non-key-fields/candidates
- Key fields/candidates
- Key fields of a text table

Cardinality :

Copy Help Up Down Delete Close

Figure 10.28 Maintain Foreign Key Relationship

10.1.5 Include Structure

If you want to use fields from an existing structure or if you're creating multiple tables with common fields that should be maintained centrally, you can add the fields of the structure to the table by including the structure.

The following requirements must be met to include a structure:

- ▶ The structure to be included in the table should be a flat structure.
- ▶ The field names in the structure should not be more than sixteen characters in length.

- ▶ The structure can't have any fields that already exist in the table, because once the structure is included in the table, the fields of the structure are considered to be part of the table.

Follow these steps to include a structure in table:

1. On the table maintenance screen, place your cursor on the field where you want to include the structure. Follow the menu path EDIT • INCLUDE • INSERT, as shown in [Figure 10.29](#).

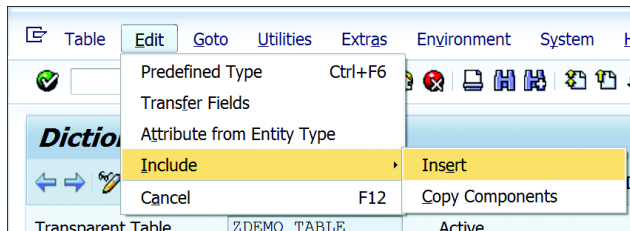


Figure 10.29 Include Structure

2. In the INSERT INCLUDE window shown in [Figure 10.30](#), enter the name of the STRUCTURE. Optionally, you can input a GROUP name and NAME SUFFIX. If the group name is provided, the components of the structure can be addressed from programs once or individually using that group name. If the name suffix is provided, the fields of the structure will be suffixed with the entered value to avoid any naming conflicts with the existing fields of the table.
3. Click OK to insert the structure and activate the table.

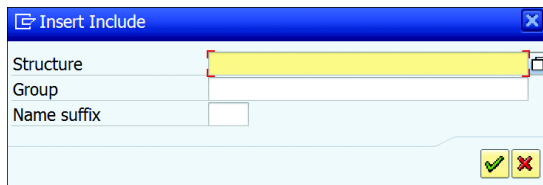


Figure 10.30 Insert Include Window

4. You can display the fields of the include structure by using the EXPAND and COLLAPSE button (see [Figure 10.31](#)). An include structure can be included anywhere in the table.

Field	Key	Ini...	Data element	Data Type
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT
MATNR	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MATNR	CHAR
SPRAS	<input type="checkbox"/>	<input checked="" type="checkbox"/>	SPRAS	LANG
MAKTX	<input type="checkbox"/>	<input type="checkbox"/>	MAKTX	CHAR
.INCLU-A	<input type="checkbox"/>	<input type="checkbox"/>	ZARB	STRU

Figure 10.31 Expand/Collapse Include Structure

10.1.6 Append Structure

Append structures are used to enhance standard SAP tables by adding your own fields. *Append structures* are specific to a table and are added at the bottom of the table. You can have more than one append structure for a table, but the append structure of one table can't be used in another table.

The append structure is created in the customer namespace so that it won't be overwritten in an upgrade. The fields in the append structure should also be in the customer namespace and should start with YY or ZZ. If the new version of the standard table is imported after an upgrade, the fields of the append structure are added to the standard table.

As noted, an append structure can be assigned only to one table. If you want similar fields in multiple tables, you can create a structure with the fields and include the structure inside an append structure. You can then create append structures for each table and include the include structure.

The following steps will walk you through adding an append structure:

1. Open the table in change mode in ABAP Data Dictionary.
2. Click the APPEND STRUCTURE button in the application toolbar, shown in [Figure 10.32](#).
3. In the window that appears (see [Figure 10.32](#)), provide the append structure name in the customer namespace.
4. Maintain the components of the append structure. The field names should start with ZZ or YY, as shown in [Figure 10.33](#).

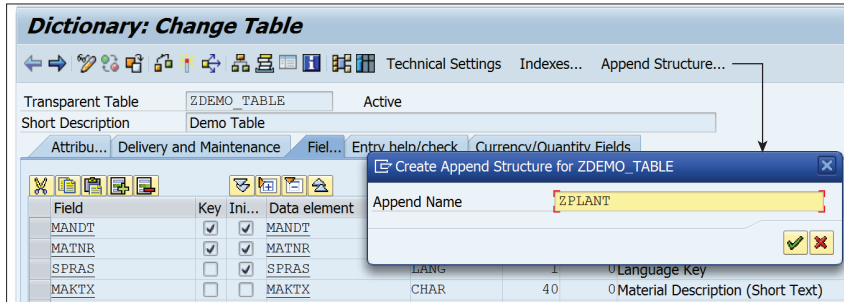


Figure 10.32 Adding an Append Structure

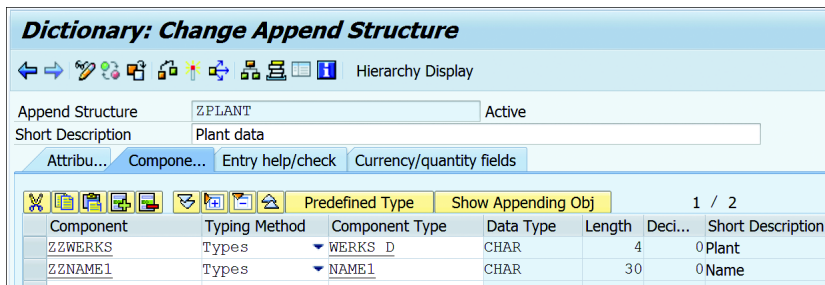


Figure 10.33 Creating Append Structure

5. Activate the append structure and click BACK to return to the table. The append structure will be shown at the end of the fields, as in Figure 10.34. You can click the EXPAND and COLLAPSE buttons in the toolbar to show the append structure fields or hide them from view.

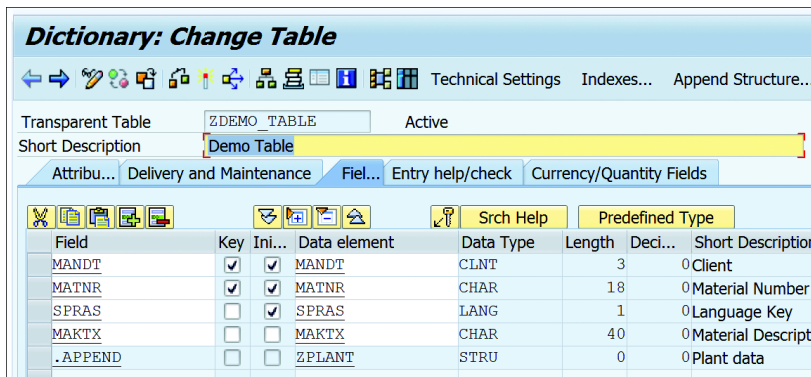


Figure 10.34 Append Structure in Table

Note

A database table can't have long fields (data types `VARC`, `LCHR`, or `LRAW`), because these long fields should always be at the end of the table.

In this section, we discussed creating, using, and maintaining database tables and their components. In the next section, we'll shift our focus to using views in ABAP Data Dictionary.

10.2 Views

Application data is often spread across multiple tables; you can define *views* to combine this data. The lists of database tables accessed by a view are called *basis tables*. If the view contains more than one basis table, the basis tables can be joined using *joins*. The structure of the view is made from the fields from the basis tables that are combined. A view is not created physically in the database, so a view doesn't store data on its own. A view can be used in ABAP programs for data selection or as a structure type in data declarations.

The following are different general types of views that can be created in ABAP:

- ▶ **Database views**
Allow you to combine multiple tables and link them using join conditions.
- ▶ **Projection views**
Allow you to create a projection of one table in which fields that aren't required can be hidden, thus simplifying the interface.
- ▶ **Maintenance views**
Allow you to join related tables for maintenance.
- ▶ **Help views**
Used for search helps.
- ▶ **ABAP CDS views**
CDS views form an infrastructure that can be used to create the underlying (persistent) data model in SAP HANA. A CDS view is an entity that isn't persistent and is defined as a projection of other entities. CDS views were originally available only with SAP HANA, but with SAP NetWeaver 7.4 SP 5, ABAP CDS views were introduced. ABAP CDS views implement the general CDS concept of SAP NetWeaver for AS ABAP. Once the ABAP CDS view is defined, it

becomes part of ABAP Data Dictionary and can be accessed from ABAP programs similarly to other ABAP Data Dictionary objects. ABAP CDS views help to push certain resource-intensive operations (for example, performing aggregate functions) down to the database level (called *code pushdown*; see [Chapter 5, Section 5.3.1](#)).

The database view, maintenance view, help view, and projection view can be defined using ABAP Data Dictionary (Transaction SE11) and are called *classic views*. ABAP CDS views can be defined only by using ABAP in Eclipse.

10.2.1 Database Views

A *database view* is a general view for one or more tables. If more than one table is used in a view, they can be joined using an inner join.

The following steps detail how to create a database view:

1. On the initial ABAP Data Dictionary screen, select the VIEWS radio button, input the name of the view, and click the CREATE button. Note that the view name should be in the customer namespace.
2. A dialog window appears in which you can select the type of view, as shown in [Figure 10.35](#). You're creating a database view here, so select the DATABASE VIEW radio button and click the green checkmark.

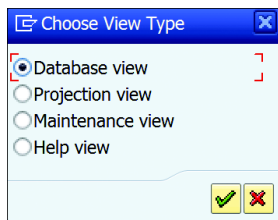


Figure 10.35 View Type

3. Under the TABLE/JOIN CONDITIONS tab, enter the database table names under the table column that you wish to join. You only can use transparent tables as the base tables in a database view. Under JOIN CONDITIONS, write the condition to join the tables, as shown in [Figure 10.36](#). Include the MANDT field in the join condition to restrict the data to the current client. Only inner joins are allowed.

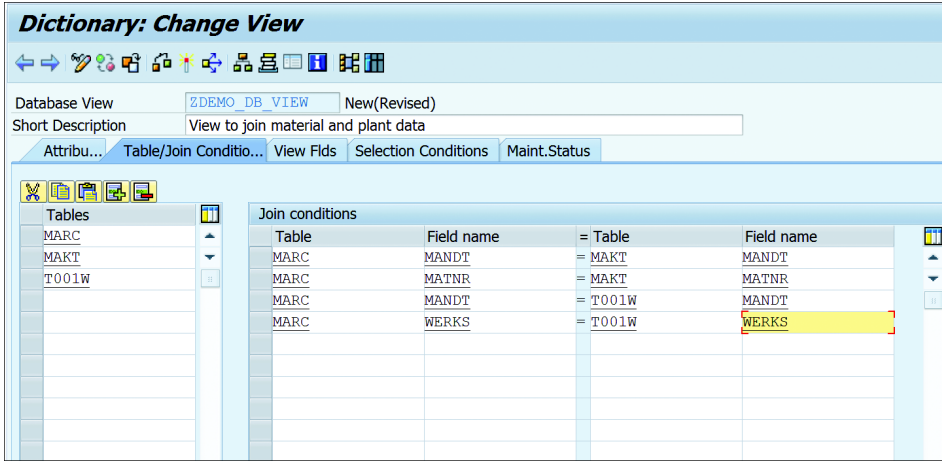


Figure 10.36 Join Conditions

- Click on the VIEW FIELDS tab. Use the TABLE FIELDS button to select the fields that you want to show in the view, as shown in Figure 10.37. You can select the fields from the basis tables that are listed in the TABLE/JOIN CONDITIONS tab.

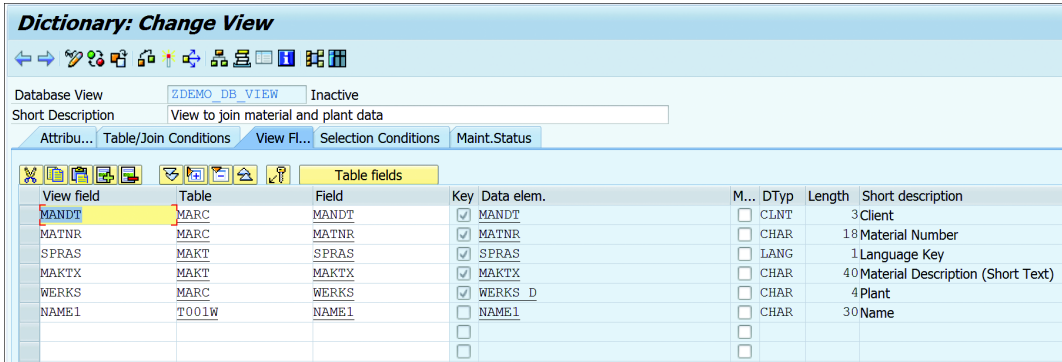


Figure 10.37 View Fields

- Optionally, you can maintain the select condition under SELECT CONDITIONS tab (see Figure 10.38).
- You can set the maintenance status under the MANT.STATUS tab. Database views with more than one basis table only can be read.
- Activate the view by clicking the ACTIVATE button in the application toolbar.

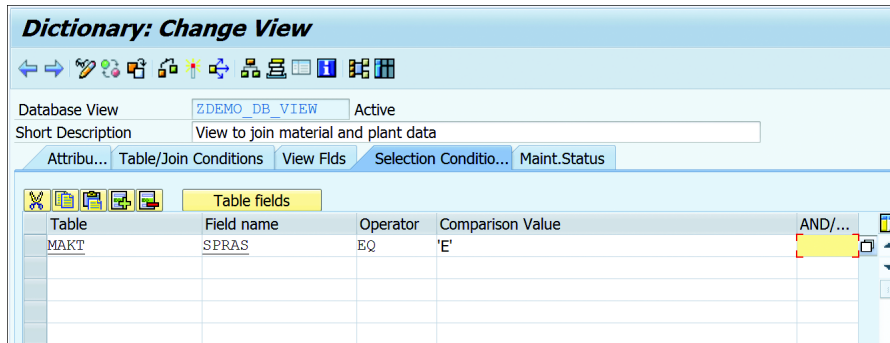


Figure 10.38 Select Conditions

8. You can execute the view by clicking on the CONTENTS icon in the application toolbar, as shown in [Figure 10.39](#).

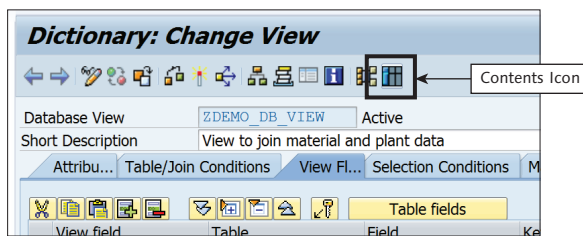


Figure 10.39 Contents Icon

10.2.2 Projection Views

Projection views are the simplest form of view. They can be used to hide unwanted fields from the table to simplify the interface. Projection views can't be used to join multiple tables; they're used to create a new projection for an existing table.

To create a projection view, follow these steps:

1. Enter the name of the view on the initial screen of ABAP Data Dictionary and click CREATE.
2. In the VIEW TYPE window (shown earlier in [Figure 10.35](#)), select PROJECTION VIEW.
3. On the DICTIONARY: CHANGE VIEW screen (see [Figure 10.40](#)), maintain a short description for the view. Enter the name of the base table in the BASE TABLE field. Use the TABLE FIELDS button to select the fields for the view.

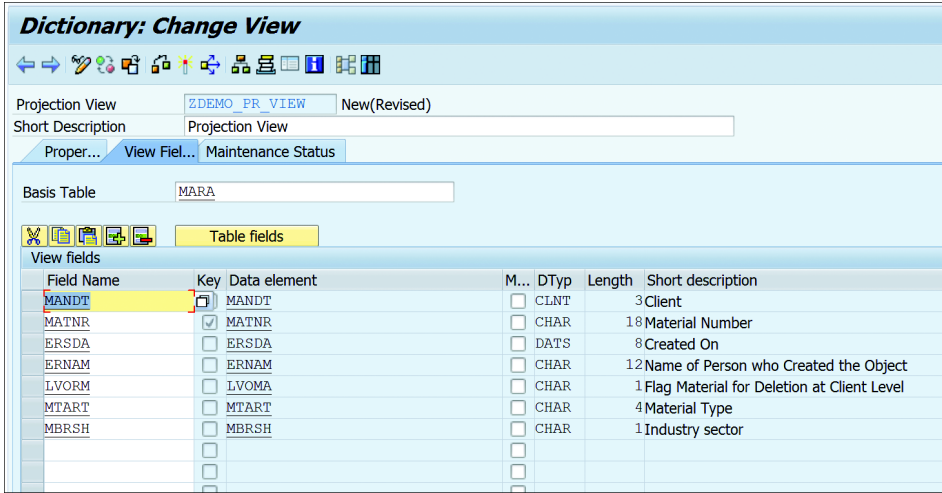


Figure 10.40 Projection View

- Under the MAINTENANCE STATUS tab, select the ACCESS type as either READ ONLY or READ AND CHANGE, as shown in Figure 10.41.
- If the READ AND CHANGE option is set, you can select DISPLAY/MAINTENANCE ALLOWED for the TABLE VIEW MAINTENANCE field to allow the user to manually maintain the data.

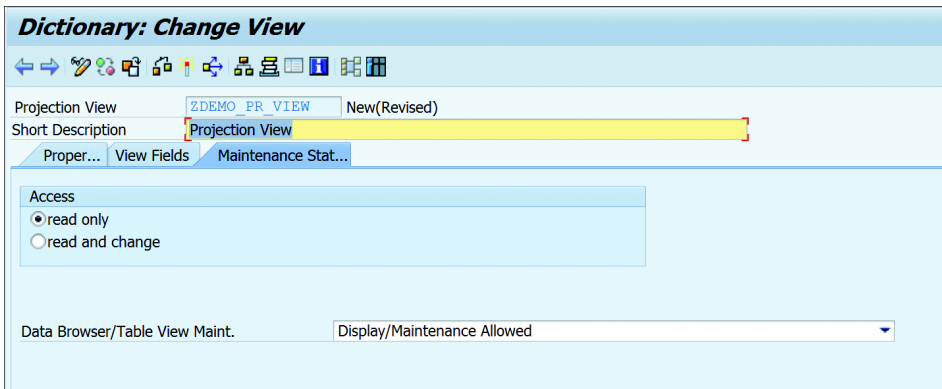


Figure 10.41 Maintenance Status

- Activate the view by clicking the ACTIVATE button in the application toolbar.

10.2.3 Maintenance Views

Maintenance views are special views used to perform writes on multiple tables at once. You can use one maintenance view to modify data of multiple related database tables. Unlike other views, maintenance views can't be accessed using Open SQL statements.

To create a maintenance view, proceed with following steps:

1. Select MAINTENANCE VIEW in the VIEW TYPE window (see [Figure 10.35](#)).
2. Creating a maintenance view is similar to creating a database view. However, in maintenance views, only related tables can be joined. Select join tables by clicking the RELATIONSHIPS button under the TABLE/JOIN CONDITIONS tab in the view maintenance screen, as shown in [Figure 10.42](#).

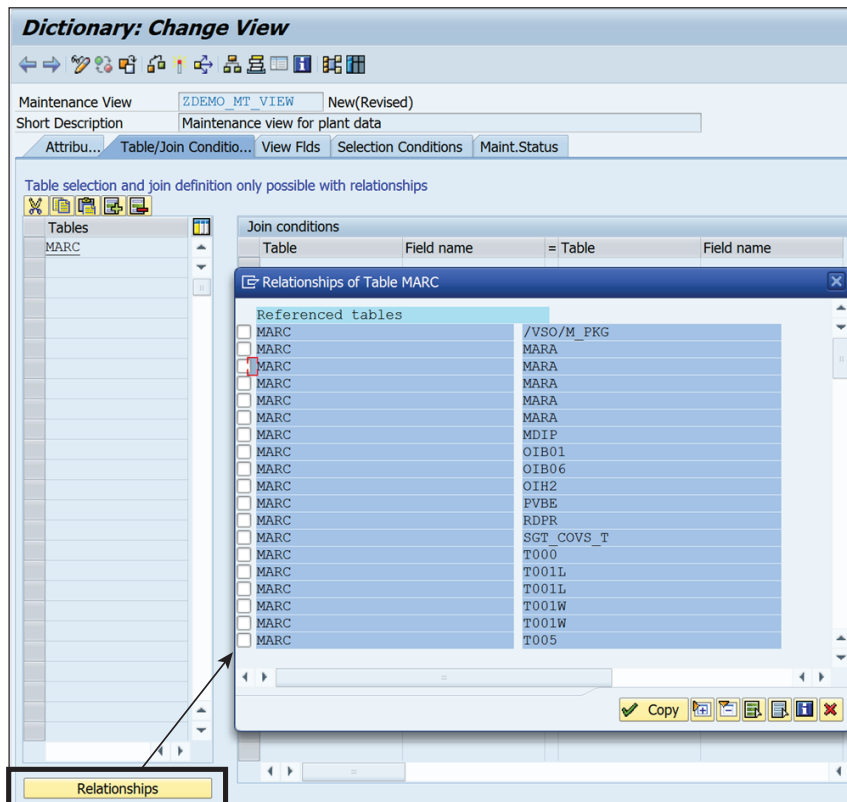


Figure 10.42 Maintenance View

3. Enter the primary table name and click the RELATIONSHIPS button to select secondary tables. Additional tables can be selected by keeping the cursor on the secondary table and using the RELATIONSHIPS button. You can't maintain the join condition manually.

The view key in a maintenance view can be divided into a time-dependent area and a non-time-dependent area. The time-dependent area can contain date fields and time fields. This means that the rows of these views have chronological importance.

In the maintenance view, if a database table is joined with its text table, the logon language is selected automatically for this table.

4. Under the MAINTENANCE STATUS tab, you can select the access type, as shown in [Figure 10.43](#):

- ▶ READ ONLY: The view only reads the data.
- ▶ READ, CHANGE, DELETE AND INSERT: The view can be used to change, delete, or insert data.
- ▶ READ AND CHANGE: The view can be used to change the data, but the records can't be deleted or inserted.
- ▶ READ AND CHANGE (TIME-DEPENDENT VIEWS): The view can be used to insert only entries for which the non-time-dependent part of the key doesn't differ from existing entries.

The data can be maintained using Transaction SM30.

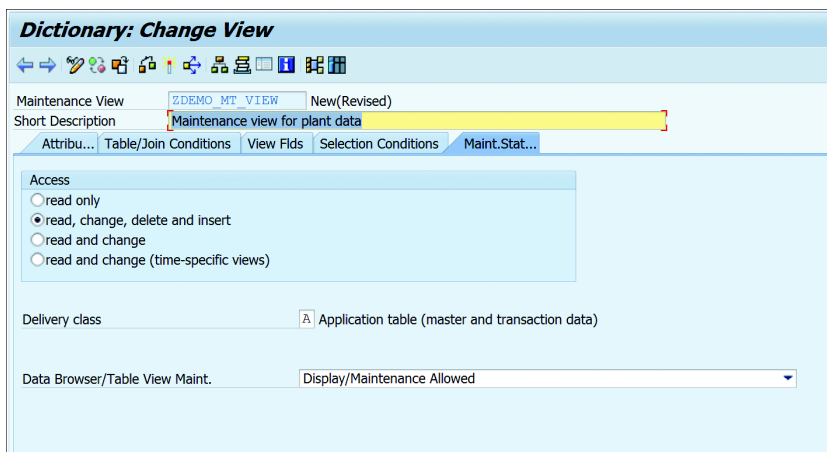


Figure 10.43 Maintenance Status

Along with the maintenance status of the view, a status can also be maintained for view fields, as shown in [Figure 10.44](#). The available options are as follows:

- ▶ **Standard (blank)**: This is the blank field. There are no restrictions on the view field.
- ▶ **Read-only (R)**: The view can't be used to perform writes on the view field.
- ▶ **Subset (S)**: Only a subset of the data is displayed. This subset is defined by entering an appropriate value in this field.
- ▶ **Hidden (H)**: A field of this type is hidden in table maintenance and can't be edited. The content is initialized implicitly.

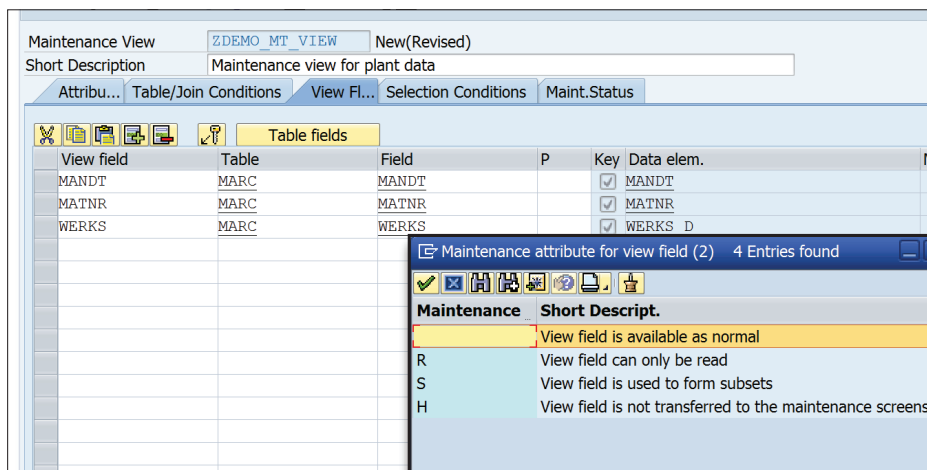


Figure 10.44 Maintenance Status of Field

10.2.4 Help Views

Help views are used as selection methods in search helps. You can create a help view if you want to display data from multiple tables in the search help. A database view should be used in search helps if you want to join multiple tables using inner join, but if you need to use an outer join to join tables, you need to create a help view.

The steps to create a help view are similar to those for a maintenance view in which only related tables can be joined. However, help views can't be used to maintain data in tables.

We'll discuss search helps in [Section 10.6](#).

10.2.5 ABAP CDS Views

ABAP Data Dictionary doesn't support defining ABAP CDS views. To define a CDS view, we need to use ABAP in Eclipse.

Defining ABAP CDS Views

To define ABAP CDS views, proceed with the following steps:

1. If you haven't done so yet, set up the Eclipse IDE and install the ABAP Development Tools (ADT) as explained in [Chapter 3, Section 3.3](#).
2. In the Eclipse IDE, create a new project (or use existing project). Chose the package in which you want to create the ABAP CDS view. Right-click the package and select **NEW • OTHER ABAP REPOSITORY OBJECT**, as shown in [Figure 10.45](#).

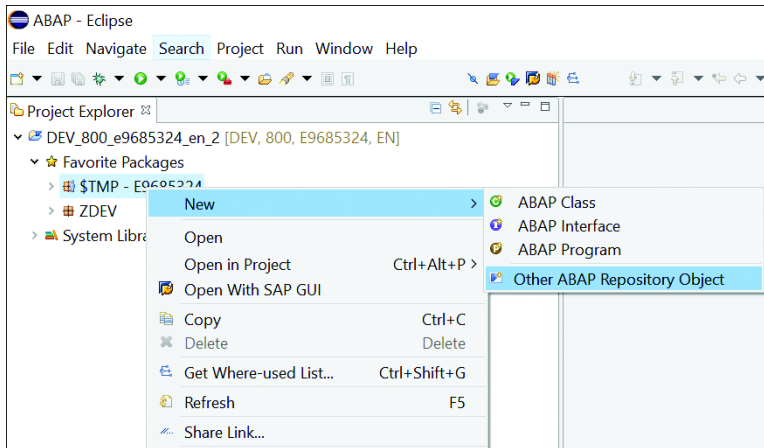


Figure 10.45 Create New Repository Object

3. In the **NEW ABAP REPOSITORY OBJECT** window, expand the **DICTIONARY** folder and select **DATA DEFINITION**. You can also search for "data definition", as shown in [Figure 10.46](#). Click **NEXT**.
4. In the **NEW DATA DEFINITION** window, enter the name and description for the ABAP CDS view, as shown in [Figure 10.47](#). Click **FINISH**.

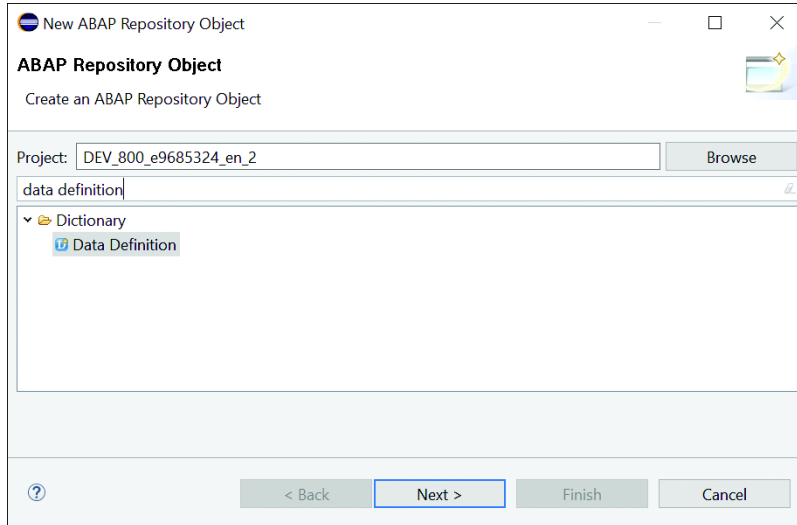


Figure 10.46 Data Definition

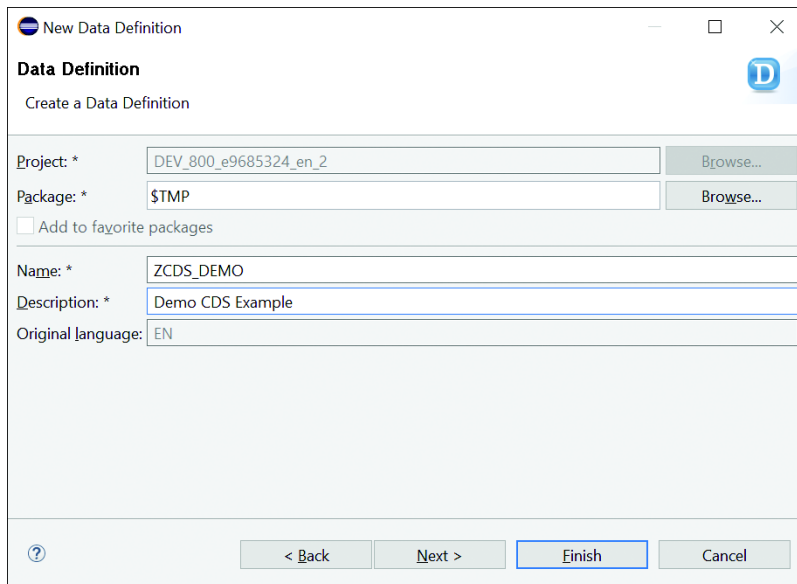


Figure 10.47 Create Data Definition

5. The new CDS view editor opens (see [Figure 10.48](#)). The editor automatically loads a template, which you need to edit to supply information. The template

code is shown in [Listing 10.3](#). From this template, replace 'sql_view_name' with the name of the view and data_source_name with the name of the database table.

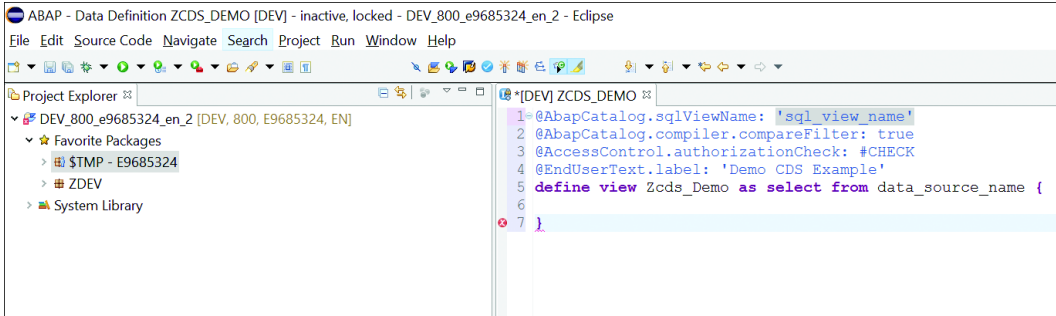


Figure 10.48 CDS View Editor

```
@AbapCatalog.sqlViewName: 'sql_view_name'
@AbapCatalog.compiler.compareFilter: true
@AccessControl.authorizationCheck: #CHECK
@EndUserText.label: 'Test'
define view Zcds_Demo1 as select from data_source_name {
}
```

Listing 10.3 CDS View Template

6. You can specify database table fields within the curly braces. For example, if you want to define the "ZDEMO_V1" view for database table VBRK and list certain fields from the table, the template can be edited as shown in [Figure 10.49](#).

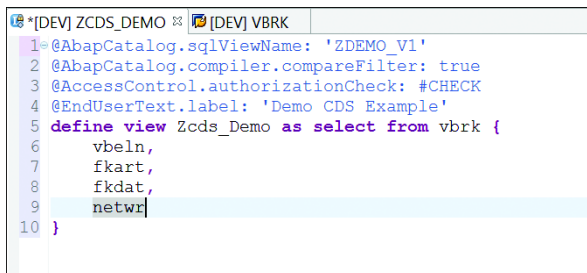
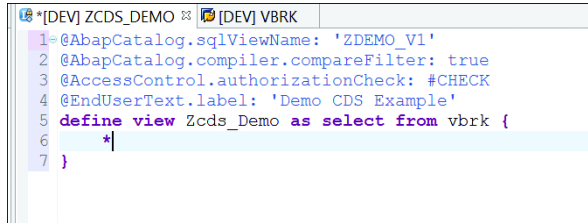


Figure 10.49 Editing Template to List Certain Fields from Table

7. To list all the fields from the table, use * within the curly braces, as shown in [Figure 10.50](#).



```

1=@AbapCatalog.sqlViewName: 'ZDEMO_V1'
2=@AbapCatalog.compiler.compareFilter: true
3=@AccessControl.authorizationCheck: #CHECK
4=@EndUserText.label: 'Demo CDS Example'
5 define view Zcds_Demo as select from vbrk {
6     *
7 }

```

Figure 10.50 Listing All the Fields from Table

8. Activate the view by clicking the ACTIVATE button in toolbar or by pressing **CTRL + F3**.

Accessing ABAP CDS Views

After following the preceding steps, you've successfully defined a CDS view. This view is now available as an ABAP Data Dictionary object (see [Figure 10.51](#)) and can be accessed from ABAP programs.

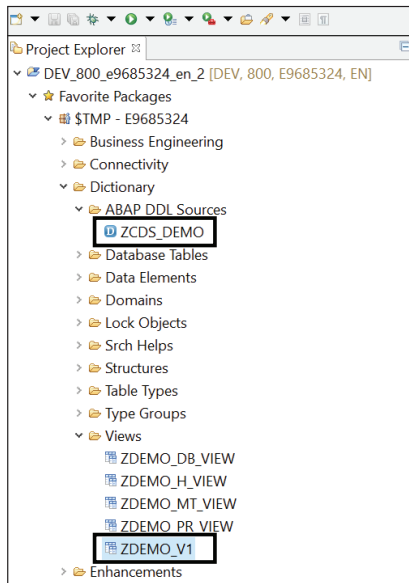


Figure 10.51 View Defined in ABAP Dictionary

You can access ABAP CDS views in ABAP programs as an ABAP type or use Open SQL statements to fetch the data.

Defining ABAP CDS Views as Replacement Objects

An ABAP CDS view can be defined as a *replacement object* for transparent tables and classic database views. When the CDS view is defined as a replacement object, all read access to the transparent table or database view will be redirected to the ABAP CDS view, using Open SQL. The replacement object will also be used for any foreign key checks for screen fields. All other access, such as write access or Native SQL statements, will access the original table or classic database view.

An ABAP CDS view defined as a replacement object should have the same structure (same number of components with the same name and technical attributes for each component) as the original object to which the CDS view is attached as a replacement object. You can check if a transparent table or classic database view is assigned a replacement object by opening the transparent table or classic database view in ABAP Data Dictionary and following the menu path EXTRAS • REPLACEMENT OBJECT.

In customer systems, an ABAP CDS view replacement object can be defined only for classic database views that are defined by customers in a customer namespace and can't be defined for transparent tables. SAP might deliver a replacement object for transparent tables, but it's not recommended to define your own replacement objects in customer systems. Replacement objects can't be defined for GTTs.

Replacement objects are intended for existing aggregate tables that aggregate data from other tables. Using an ABAP CDS view as a replacement object for aggregate tables allows you to aggregate the data for every read using the ABAP CDS view instead of selecting the data in an ABAP program for aggregation and writing it into the table. ABAP CDS view replacement objects help to improve performance by aggregating data in the database for tables with large volumes of data without invalidating the existing programs.

In this section, we discussed how to create, define, use, and access different views. In the next section, we'll turn our attention to maintaining different data types in ABAP Data Dictionary.

10.3 Data Types

By using the DATA TYPES radio button in the ABAP Data Dictionary initial screen, we can create global data types such as data elements, structures, and table types.

You've seen how to create these global types at various points in previous chapters, so this section will focus on the additional options that can be maintained for these global types centrally and can then be inherited into programs automatically.

10.3.1 Data Elements

A *data element* describes an elementary data type or a reference data type. It allows you to maintain both technical and semantic attributes. *Semantic attributes* are significant for screen fields that refer to this data element. For example, the screen field can automatically import input help (**F4** help) or field help (**F1** help) from a data element. For ABAP data objects that refer to the data element, the semantic attributes hold no significance.

For an elementary data type, technical specifications are derived from a domain or directly maintained in the data element. Similarly, for reference data types, a reference to a global class or interface can be maintained or a reference to a predefined type can be maintained.

Figure 10.52 shows the DATA TYPE tab of the data element, where the technical attributes can be maintained by selecting the relevant radio button (i.e., ELEMENTARY TYPE, PREDEFINED TYPE, or REFERENCE TYPE).

Dictionary: Change Data Element

← → ↻ 🔄 📄 📖 📄 📖 Documentation Supplementary Documentation

Data element: ZDEMO_DE New(Revised)
 Short Description: Demo data element

Attribu... Data Ty... Further Characteristics Field Label

Elementary Type

Domain Data Type [] Length 0 Decimal Places 0

Predefined Type Data Type [] Length 0

Reference Type

Name of Ref. Type []

Reference to Predefined Type Data Type [] Length 0

Figure 10.52 Data Type Tab

There are various semantic attributes that can be maintained for the data elements. Under the DATA TYPE tab and in the application toolbar of the DICTIONARY: CHANGE DATA ELEMENT page, the following attributes can be maintained:

► SHORT TEXT

Short text can be maintained for the data element under the DATA TYPE tab. The short text appears as a header in the field texts for screen fields and table/structure fields.

► DOCUMENTATION

Documentation can be maintained for a data element, to be shown automatically as **[F1]** help for screen fields that refer to this data element. To maintain documentation, click the DOCUMENTATION button in the application toolbar.

A documentation status can be set by following the menu path GoTo • DOCUMENTATION • STATUS. This path will open a window with self-explanatory options, as shown in [Figure 10.53](#).

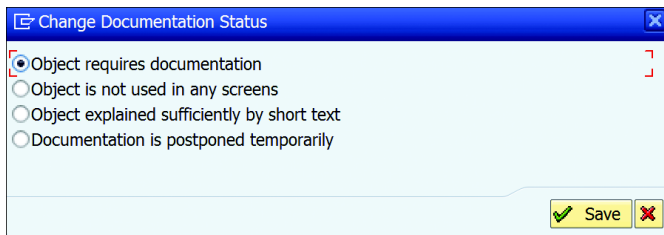


Figure 10.53 Documentation Status

► SUPPLEMENTARY DOCUMENTATION

Additional supplementary documentation can be created for the data element to be assigned to specific program screen fields as field help. Each piece of supplementary documentation is assigned a four-digit number. Click the SUPPLEMENTARY DOCUMENTATION button in the application toolbar to maintain the supplementary documentation.

Under the FURTHER CHARACTERISTICS tab, as shown in [Figure 10.54](#), the following options can be maintained:

► SEARCH HELP

A search help can be assigned to a data element with an elementary data type; it will be available automatically as **[F4]** help to screen fields that refer to this data element. A search help can't be assigned to a reference type.

▶ PARAMETERS

A parameter that is passed to the search help is entered in this field. The help for this field can be used to select the parameter based on the assigned search help.

▶ PARAMETER ID

The data element can be linked with SAP memory using the SPA/GPA parameter ID. SAP memory is the memory area that is accessible by all SAP GUI sessions in the user session. SAP memory can be used to exchange data between programs running in different sessions (GUI windows). You'll learn more about memory organization in [Chapter 13](#). The PARAMETER ID is only used if the data element is an elementary data type. The screen fields that refer to this data element are automatically filled with the contents of the parameter from SAP memory unless the field itself is manually associated with a PARAMETER ID. Data elements of a reference type can't be linked to the PARAMETER ID.

▶ DEFAULT COMPONENT NAME

For table and structure fields, a naming suggestion can be maintained to be proposed when the structure or table field references this data element. English names are generally maintained as default component names (for example, MATERIAL for MATNR). Structures (such as BAPIs) that are used in APIs should follow this convention.

▶ CHANGE DOCUMENT

This flag specifies whether changes to fields in database tables should be logged by change objects when the field refers to this data element.

▶ NO INPUT HISTORY

This flag determines if the input history in SAP GUI is active for screen fields that refer to this data element. This allows you to turn off SAP GUI history for sensitive fields.

▶ BI-DIRECTIONAL OPTIONS

This option defines the behavior of screen fields when using bidirectional fonts such as Hebrew or Arabic. There are two options:

- ▶ BASIC DIRECTION IS SET TO LTR: This sets the writing direction as left to right.
- ▶ NO BIDI FILTERING: This option sets the bidirection filtering that prevents the Unicode formatting characters that specify the writing direction of bidirectional fonts in data transport between SAP GUI and the application server.

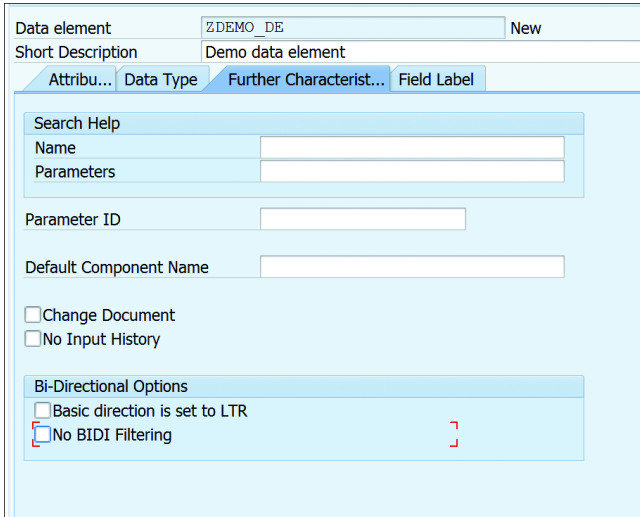


Figure 10.54 Further Characteristics

Most of the semantic attributes of data elements are significant for UI elements (screen elements), but some may be useful for database fields (such as documentation). We recommend creating separate data elements for UI fields and database table fields.

Field labels can be maintained for the data element under the FIELD LABEL tab (see Figure 10.55); these labels are used as column headings for lists on screens. You can maintain short, medium, and long headings, plus a header with a freely definable length field.

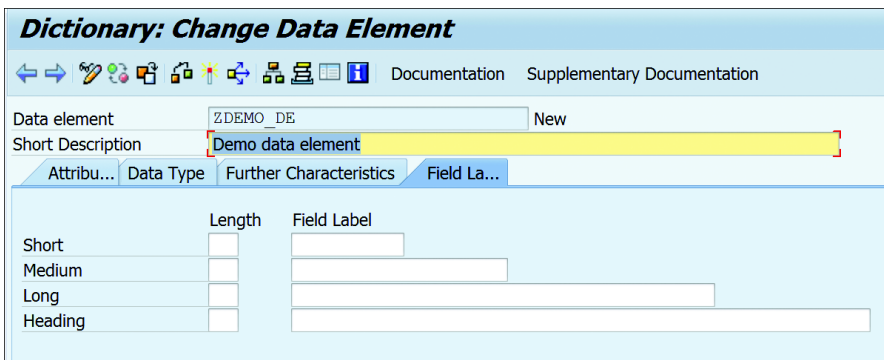


Figure 10.55 Field Label

10.3.2 Structures

We discussed creating structures in [Chapter 5](#), and most of the steps involved in creating structures are similar to those for creating database tables, discussed earlier in this chapter in [Section 10.1.1](#).

A structure can include elementary data types, other structures, or table types. Structures can be nested to any depth. Structures allow you to centrally maintain the fields when similar fields are used in multiple objects (e.g., programs, function module interfaces, etc.).

The components of a structure can include elementary types, reference types, or boxed types, as shown in the available typing methods in [Figure 10.56](#).

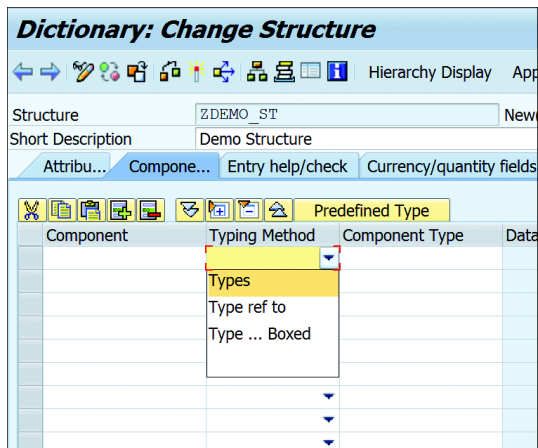


Figure 10.56 Typing Method

The following list defines each of these types:

► **Elementary types**

Elementary types can be defined by referring to a data element from which the type can take the technical and semantic attributes or to a predefined type for which the data type, length, and any decimals are manually assigned.

► **Referenced types**

These can be defined as either data references or object references. Data references can be specified by using a data element that defines a data reference or by using a predefined ABAP Data Dictionary type, length, and any decimals.

Object references can be specified by using a data element with a global class or interface reference.

► **Boxed types**

These can be defined in the structure as of SAP NetWeaver 7.0 EHP 2. Boxed types help to reduce the memory footprint in ABAP programs and improve performance. When a boxed type is used, the system doesn't allocate initial memory. Memory is allocated to one structure, and all data of a similar type points to that memory, unlike a structure with elementary types in which each data object of the type takes up its own initial memory space. This is called *initial value sharing*. Memory is shared across multiple data objects that use this type.

Reference types don't take any initial memory space, so boxed types fall somewhere between elementary types and reference types. In other words, the initial memory of boxed types is much lower than elementary types but greater than reference types.

Currently, only *static boxes* are supported. Static types are boxed components for which components are known statically. Only a substructure of a structure or attributes of classes or interfaces can be defined as boxed. Boxed types are defined in programs by adding `BOXED` to the `TYPES` statement, as shown in [Listing 10.4](#).

```
TYPES : BEGIN OF ty_lfal
        Lfal TYPE lfal BOXED,
      END OF ty_lfal.
```

Listing 10.4 Defining Boxed Component Using TYPES

The efficiency of a boxed structure is due to the initial value sharing in which all occurrences of similar data are referenced to the same initial memory. However, the initial value sharing is revoked under the following circumstances:

- Write access is made to a static box or its components.
- The static box or its components are assigned to a field symbol.
- The static box or its components are addressed using a data reference.
- The static box or its components are used as an actual parameter of a procedure.

Once the initial value sharing is revoked, the internal reference will reference an instance of the structure.

Boxed types reduce the memory requirement of structures that are used more than once. For example, if an internal table has a substructure, then each row will increment the memory store even if the row is initial. If the substructure is defined as boxed type, then it won't require multiple memory stores, because each row can point to the same initial memory as long as only read accessed is permitted.

[Figure 10.57](#) shows the initial memory allocation of structures, as shown in the debugger window, using the normal types, boxed types, and reference types defined in [Listing 10.5](#). In [Listing 10.5](#), we've defined three structures of each type to highlight how the memory is allocated individually when the object of the same type is repeated.

As you can see in [Figure 10.57](#), boxed types have a much smaller memory footprint than normal types. Boxed types are deep structures—that is, structures that contain table types.

```
*Normal type
TYPES : BEGIN OF ty_kna1,
      kna1 TYPE kna1,
END OF ty_kna1.
*Boxed type
TYPES : BEGIN OF ty_kna1_b,
      kna1 TYPE kna1 BOXED,
END OF ty_kna1_b.
*Data declarations using normal type
DATA : st_kna1_1 TYPE ty_kna1.
DATA : st_kna1_2 TYPE ty_kna1.
DATA : st_kna1_3 TYPE ty_kna1.
*Data declarations using boxed type
DATA : st_kna1_b_1 TYPE ty_kna1_b.
DATA : st_kna1_b_2 TYPE ty_kna1_b.
DATA : st_kna1_b_3 TYPE ty_kna1_b.

*Data declaration using reference type
DATA : st_kna1_o_1 TYPE REF TO ty_kna1.
DATA : st_kna1_o_2 TYPE REF TO ty_kna1.
DATA : st_kna1_o_3 TYPE REF TO ty_kna1.
BREAK-POINT.
```

Listing 10.5 Different Types of Components

S...	Variable Name	VariableType	Bound Used Memory [Bytes]	Bound Allocated Memory [Bytes]	Used Object Memory [Bytes]	Allocated Object Memory [Bytes]
	ST_KNA1_1	Structure: flat, not charlike(4038)	0	0	4,038	4,038
	ST_KNA1_2	Structure: flat, not charlike(4038)	0	0	4,038	4,038
	ST_KNA1_3	Structure: flat, not charlike(4038)	0	0	4,038	4,038
	ST_KNA1_B_1	Structure: deep(8)	8	8	8	8
	ST_KNA1_B_2	Structure: deep(8)	8	8	8	8
	ST_KNA1_B_3	Structure: deep(8)	8	8	8	8
	ST_KNA1_O_1	Fref	0	0	0	0
	ST_KNA1_O_2	Fref	0	0	0	0
	ST_KNA1_O_3	Fref	0	0	0	0

Figure 10.57 Memory Allocation for Different Types of Components

10.3.3 Table Types

Table types describe the structure and functional attributes of internal tables. You learned how to create a global table type in ABAP Data Dictionary in [Chapter 7](#), but we limited our discussion there in order to quickly define a table type with default options. In this section, we'll discuss table types in greater detail and go over the various options that are available while defining a table type in ABAP Data Dictionary.

The **LINE TYPE** tab of the **DICTIONARY: CHANGE TABLE TYPE** screen in ABAP Data Dictionary (see [Figure 10.58](#)) allows you to maintain the line type for the table type. This line type then defines the components of the table type. Internal tables that refer to this table type will have a row type as defined by the line type of the table type.

There are multiple options to define the line type of the table type:

- ▶ **LINE TYPE**
In this field, you can enter an existing type such as a data element, structure, table type, table, view, and so on; its properties are then copied as properties of the table line.
- ▶ **PREDEFINED TYPE**
You can select this option to manually input the ABAP Data Dictionary type, length, and any decimals.
- ▶ **REFERENCE TYPE**
Using this option, you can specify an object reference (class or interface) or data reference (elementary data type).

Dictionary: Change Table Type

← → ↺ ↻ 📄 📁 📂 📅 📆 📇 📈 📉 📊 📋 📌 📍 📎 📏 📐 📑 📒 📓 📔 📕 📖 📗 📘 📙 📚 📛 📜 📝 📞 📟 📠 📡 📢 📣 📤 📥 📦 📧 📨 📩 📪 📫 📬 📭 📮 📯 📰 📱 📲 📳 📴 📵 📶 📷 📸 📹 📺 📻 📼 📽 📾 📿 📰 📱 📲 📳 📴 📵 📶 📷 📸 📹 📺 📻 📼 📽 📾 📿

Hierarchy Display

Table Type: ZDEMO_ST_TB New(Revised)

Short text: Demo Table Type

Attribu... | **Line Ty...** | Initialization and Access | Primary Key | Secondary Key

Line Type

Predefined Type

Data Type: []

No. of Characters: 0 Decimal Places: 0

Reference type

Name of Ref. Type: []

Reference to Predefined Type

Data Type: []

Length: 0 Decimal Places: 0

Figure 10.58 Line Type View in Table Type Maintenance Screen

Under the **INITIALIZATION AND ACCESS** tab (see [Figure 10.59](#)), attributes such as the type of table and initial size can be maintained. The available options are as follows:

▶ **INITIAL LINE NUMBER**

This value is used for internal memory management to allocate the initial size of the table. This is similar to using the `INITIAL SIZE` addition in ABAP programs while defining the internal table.

▶ **ACCESS**

You can select the kind of table that is defined when a data object is referred to this table type. The following options are found under the **ACCESS** section:

- ▶ **STANDARD TABLE:** Defines a standard internal table.
- ▶ **SORTED TABLE:** Defines a sorted internal table. The key should be specified in the **PRIMARY KEY** and **SECONDARY KEY** tabs.
- ▶ **HASHED TABLE:** Defines a hashed internal table. The key should be specified in the **PRIMARY KEY** and **SECONDARY KEY** tabs.
- ▶ **INDEX TABLE:** Index tables define a generic table category for standard and sorted tables (because they have an index).
- ▶ **NOT SPECIFIED:** Defines a generic category for any table (all table categories).

Generic types can only be used for generic typing of formal parameters in procedures or with field symbols.

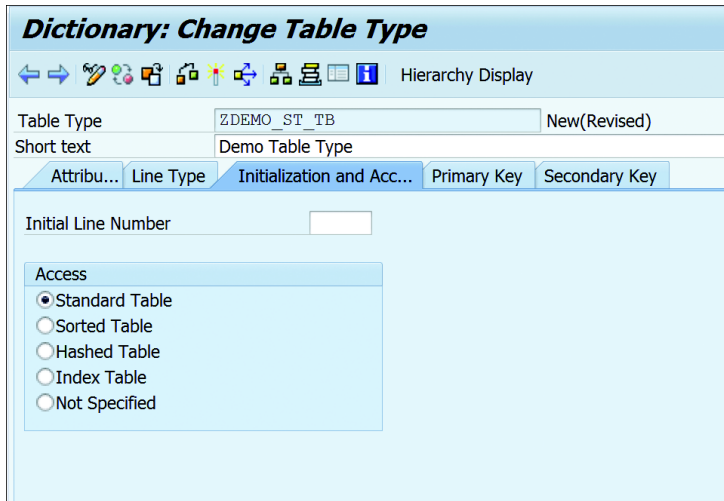


Figure 10.59 Initialization and Access Tab

The PRIMARY KEY tab (see [Figure 10.60](#)) allows you to define the primary key. This property will then be set for internal tables that refer to this table type. The available options on this screen are as follows:

► **KEY DEFINITION**

Under the KEY DEFINITION section, you can define one of the following for the primary key:

- **STANDARD KEY:** This sets the standard key in which all the fields of the row are part of the key. This option can be selected only if the line type of the table type is not a table type itself or the line type doesn't contain any components that are table types.
- **LINE TYPE:** The primary key is the line type used for the table type.
- **KEY COMPONENTS:** The primary key fields can be selected manually by clicking the CHOOSE COMPONENTS button.
- **NOT SPECIFIED:** This is used for generic table types for which a key is not defined.

► KEY CATEGORY

KEY CATEGORY defines whether the key is unique or nonunique. If the key fields uniquely define a row, the key can be set to UNIQUE; otherwise, NON-UNIQUE. You can select NOT SPECIFIED for generic types.

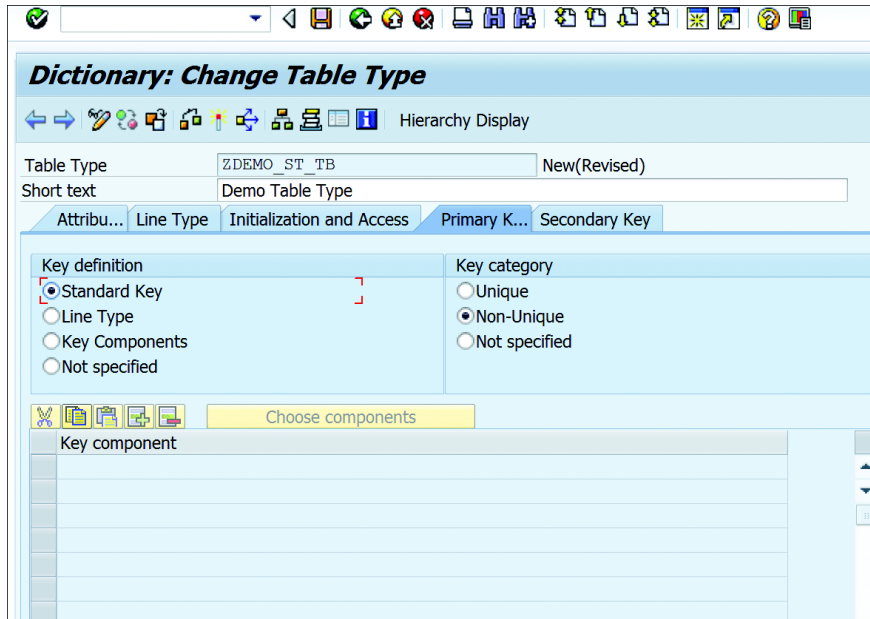


Figure 10.60 Primary Key Tab

Beyond the primary key, additional secondary keys can be specified under the SECONDARY KEY tab (see [Figure 10.61](#)). You can define up to fifteen secondary keys for the table type. Each secondary key should have a unique name. The name of the secondary key can't be PRIMARY_KEY or LOOP_KEY. The secondary key can be either a sorted key or a hash key. If it's sorted key, it can be unique or non-unique; a hash key is always unique. The key definition for a secondary key is defined by the entire table row if the line type is not a table type or doesn't contain a table type as a component.

In this section, we discussed using data types, including data elements, structures, and table types. In the next section, we'll see how type groups are used and maintained.

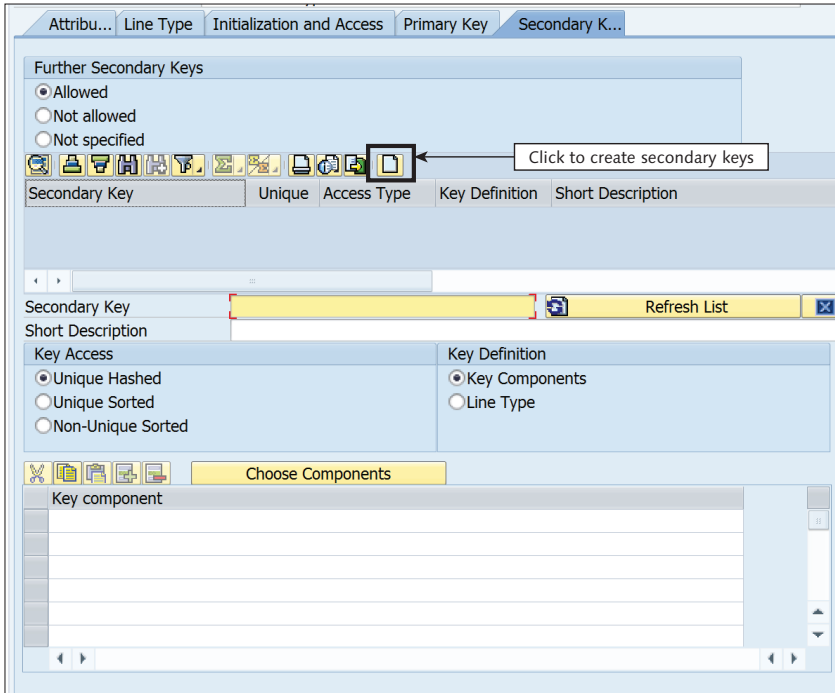


Figure 10.61 Secondary Keys

10.4 Type Groups

Type groups are created in ABAP Data Dictionary where global data type declarations are maintained. Type groups are obsolete; we have better ways to maintain data types with all the semantic attributes, as discussed in the previous section.

One of the disadvantages of data types defined in a type group is that, except for any comment lines maintained in the code, no documentation or other semantic attributes can be maintained. Another drawback of using type groups is that the complete type group should be loaded even if you want to reference only one or two data types or constants from the type group.

Type groups still exist to provide backward compatibility to programs that use definitions from type groups. You may come across function modules that define their formal parameters by referring to the data types in a type group. To ensure

compatibility, the actual parameters in the calling program should also be referred to the data types of the type group.

In older systems, type group declarations were made visible to the program by using `TYPE-POOLS` statement. This is no longer required, because type groups are visible to programs by default.

Because type groups are obsolete, you shouldn't create any new type groups and should only use existing type groups if required. For example, we can use the constants `abap_true` and `abap_false` from the type group `ABAP` (see [Figure 10.62](#)) to check for `X` or a space, respectively.

```

13 | * abap_endia
14 | * abap_repl
15 |
16 | □ *****
17 | L **** GENERAL *****
18 | types:
19 |   abap_bool type c length 1.
20 | * constants for abap_bool
21 | constants:
22 |   abap_true      type abap_bool value 'X',
23 |   abap_false     type abap_bool value ' ',
24 |   abap_undefined type abap_bool value '-',
25 |   abap_on        type abap_bool value 'X',
26 |   abap_off       type abap_bool value ' '.
27 |
28 |
29 | □ *****
30 | L **** DESCRIBE *****
31 | constants:
32 |   abap_max_abs_type_name_ln type i value 200,
33 |   abap_max_class_name_ln   type i value 30,
34 |   abap_max_intf_name_ln    type i value 30,

```

Figure 10.62 Type Group ABAP

10.5 Domains

Domains provide the technical attributes for data elements. The important property of a domain is that it defines a *value range*. All the table fields and structure components that use the data element to which a domain is attached can have value ranges defined by the domain. Value ranges provides possible entries for a field.

In [Chapter 4, Section 4.4.1](#), we discussed how to define a domain and attach it to a data element. In this section, we'll discuss some of the options that can be set for the domain.

You can create a domain in ABAP Data Dictionary by selecting the DOMAIN radio button on the ABAP Data Dictionary initial screen. The DEFINITION tab on the DICTIONARY: CHANGE DOMAIN screen (see [Figure 10.63](#)) includes the following options:

► **FORMAT**

Use the help to maintain an ABAP Data Dictionary elementary data type. Maintain the number of characters and any decimals.

► **OUTPUT CHARACTERISTICS**

Output characteristics define how the fields referenced to a data element that uses this domain are displayed on screens or when using the WRITE statement. The following options are found under the OUTPUT CHARACTERISTICS section:

- **OUTPUT LENGTH:** The *output length* value is determined based on the elementary data type selected. This value can be overridden by maintaining a value in this field. The output length of the field should be determined based on any formatting characters in the field. For example, for a date field, the internal length is 8 (YYYYMMDD), but the output length should be larger to accommodate any separators (DD-MM-YYYY).
- **OUTPUT STYLE:** A domain that is a decimal floating point should be assigned an output style by selecting one of the available options.
- **CONVERSION ROUTINE:** *Conversion routines* (also known as *conversion exits*) are special function modules that are created with the naming conventions `CONVERSION_EXIT_XXXXX_INPUT` and `CONVERSION_EXIT_XXXXX_OUTPUT`, where `XXXXX` is the five-character name of the conversion routine.

For each conversion routine, both input and output function modules exist. If a screen field is referencing a data element to which this domain is attached and if the conversion routine is maintained for the domain, then the `_INPUT` function module is called for each input on the screen and the `_OUTPUT` function module is called for the screen output. This allows you to convert the data into internal and external formats.

For example, the user will input the date in an external format (DD-MM-YYYY), and this value is passed to a conversion routine (`_INPUT` function module), which converts the date to an internal format (YYYYMMDD) that the program

can use internally (say, in a select WHERE clause). When this date is outputted to the screen, the date is converted to an external format (_OUTPUT function module) and shown to the user.

You can maintain a five-character conversion routine name in this field. The conversion routine name can also be assigned to screen fields directly. If you want to create a custom conversion routine, you can do so in Function Builder following the naming convention for conversion routines. The conversion exit function modules should have one mandatory parameter as INPUT and one mandatory parameter as OUTPUT. The input parameter will have the value being converted, and the output parameter will have the converted value.

- ▶ SIGN: If the data type is numeric, then this option can be selected to show a negative sign for negative values on screens.
- ▶ LOWER CASE: For character-type fields, setting this option will ensure that a value is not transformed to uppercase. For example, when you input a value in a selection screen field, the value is automatically transformed to uppercase when passed to the program. When you set this value at the domain of the field, the value will not be transformed and will be passed as is to the program field.
- ▶ TIME FORMAT: For time fields (type TIMS), the time format can be set as 12-hour or 24-hour format.

Dictionary: Change Domain

Domain: ZDEMO_DOMAIN New(Revised)

Short Description: Demo domain

Proper... Definiti... Value Range

Format

Data Type: []

No. Characters: []

Decimal Places: []

Output Characteristics

Output Length: []

Output: Normal

Convers. Routine: []

Sign

Lower Case

Figure 10.63 Domain Maintenance Screen

The VALUE RANGE tab (see [Figure 10.64](#)) allows you to maintain the value range for fields that reference data elements to which the domain is attached. Here, you can set fixed values, intervals (a value range), or a combination of both. The value range can be defined for elementary data types CHAR, NUMC, DEC, INT1, INT2, and INT4. For fields of type CHAR and NUMC, the value range specifies the valid input values for screen fields that refer to the data element to which the domain is attached.

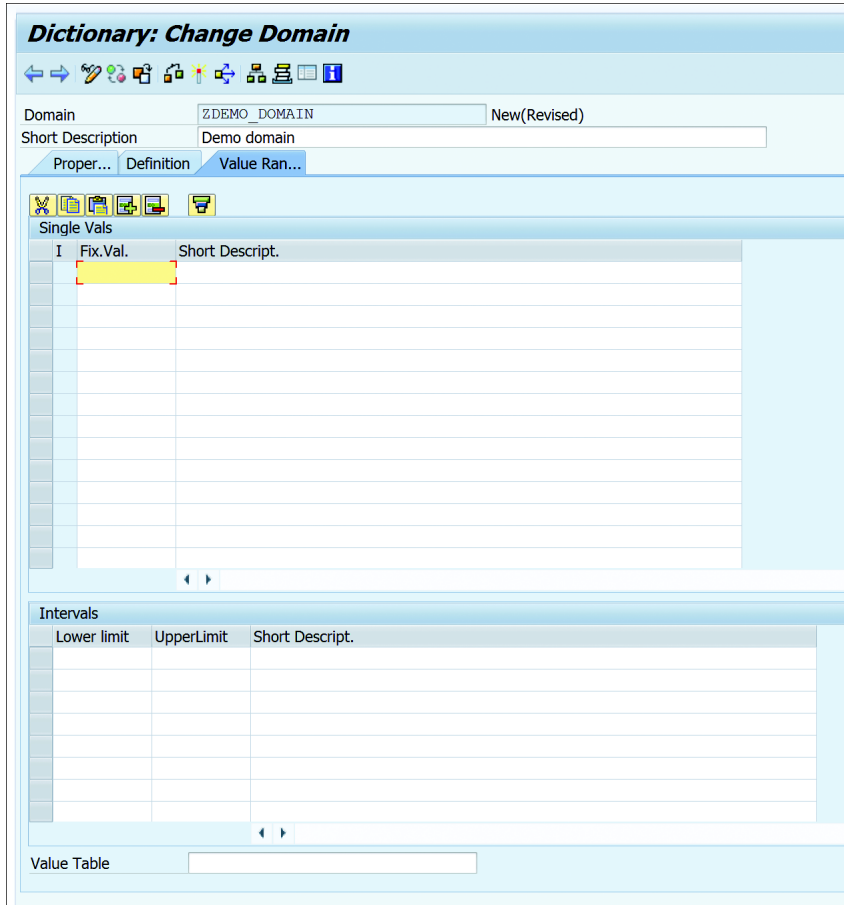


Figure 10.64 Value Range

The input field doesn't have a search help attached; the value range is shown as the input help. The value range is ignored for the content of data objects in ABAP

programs and database fields. Short text can be maintained for each individual value and is then displayed in the input help if the value range is used for the input help.


A value table can be maintained and is then proposed as a check table when a check table is assigned for table fields using foreign keys. The value table itself doesn't trigger a check.

For standard domains, a fixed value append can be defined to maintain additional fixed values. The additional fixed values will be not be lost in upgrades. A domain can have more than one fixed value append. Use the menu path GoTo • FIXED VALUE APPEND to define a fixed value append for the domain.

In this section, we discussed the use of domains and how to create a domain. In the next section, we'll explore the different types of search helps that can be used in ABAP Data Dictionary.

10.6 Search Helps

Search helps provide input help for screen fields. A search help can be assigned to a screen field, data element, or table field. *Input helps* are a standard function of the SAP system. When we develop applications, we should maintain input helps for our screens to provide a consistent UI experience for users.

To use an input help, the user can press **F4** on the keyboard or click the INPUT HELP  button on the screen field. Once the input help is called, it displays the possible entries for the field in multiple rows, and the user can double-click the most suitable row to copy the value to the input field.

Input helps also can offer multiple search paths and restrictions to let the user search for a suitable value. For example, if the user needs to input a plant number but he only knows the plant name, the input help can allow the user to search for the plant number based on the plant name.

Multiple search paths are displayed as tabs in input helps. For example, [Figure 10.65](#) shows the search helps with a single search path (single tab) and multiple search paths (multiple tabs). [Figure 10.66](#) shows the search help restriction that is displayed to the user to search for the value.

Plant (1) 264 Entries found

Restrictions

Search help with single search path

Plant	Search Term 2	Search Term 1	Postl Code	City
0001		SAP	69190	Walldorf
0002		SAP	69190	Walldorf
0005	1000	HAMBURG	22299	Hamburg
0006			10001	NEW YORK
0007	1000	HAMBURG	22299	Hamburg
0008			10001	NEW YORK
006			10001	New York
007			22299	

Material Number (1) 1000 Entries found

Material Number/Material Description Material by old material number Material by bill of material

Search help with multiple search paths

Material description		
\$00000000000000000000000001	EN	2538
\$00000000000000000000000001	EN	2539
\$00000000000000000000000001	EN	2540
\$00000000000000000000000001	EN	2541
\$00000000000000000000000001	EN	2542
\$00000000000000000000000001	EN	2543
\$00000000000000000000000001	EN	2544
\$00000000000000000000000001	EN	2545
\$00000000000000000000000001	EN	2546
\$00000000000000000000000001	EN	2547
\$00000000000000000000000001	EN	2548
\$00000000000000000000000001	EN	2550
\$00000000000000000000000001	EN	2551

Figure 10.65 Search Helps

Material Number (1)

Material Number/Material Description Material by old material number Material by bill of material

Material description:

Language Key:

Material:

Maximum No. of Hits:

Figure 10.66 Search Help Restriction

There are two types of search helps that can be created in ABAP Data Dictionary:

▶ **Elementary search helps**

Elementary search helps provide a single search path for input help. Two or more elementary search helps can be used to create a collective search help.

▶ **Collective search helps**

A collective search help provides multiple search paths for the input help.

Ahead, we'll discuss these two kinds of search helps and how to create them. We'll then look at how to assign search helps to screen fields and data elements before walking through search help exits.

10.6.1 Elementary Search Helps

Elementary search helps can be created by using a single table or a projection view. If search help fields are required from more than one table, you can define a database view or help view and use it as a selection method for the search help.

The following steps take you through creating an elementary search help:

1. On the initial ABAP Data Dictionary screen, select the SEARCH HELP radio button, provide the search help name in the customer namespace (starting with Z or Y), and click the CREATE button, as shown in [Figure 10.67](#).

ABAP Dictionary: Initial Screen

Database table

View

Data type

Type Group

Domain

Search help

Lock object

Display Change Create

Figure 10.67 ABAP Data Dictionary: Initial Screen

2. In the dialog window that appears, select **ELEMENTARY SEARCH HELP** and click the green checkmark (see [Figure 10.68](#)).

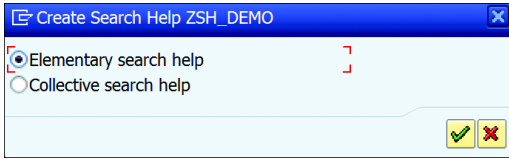


Figure 10.68 Create Search Help Window

This takes you to the **DICTIONARY: CHANGE SEARCH HELP** screen, as shown in [Figure 10.69](#).

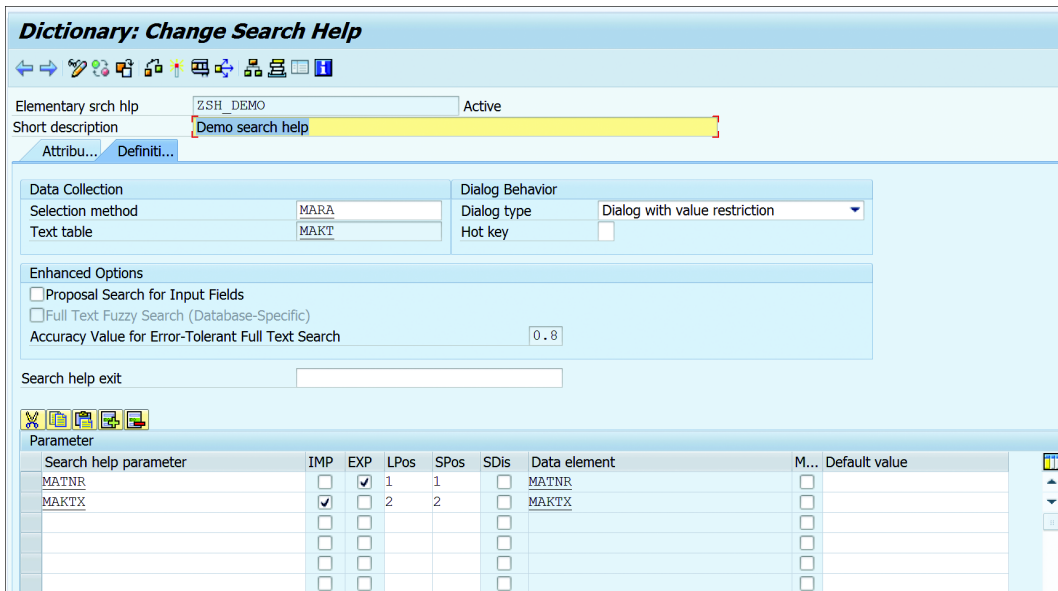


Figure 10.69 Search Help Maintenance Screen

3. Maintain the following options:

- ▶ **SHORT DESCRIPTION:** Provide a meaningful short description. This description can be used to search for the search help.
- ▶ **SELECTION METHOD:** Under the **DATA COLLECTION** section, provide the table or view name (except maintenance view) that should be used for data selection.

- ▶ **TEXT TABLE:** If the selection table has a text table attached, it will be automatically filled in the TEXT TABLE field of the DATA COLLECTION section.
- ▶ **DIALOG TYPE:** This setting influences how the search help dialog is displayed. Using the dropdown menu, choose one of the following options:
 - **IMMEDIATE VALUE DISPLAY:** When the input help is called, the hit list is displayed immediately. Select this option if the hit list is small and usually contains only a few records.
 - **DIALOG WITH VALUE RESTRICTION:** When the input help is called, the dialog box for restricting values is displayed, which allows the user to restrict the hit list. Select this option if the list of possible entries is normally very large. Restricting the hit list improves system performance and clarity for the user.
 - **DIALOG DEPENDS ON VALUE SET:** This setting allows you to dynamically decide if the hit list should be displayed immediately or the restrictions dialog should be displayed. If the hit list contains less than one hundred entries, it's displayed immediately; if the hit list contains more than one hundred entries, the dialog box for restricting values is displayed.
- ▶ **HOT KEY:** When the elementary search help is used in a collective search help, a *hot key* allows the user to select the elementary search help using the hot key. You can enter a two-character alphanumeric value.
- ▶ **ENHANCED OPTIONS:** These are additional enhanced options that work with SAP GUI 7.40 or later:
 - **PROPOSAL SEARCH FOR INPUT FIELDS:** Selecting this option will display the search help values below the field automatically when the user starts typing. This does not require the search help to be called manually via **F4**.
 - **FULL TEXT FUZZY SEARCH:** Selecting this option allows the system to perform the search across multiple columns so that the user can identify the input value based on the value in different columns. For example, if the user doesn't know the customer number, the data in the NAME or ADDRESS columns can help the user identify the customer for input.
 - **ACCURACY VALUE:** This allows you to provide the accuracy value that should be considered—for example, if the system should exactly match the value when performing a search or use variations to address any spelling mistakes.

- ▶ **SEARCH HELP EXIT:** Standard search helps can be enhanced using a search help exit. We'll discuss search help exits in [Section 10.6.4](#).
- ▶ **PARAMETER:** These options allow you to maintain fields that are displayed in the search help. The following columns are found under the **PARAMETER** table:
 - **SEARCH HELP PARAMETERS:** Enter the fields from the selection table that should be displayed in the search help.
 - **IMP:** Flag for setting the parameter as an importing parameter. The value of this parameter is imported to search help from the screen fields.
 - **EXP:** Flag for setting the exporting parameter. The value of this parameter is exported to the screen fields.
 - **LPOS:** Sets the position of the field on the hit list. For example, if the value 3 is set, then the field will be displayed as the third column. If no value or 0 is maintained, then this field is hidden.
 - **SPOS:** Sets the position of the field on the restrictions dialog. For example, in [Figure 10.66](#), the **SPOS** value of the **MATERIAL** field is 3. If no value or 0 is maintained, then this field is hidden.
 - **SDIS:** Flag to set the parameter as a display-only field. The contents of the parameter restrict the hit list, and the user can't change the value. A constant value can be assigned to this parameter.
 - **DATA ELEMENT:** The data element used for the field in the selection table is automatically populated based on the field name entered. The screen field to which this search help is assigned uses the documentation maintained for this data element as **F1** help.
 - **MODIFIED:** If you want to change the data element name that was auto-filled, select this flag to make the data element field input-enabled.
 - **DEFAULT VALUE:** Maintain a default value that should be used for the parameter.

Once all the required parameters are maintained, activate the search help. You can execute the search help directly for testing by using the **EXECUTE** button in the application toolbar.

10.6.2 Collective Search Helps

Two or more search helps can be used to create a collective search help. A collective search help provides multiple search paths to users, with each elementary search help displayed as a tab.

Like elementary search helps, the collective search help has an interface to import values from screen fields and export values to screen fields. When creating a collective search help, you can include all the fields of the elementary search helps or only selective fields from the elementary search helps.

The following steps walk through creating a collective search help. For this example, we'll use two elementary search helps: ZSH_DEMO (see [Figure 10.69](#)) and ZSH_DEMO1 (see [Figure 10.70](#)). The elementary search help ZSH_DEMO uses the table MARA as the selection method with MATNR and MAKTX as parameters. The elementary search help ZSH_DEMO1 uses the table MARC as the selection method with MATNR and WERKS as the parameters.

Dictionary: Change Search Help

Elementary srch hlp: ZSH_DEMO1 Active
Short description: Material by Plant

Selection method: MARC
Text table:
Dialog Behavior: Dialog type: Dialog with value restriction
Hot key:

Enhanced Options
 Proposal Search for Input Fields
 Full Text Fuzzy Search (Database-Specific)
Accuracy Value for Error-Tolerant Full Text Search: 0.8

Search help exit:

Search help parameter	IMP	EXP	LPos	SPos	SDIs	Data element	M...	Default value
MATNR	<input type="checkbox"/>	<input checked="" type="checkbox"/>	1	1	<input type="checkbox"/>	MATNR	<input type="checkbox"/>	
WERKS	<input checked="" type="checkbox"/>	<input type="checkbox"/>	2	2	<input type="checkbox"/>	WERKS_D	<input type="checkbox"/>	
	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>		<input type="checkbox"/>	
	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>		<input type="checkbox"/>	
	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>		<input type="checkbox"/>	
	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>		<input type="checkbox"/>	

Figure 10.70 Elementary Search Help Using Table MARC as Selection Method

Proceed with the following steps to create a collective search help:

1. On the initial ABAP Data Dictionary screen, select the SEARCH HELP radio button, provide the search help name, and click CREATE.

2. In the create search help window (refer back to [Figure 10.68](#)), select COLLECTIVE SEARCH HELP and click the green checkmark.
3. On the DICTIONARY: CHANGE SEARCH HELP screen (see [Figure 10.71](#)), enter the search help parameters and the corresponding data elements. Set the IMPORT and EXPORT flags for the parameters. Optionally, you can set a default value for each parameter.

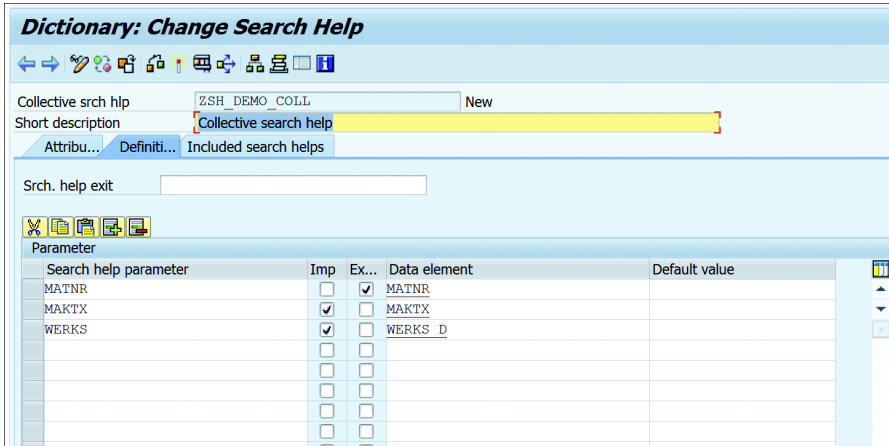


Figure 10.71 Collective Search Help Maintenance Screen

4. Click the INCLUDE SEARCH HELPS tab (see [Figure 10.72](#)) and include the search helps that should be part of the collective search help. You can include both elementary and collective search helps.

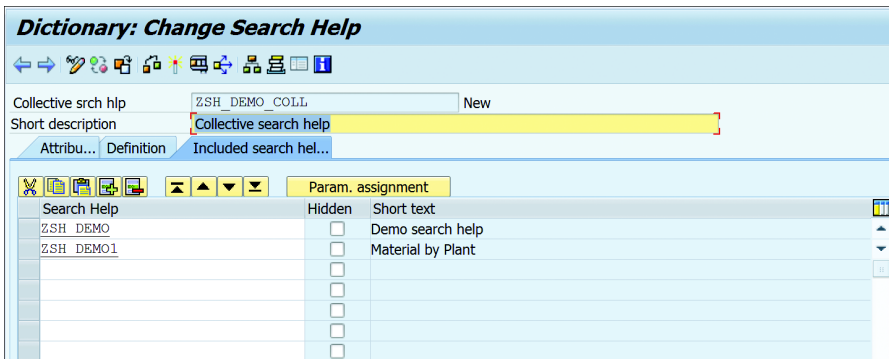


Figure 10.72 Include Search Helps

- Keep the cursor on a search help name and click the PARAM.ASSIGNMENT button. The DEFINE PARAMETER ASSIGNMENT dialog box will appear (see [Figure 10.73](#)). Click YES to let the system propose the assignment, and click COPY on the next screen. Repeat this process for all search helps listed.

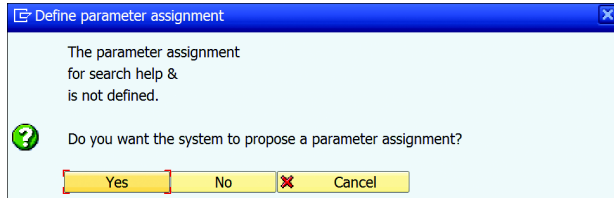


Figure 10.73 Parameter Assignment

- For each search help listed, you can set the HIDDEN flag (refer to [Figure 10.72](#)) to hide the search help. This is useful when you want to hide a particular search path in a certain system.
- Activate the search help by clicking the EXECUTE button to test it.

10.6.3 Assigning a Search Help

Search helps can be assigned to screen fields directly or maintained for data elements and then used when the field refers to those data elements.

To assign a search help to selection screen field, use MATCHCODE OBJECT as shown:

```
PARAMETERS p_matnr TYPE matnr MATCHCODE OBJECT ZSH_DEMO_COLL.
```

For screen fields of the general screen, the search help can be assigned in Screen Painter under the field attributes, as shown in [Figure 10.74](#). We'll work with Screen Painter in [Chapter 12](#).

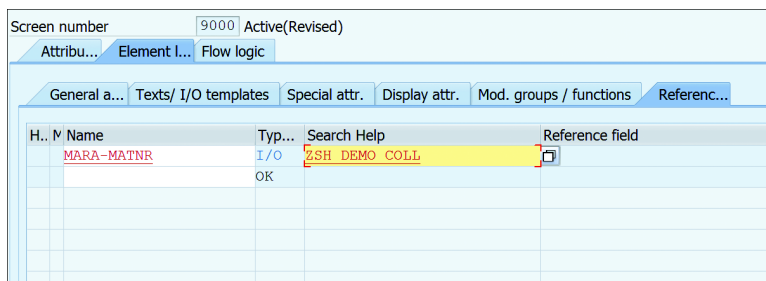


Figure 10.74 Field Attributes

The search help can be assigned to the data element under the FURTHER CHARACTERISTICS tab, as shown in [Figure 10.75](#). Enter the search help name in the NAME field and use the **F4** help for the PARAMETERS field to select the parameter of the search help that should be passed to this field. Screen fields referring to this data element will automatically inherit the search help unless overridden by the screen field attribute value.

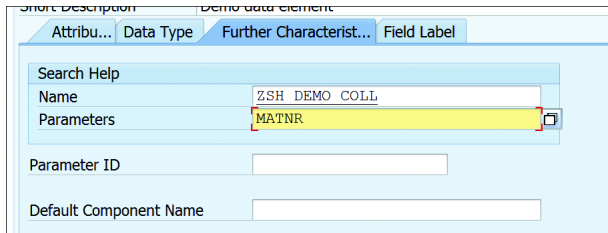


Figure 10.75 Assigning Search Help to Data Element

10.6.4 Search Help Exits

Search help exits are used to enhance the standard functionality of a search help. This allows you to dynamically manipulate the search help functionality. For example, you can adjust the column width or combine the values of two columns. A search help exit is implemented as a function module with a predefined interface. The demo function module `F4IF_SHLP_EXIT_EXAMPLE` can be used as a reference to create your own function module for a search help exit.

Search help exits are called on different events by the help processor, as listed:

- ▶ Before displaying the dialog box for selecting the required search path (only for collective search helps)
- ▶ Before starting the **F4** process for the elementary search help
- ▶ Before displaying the dialog box for entering search conditions
- ▶ Before selecting data
- ▶ Before displaying the hit list
- ▶ Before returning the values selected by the user to the input template

These steps are passed to the function module parameter `CALLCONTROL-STEP` (refer to the demo function module `F4IF_SHLP_EXIT_EXAMPLE` in [Function Builder](#)). This

field is checked using an IF condition in which suitable logic can be implemented for each step.

In this section, we looked at the two different types of search helps used and how to create and assign them before discussing the use of search help exits to enhance standard system functionality. In the next section, we'll discuss the final topic of this chapter: lock objects.

10.7 Lock Objects

Lock objects are used to provide logical locking functionality in applications. For example, if you don't want two users to edit the same document simultaneously, then you can implement lock objects to restrict a user from opening a document if another user has already opened the document for editing.

Lock objects are defined in ABAP Data Dictionary. A lock object creates two function modules, `ENQUEUE` and `DEQUEUE`, that can be called in ABAP programs to lock and unlock the objects, respectively.

Follow these steps to create a lock object in ABAP Data Dictionary:

1. On the ABAP Data Dictionary initial screen, select the LOCK OBJECTS radio button. Enter the name of the lock object and click CREATE. The lock object name should start with `EZ`, as shown in [Figure 10.76](#).

ABAP Dictionary: Initial Screen

Database table

View

Data type

Type Group

Domain

Search help

Lock object

Display Change Create

Figure 10.76 ABAP Data Dictionary: Initial Screen

2. On the **DICTIONARY: CHANGE LOCK OBJECT** screen, select the **TABLES** tab, as shown in [Figure 10.77](#). Enter the **NAME** of the **PRIMARY TABLE** and select the **LOCK MODE**. The lock mode determines how the lock is implemented.

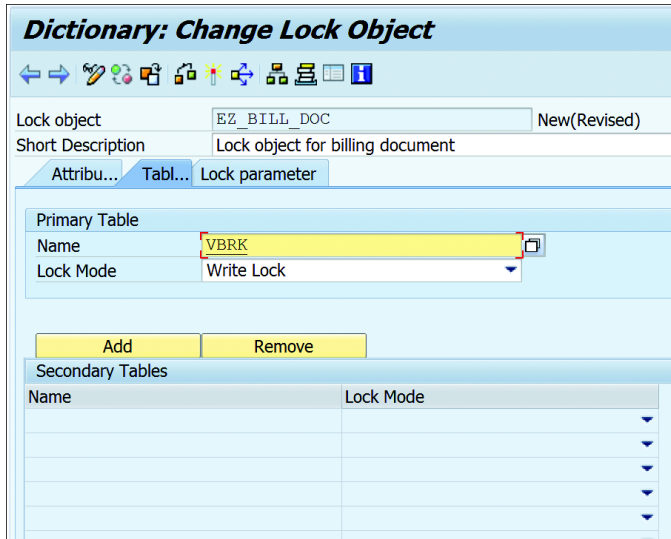


Figure 10.77 Lock Object Maintenance Screen

3. The following options can be set for **LOCK MODE** under the **PRIMARY TABLE** section:
 - ▶ **EXCLUSIVE LOCKS (WRITE LOCK):** When an exclusive lock is applied, the locked data can be displayed or edited only by one user. Any further requests for either another exclusive lock or a shared lock are rejected.
 - ▶ **SHARED LOCK (READ LOCK):** When a shared lock is applied, several users can access the data in read mode. However, as soon as any one user goes into change mode to process the data, a second user will not have any access to the data. Requests from further shared locks are accepted, even if they're from different users. Exclusive locks are rejected.
 - ▶ **EXCLUSIVE BUT NOT CUMULATIVE LOCK:** Unlike exclusive locks that can be requested several times by the same transaction, an exclusive, noncumulative lock can only be requested once by the same transaction. Each subsequent lock request will be rejected.

- ▶ OPTIMISTIC LOCK (O): Optimistic locks initially behave like shared locks and can be converted into exclusive locks.
- ▶ PROMOTE OPTIMISTIC LOCK (R): Converts an optimistic lock into an exclusive lock.
- ▶ COLLISION CHECK/PROMOTION CHECK ('U', 'V', 'W', 'C'): Checks whether one of the respective locks ('E', 'S', 'X', 'O') is possible, but doesn't set a lock

If required, further secondary tables can be added to the lock object.

4. Select the LOCK PARAMETER tab, as shown in [Figure 10.78](#). The key fields of the table are automatically populated. Verify and activate the lock by clicking the ACTIVATE button in the toolbar.

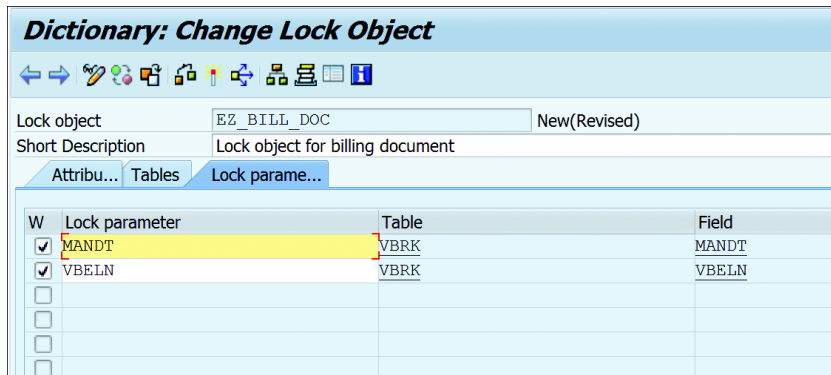


Figure 10.78 Lock Parameters

5. Once the lock object is activated, the system generates two function modules: ENQUEUE_obj name and DEQUEUE_obj name. For this example, the function modules ENQUEUE_EZ_BILL_DOC and DEQUEUE_EZ_BILL_DOC are generated, as shown in [Figure 10.79](#). These function modules can be called in ABAP programs to set and release locks. The lock parameters are passed to the function module during requests to set or release the lock.

[Listing 10.6](#) shows example code to implement lock objects using the generated function modules. In this example, we're calling the ENQUEUE function module ENQUEUE_EZ_BILL_DOC within the subroutine REQUEST_LOCK. To this function module, we're passing the document number for which a lock should be set.

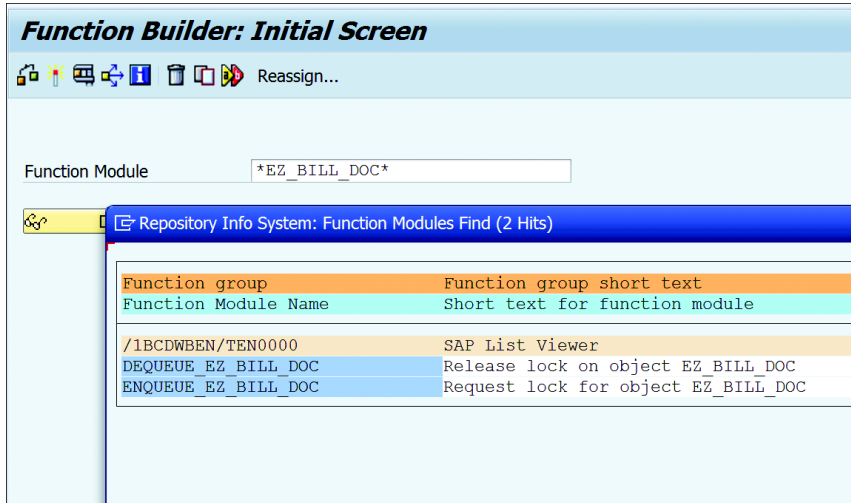


Figure 10.79 Function Modules Generated by Lock Object

Once the lock is set, we perform other operations (such as updating the document). Locking the document ensures that no other user can edit the same document using the same program while another user is editing it. At the end, we release the lock by calling the `DEQUEUE` function module `DEQUEUE_EZ_BILL_DOC` within the subroutine `RELEASE_LOCK`.

We pass the same document number to the `DEQUEUE` function module that was originally locked using the `ENQUEUE` function module. Once the lock is released, another user will be able to edit the document.

```
*&-----*
*&      Module  CHANGE_DOCUMENT  INPUT
*&-----*
MODULE change_document INPUT.
  PERFORM request_lock CHANGING retcode.
  IF retcode IS INITIAL.
    PERFORM update.
  ENDIF.
  PERFORM release_lock.
ENDMODULE.
*&-----*
*&      Form    REQUEST_LOCK
*&-----*
FORM request_lock CHANGING p_retcode.
  CALL FUNCTION 'ENQUEUE_EZ_BILL_DOC'
    EXPORTING
```

```

mode_vbrk      = 'E'
mandt          = sy-mandt
vbeln         = vbrp-vbeln
x_vbeln       = ' '
_scope        = '2'
_wait         = ' '
_collect      = ' '
EXCEPTIONS
  foreign_lock = 1
  system_failure = 2
  OTHERS      = 3.
IF sy-subrc <> 0.
  p_returncode = sy-subrc.
ENDIF.
ENDFORM.
*&-----*
*&      Form  RELEASE_LOCK
*&-----*
FORM release_lock .
  CALL FUNCTION 'DEQUEUE_EZ_BILL_DOC'
    EXPORTING
      mode_vbrk = 'E'
      mandt     = sy-mandt
      vbeln     = vbrp-vbeln
      x_vbeln   = ' '
      _scope    = '3'
      _synchron = ' '
      _collect  = ' '.
ENDFORM.

```

Listing 10.6 Working with Lock Objects

10.8 Summary

In this chapter, you learned how to create various ABAP Data Dictionary objects. Understanding dictionary objects is vital to ABAP development, because you'll use objects from the ABAP Data Dictionary in every application.

We've discussed how to create database tables using ABAP Data Dictionary which immunizes us from knowing the intricacies of the underlying relational database when performing various tasks related to defining tables. Using ABAP Data Dictionary also ensures we need not use the DDL part of SQL to perform various tasks, such as creating tables, deleting tables, creating indexes, and so on. This ultimately ensures that the database is consistent and avoids redundancy.

We discussed how views can be created by joining different tables to quickly process data from multiple tables. With the help of data types, domains, and search helps, we can easily extend the functionality of applications while maintaining them centrally.

We concluded our discussion of ABAP Data Dictionary by looking at how lock objects can be used for the lock management of application data. All such data generated by applications is transient, which means the data is lost when the execution of the application ends. In the next chapter, we'll discuss the ways in which data can be saved persistently.

ABAP programs generally work with data that is read from a persistent storage source, like a database, meaning that the data exists even when the program isn't reading it. This chapter explains how to save persistent data for use in ABAP programs.

11 Persistent Data

Data exists in two states: a transient state and a persistent state. When ABAP programs are processing data, the data in the program is *transient* in nature; that is, its lifetime is limited to when the application is running. When the application is closed, the application and the data in the application are cleared from memory. This data can't be accessed again. If you want to access the data processed by the application, you need to save the data in a persistent form in a persistent storage media.

Data stored in a persistent form is called *persistent data*. All ABAP programs work with persistent data in a user-driven manner. Data can be stored persistently in different ways, such as the following:

- ▶ In database tables in the underlying relational database
- ▶ In data clusters in special database tables
- ▶ In text or binary files on the application server
- ▶ In text or binary files on the presentation server (user desktop)

The most popular way of storing data persistently is to save it to database tables, which are fully supported by SAP. However, based on your requirements, we may have to use other persistent media to save the data. For example, if the user wants to download the report output to a spreadsheet for further analysis, it can be downloaded as a spreadsheet file to the user's desktop. The first choice for storing persistent data should always be storage in relational database tables, because this option allows you to work with the data using Open SQL, which is integrated into the ABAP language.

You can also have object-oriented access to the data in the database via ABAP Objects. Sometimes, it may be advantageous to store the data in data clusters, especially when the data isn't available in first normal form (1NF)—for example, in nested internal tables. Refer back to [Chapter 5](#) for a refresher on normal forms.

Storing persistent data as files on the application server or presentation server should be the least preferred choice, because unlike data stored in database tables or data clusters, application or presentation server files offer no built-in support for different code pages or byte sequences. Storing data in files on the presentation or application server is more suitable for interfaces for data transport to and from external systems.

In this chapter, we'll discuss all the different ways in which data can be stored persistently. We'll begin in [Section 11.1](#) by looking at how to work with data in a database, including using Open SQL statements and providing data consistency with logical units of work (LUW).

In [Section 11.2](#), we'll discuss ABAP Object Services, the framework used to provide services to create and work with persistent objects. We'll include steps for building persistent classes and working with persistent objects.

[Section 11.3](#) discusses storing data in the application and presentation layers (file interfaces). Then, in [Section 11.4](#), we'll look at the data storage capabilities of data clusters.

Finally, in [Section 11.5](#), we'll turn from data consistency to data security and steps for ensuring the right users have access to the right data.

11.1 Working with Data in Databases

Each SAP system uses an underlying relational database. The database system consists of tables in which the data is stored in rows and columns. Each column represents a field of the table, and the data of the fields make up the rows. Each table has a field (or set of fields) that uniquely identify a row, called the *primary key* of the table. One table can be related to other tables in the database using *foreign keys*. As discussed in [Chapter 10](#), we can also define *secondary keys* for database tables, which the SQL optimizer can use to speed up the data lookup process.

SAP NetWeaver AS ABAP is delivered with many standard SAP tables. Some of these tables are prepopulated with data related to the system, such as repository objects or customization settings. Many database tables, such as application tables, are delivered empty and then updated by customers via ABAP programs or by transferring external data from legacy systems. Each relational database system provides a programming interface to access the tables of the database using the SQL query language. SQL statements aren't completely standardized and vary from one database vendor to another.

As discussed in [Chapter 5](#), standard SQL is comprised of the following languages:

► **Data Definition Language (DDL)**

DDL contains statements such as `CREATE`, `ALTER`, and `DROP`, which allow you to define or alter database tables. ABAP doesn't support these statements directly because they're vendor-specific; that is, they vary from vendor to vendor. Rather than using these DDL statements in ABAP programs, we use the ABAP Data Dictionary tool to define or alter tables in the database; the tool implements these DDL statements such that they work with the underlying database. DDL statements are part of the Native SQL statements—that is, native to the underlying database.

► **Data Manipulation Language (DML)**

DML contains statements such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, which allow you to process data in the database tables. ABAP implements these statements as Open SQL statements; they're independent of the database. SAP NetWeaver AS ABAP contains a database interface that handles converting Open SQL statements in ABAP to vendor-specific SQL (or Native SQL) statements.

The database interface makes ABAP programs portable and database-independent. We discussed Native SQL and Open SQL statements in [Chapter 5](#) and provided a brief overview of different Open SQL statements, and we recommend that you refer to that section if you need a quick recap. In the next subsection, we'll examine Open SQL statements in more depth and look at the different Open SQL statements supported in ABAP.

► **Data Control Language (DCL)**

DCL contains statements that allow you to define user views, execute access control, and ensure data consistency. ABAP doesn't contain any DCL statements. Access control is ensured using SAP NetWeaver AS ABAP authorization

concepts (see [Section 11.5](#)), and data consistency is ensured via the LUW concept.

Data in a database should always be stored in a consistent form, but different applications trying to work with the same data can easily leave the database in an inconsistent state. In [Section 11.1.2](#), you'll learn about concepts like LUW that help ensure data consistency.

11.1.1 Open SQL

ABAP programs don't interact with the database system directly; instead, the database interface of SAP NetWeaver AS ABAP is responsible for communicating with the underlying database. When we use Open SQL statements in ABAP programs, the database interface converts the Open SQL statements into Native SQL statements that are native to the underlying database. This layer of abstraction makes ABAP programs portable; that is, they can work in any database environment.

Open SQL comprises the DML part of standard SQL that allows data from the database tables to be read and changed. For example, the `SELECT` statement allows you to read data from a database table, whereas the `INSERT`, `UPDATE`, `MODIFY`, and `DELETE` statements allow you to change data in a database table. Open SQL statements in ABAP offer additional variations compared to standard SQL; these variations simplify and accelerate database access.

Open SQL statements must be monitored for negative impacts on an application's performance. If the database access isn't programmed properly, a program can adversely affect system performance. We can use various tools, such as SQL trace (Transaction ST05) or runtime analysis (Transaction SAT), to monitor the performance of database accesses. You'll learn more about the various tools that can be used to monitor and improve the performance of ABAP programs in [Chapter 21](#).

You can follow certain general guidelines while programming database access to ensure your programs perform at high level of efficiency. When you fetch data from the database, the amount of data fetched puts a load on the memory of the database as well on the network between the database and the application server. We recommend keeping database hits and the amount of data to a minimum.

For example, if you want to process only certain data fields from the database table, you should fetch only the required fields instead of fetching the complete

row. Also, you should program in such a way that you only fetch the data from the database table once and work with that data locally by accessing the data objects of the program, instead of writing a `SELECT` statement to fetch the data from the same table multiple times in the program, because each database access adds to the workload on the database and the network.

The `WHERE` clause of the `SELECT` statement also plays a crucial role in improving the performance of the program. You should ensure the search effort is low by formulating the `WHERE` clause to use any available index, or in general keep the search effort as low as possible. Unnecessary database access should be avoided because generally many application programs access the same database, and each database access by each program will increase the workload on the database. There are no fixed rules for programming for efficient database access, so you need to decide the best possible approach on a case-by-case basis depending on the data you plan to process.

In the following subsections, we discuss the different Open SQL statements.

SELECT

The `SELECT` statement allows you to import data from the database tables into data objects of the ABAP program. The functions of the `SELECT` statement range from simple data selection to programming complex database queries.

We discussed the basic variations of the `SELECT` statement, such as fetching data from a single database or multiple database tables, in [Chapter 5](#). As we noted there, a basic `SELECT` statement contains different clauses, as shown in [Listing 11.1](#).

```
SELECT result
      FROM source
      INTO target
      WHERE cond.
```

Listing 11.1 SELECT Statement Clauses

In the listing, the `SELECT` clause `result` determines the result set of the database table—that is, the fields of the database table that are read. You can read only certain fields or all the fields of the database table. The `FROM` clause `source` determines the database table or view from which the data is read.

The `INTO` clause `target` determines the data object of the program into which the data from the database table is read. This data object is updated with the data

from the database after successful execution of the `SELECT` statement. If no data is read, the `SY-SUBRC` system field is updated with 4. The `SY-DBCNT` system field is updated with the number of records read.

The `WHERE` clause `cond` specifies the conditions under which the data is read. The system performs a lookup on the database table based on the condition specified in the `WHERE` clause. If no `WHERE` clause is specified, all the data from the database table specified in the `FROM` clause will be read to the data object specified in the `INTO` clause. Executing `SELECT` without a `WHERE` clause should always be avoided, because it can result in huge data transfer and cause memory overflow under certain conditions.

In addition to these basic clauses, the `SELECT` statement can contain three other clauses:

- ▶ **GROUP BY**
Used to aggregate several rows and import to a single row.
- ▶ **HAVING**
Used to specify conditions for aggregated rows.
- ▶ **ORDER BY**
Used to sort the rows of the result set.

Now, let's discuss the different clauses of the `SELECT` statement and the different additions that are supported.

SELECT Clause

The *SELECT clause* determines if one or several rows are read from a database table and what columns are read. If the `SINGLE` addition is used, only one record is read. If the `SINGLE` addition isn't specified, the `SELECT` statement should be closed with an `ENDSELECT` statement (or the target variable should be an internal table), and the system will assume that the result set contains multiple rows. The system loops between the `SELECT` and `ENDSELECT` statements, transferring one record at a time to the data object specified in the `INTO` clause.

The `SELECT...ENDSELECT` statement allows you to evaluate each row as it's transferred to the program. When using the `SINGLE` addition, the `WHERE` clause should use the complete key of the database table to uniquely identify the row; otherwise, the first matching row is fetched into the program. To ignore duplicates

when using a non-unique WHERE clause with SELECT...ENDSELECT, you can use the DISTINCT addition.

You can specify the columns that should be selected by listing the field names separated by spaces, or use * to select all the fields of the table. The following statement shows an example of using SELECT SINGLE to select certain fields of table SPFLI:

```
SELECT SINGLE carrid connid FROM spfli INTO st_
spfli WHERE carrid EQ 'AA' AND connid EQ '0017'.
```

The following statement shows an example of using SELECT SINGLE to select all the fields of table SPFLI:

```
SELECT SINGLE * FROM spfli INTO st_
spfli WHERE carrid EQ 'AA' AND connid EQ '0017'.
```

Listing 11.2 shows an example of how to use the SELECT...ENDSELECT statement to fetch certain fields from table SPFLI. The system loops between SELECT and ENDSELECT, transferring one record at a time.

```
SELECT carrid connid FROM sflight INTO st_
sflight WHERE carrid EQ 'AA' AND connid EQ '0017'.
*Code to process the record
ENDSELECT.
```

Listing 11.2 Fetching Particular Fields from Table SPFLI

Listing 11.3 shows an example of using the SELECT...ENDSELECT statement to fetch all fields from table SPFLI. The system loops between SELECT and ENDSELECT, transferring one record at a time.

```
SELECT * FROM sflight INTO st_
sflight WHERE carrid EQ 'AA' AND connid EQ '0017'.
*Code to process the record
ENDSELECT.
```

Listing 11.3 Fetching All Fields from Table SPFLI

Aggregate functions can be used with the SELECT clause, which then calculates the value in the database system and returns a single value. This is especially useful with CDS views, in which calculations can be pushed to a database, such as SAP HANA, which performs the aggregations much faster than fetching the records into the ABAP program and then performing aggregations. Aggregate functions

also reduce the data transfer between the database and the ABAP program by returning just the result. Aggregate functions bypass the buffer.

The following aggregation functions are supported:

► **Average (AVG)**

Returns the average value for the specified column field of the result set. The AVG function can only be applied to numeric fields. For example:

```
SELECT AVG( netwr ) FROM vbrp INTO lv_avg_netwr WHERE vbeln EQ
p_vbeln.
```

In the preceding example, the `lv_avg_netwr` variable should be of type F (floating point).

► **Minimum (MIN)**

Returns the smallest value for the specified column in the result set. NULL values are ignored unless all the rows in the column contain NULL values.

For example:

```
SELECT MIN( fking ) FROM vbrp INTO lv_min_fking WHERE vbeln EQ
p_vbeln.
```

► **Maximum (MAX)**

Returns the greatest value for the specified column in the result set. NULL values are ignored unless all the rows in the column contain NULL values.

For example:

```
SELECT MAX( fking ) FROM vbrp INTO lv_max_fking WHERE vbeln EQ
p_vbeln.
```

► **Total (SUM)**

Returns the sum of all values for the specified column in the result set. SUM can only be applied to numeric fields.

For example:

```
SELECT SUM( fking ) FROM vbrp INTO lv_sum_fking WHERE vbeln EQ
p_vbeln.
```

► **Count (COUNT)**

Returns the number of rows or distinct rows for the specified column in the result set. Using `COUNT DISTINCT` returns the number of distinct rows, whereas `COUNT (*)` returns the number of rows for the given selection.

For example:

```
SELECT COUNT( DISTINCT MATNR ) FROM vbrp INTO lv_
count WHERE vbeln EQ p_vbeln.
```

In the preceding example, the `lv_count` variable should be of type `integer`.

Each column specified in the `SELECT` clause can be set as an alternate column name via the `AS` addition. This is useful when you want to address the column name in other additions of the `SELECT` statement, such as `ORDER BY` or `JOIN`. For example:

```
SELECT vbeln AS a1 fking AS a2 FROM vbrp INTO CORRESPONDING FIELDS OF
TABLE it_vbrp WHERE vbeln EQ p_vbeln ORDER BY a2.
```

FROM Clause

The *FROM clause* allows you to specify an ABAP Data Dictionary table, a classic view, or a CDS view. You can specify one or multiple tables using inner and outer joins.

In the example of [Listing 11.1](#), `FROM source` was used to show the database table or view from which the data is read.

INTO Clause

The *INTO clause* determines the ABAP program data objects into which the data should be fetched. The data object specified in the `INTO` clause must match the table columns listed in the `SELECT` clause. With `SELECT SINGLE`, the `INTO` clause can't list an internal table. If `SELECT SINGLE` isn't used, the `INTO` clause can list an internal table using the `TABLE` addition. For example:

```
SELECT * FROM vbrp INTO TABLE it_vbrp WHERE vbeln EQ p_vbeln.
```

The `INTO TABLE` addition copies the result set into the internal table, and any previous records in the internal table will be replaced. If you want to keep the previous records in the internal table and append the new records, you can use the `APPENDING TABLE` addition instead of the `INTO TABLE` addition.

The `CORRESPONDING FIELDS OF` addition is used to automatically map the identical fields from the result set to the target data object. If the column names are supplied with alternate names using the `AS` addition, then the alternate names are matched with the target data object. This is especially useful when selecting data from multiple tables into a single target data object when more than one table contains the same field name.

WHERE Clause

The *WHERE clause* allows you to restrict the data selection; it plays a crucial role in performance of the program. You can specify logical expressions with the *WHERE* clause, which can be combined with *AND* and *OR*, or negated with *NOT*. Logical expressions can form conditions by using the comparison operators *=*, *<*, *>*, *<>*, *<=*, or *>=*, such as in the following example: `WHERE carrid = 'AA'`.

The left operand in the comparison must always be a field of the table specified in the *FROM* clause. You can also specify the predicates *BETWEEN* and *IN* in the *WHERE* clause—for example: `WHERE col BETWEEN v_col1 AND v_col2`. The *IN* predicate is used with selection tables (range tables).

INSERT

The *INSERT* statement inserts a new record in the database table. If the primary key of the record already exists, the new record isn't inserted.

Similar to other Open SQL statements, the *INSERT* statement sets the value for system fields *SY-SUBRC* and *SY-DBCNT*. If the *SY-SUBRC* value is 0 after the statement is executed, it means the record is inserted to the database table successfully. The *SY-SUBRC* field will be set to 0 if an empty internal table is supplied in the source. If the *SY-SUBRC* value is 4, it means the record isn't inserted to the database table, because the primary key or unique secondary key already exists in the table. The *SY-DBCNT* system field is updated with the number of rows that are inserted.

With the *INSERT* statement, you can insert a single row from a structure (e.g., `INSERT table_name FROM wa`) or multiple rows from an internal table (e.g., `INSERT table_name FROM TABLE itab`).

[Listing 11.4](#) shows an example of inserting a single row using the *INSERT* statement.

```
DATA : wa_makt TYPE makt.

wa_makt-matnr = '100'.
wa_makt-spras = 'E'.
wa_makt-maktx = 'Pen'.

INSERT makt FROM wa_makt.
IF sy-subrc IS INITIAL.
MESSAGE 'Record inserted successfully' TYPE 'I'.
ENDIF.
```

Listing 11.4 Inserting Single Row

Listing 11.5 shows an example of inserting multiple rows.

```
DATA : it_makt TYPE STANDARD TABLE OF makt,
      wa_makt TYPE makt.
wa_makt-matnr = '100'.
wa_makt-spras = 'E'.
wa_makt-maktx = 'Pen'.
APPEND wa_makt TO it_makt.

wa_makt-matnr = '200'.
wa_makt-spras = 'E'.
wa_makt-maktx = 'Pencil'
.
APPEND wa_makt TO it_makt.

INSERT makt FROM TABLE it_makt.
IF sy-subrc IS INITIAL.
MESSAGE 'Records inserted successfully' TYPE 'I'.
ENDIF.
```

Listing 11.5 Inserting Multiple Rows

UPDATE

The *UPDATE* statement updates an existing record in the table. When this statement is used, the non-key fields of the existing row are updated. If the key doesn't exist, no record is updated.

Similar to the *INSERT* statement, you can update a single record by supplying a structure or update multiple records by passing an internal table. The system sets the *SY-SUBRC* and *SY-DBCNT* fields, similar to the *INSERT* statement. These system fields can be checked to identify if the record is updated successfully and the number of records that were updated, respectively.

The syntax to use the *UPDATE* statement is similar to that for the *INSERT* statement. For example, for the code in Listing 11.4, to update a record from a work area, you can use the *UPDATE* statement in place of *INSERT* as follows:

```
UPDATE makt FROM wa_makt
```

Similarly, in Listing 11.5, to update from an internal table, you can use the following statement:

```
UPDATE makt FROM TABLE it_makt.
```

MODIFY

The *MODIFY* statement works as a combination of the *INSERT* and *UPDATE* statements. When you use the *MODIFY* statement, a record is inserted if the key doesn't exist and updated if the key does exist. The *SY-SUBRC* and *SY-DBCNT* system fields are set in a similar way as with the *INSERT* and *UPDATE* statements.

The syntax to use the *MODIFY* statement is also similar to the other two statements. For example, to modify a single record, you can use the *MODIFY* statement as follows:

```
MODIFY makt FROM wa_makt.
```

To modify multiple records using an internal table, you can use the statement as follows:

```
MODIFY makt FROM TABLE it_makt.
```

DELETE

The *DELETE* statement deletes a row or multiple rows from the database table permanently. With the *DELETE* statement, you can supply a *WHERE* condition or a source structure/internal table. When a *WHERE* condition is used, all the matching records are deleted. The *SY-SUBRC* and *SY-DBCNT* system fields are set similarly to the other three statements.

Listing 11.6 shows the syntax to use the *DELETE* statement.

```
*Delete using WHERE condition
DELETE FROM makt WHERE matnr EQ '100' AND spras EQ 'E'.
*Delete using work area/structure
DELETE makt FROM wa_makt.
*Delete using internal table
DELETE makt FROM TABLE it_makt.
```

Listing 11.6 Using *DELETE* Statement

These Open SQL statements don't update the database immediately when the statement is executed in the application server; instead, the data is committed to the database when a commit work is performed in an LUW. Until the commit work is performed, these statements set a database lock. If used incorrectly, these statements can result in a dead lock (blocking access to the data).

New SQL Syntax

With the release of SAP NetWeaver 7.4, SAP has introduced new SQL syntax, which was further updated in SAP NetWeaver 7.5. As previously discussed, Open SQL is a subset of standard SQL and contains limited functions compared to those in standard SQL. With SAP NetWeaver 7.4 and 7.5, SAP has expanded the possibilities of Open SQL.

The following are some of the important additions introduced to Open SQL:

► Inline declarations

With SAP NetWeaver 7.4, SAP introduced inline declarations, which allows you to declare data objects when required as opposed to declaring the data object beforehand. Before SAP NetWeaver 7.4, a data object needed to be declared as follows before it could be accessed by ABAP statements:

```
DATA lv_date TYPE sy-datum.  
date = sydatum.
```

As of SAP NetWeaver 7.4, you can perform the data declaration inline and avoid extra lines of code as follows:

```
DATA(lv_date) = sy-datum.
```

The support for inline declarations extends to SQL statements—so instead of defining the structures and internal tables beforehand, you can declare the target field while reading the data from the database. For example, [Listing 11.7](#) shows the traditional way to write code.

```
REPORT ZDEMO_EXAMPLE.  
TYPES : BEGIN OF ty_makt,  
        matnr TYPE matnr,  
        spras TYPE spras,  
        maktx TYPE maktx,  
        END OF ty_makt.  
DATA it_makt TYPE STANDARD TABLE OF ty_makt.  
PARAMETERS p_matnr TYPE matnr.  
SELECT matnr spras maktx FROM makt INTO TABLE it_  
makt WHERE matnr eq p_matnr.
```

Listing 11.7 Traditional Code

As of SAP NetWeaver 7.4, you can save considerable effort by not defining the structure type and internal table beforehand. The code in [Listing 11.7](#) can be written as shown in [Listing 11.8](#).

```
PARAMETERS p_matnr TYPE matnr.
SELECT matnr, spras, maktx FROM makt WHERE matnr EQ @p_matnr
INTO TABLE @DATA(it_makt).
```

Listing 11.8 New Syntax as of SAP NetWeaver 7.4

In the preceding example, we're using the new SQL syntax, in which the fields of the table are separated by commas and all the host variables are escaped using the escape character @. *Host variables* are the program data objects that should be escaped to tell the compiler that we're not talking about the fields in the database.

► **Strict mode**

When using new SQL syntax, the syntax checker runs in strict mode to ensure all of the new syntax rules are followed. Strict mode helps maintain backward compatibility by forcing you to write the statement either completely using new syntax or completely using old syntax, but not a mix of both.

► **INTO clause**

The `INTO` clause has moved to the end of the statement. By moving the `INTO` clause to the end of the statement, SAP can further enhance the Open SQL statement by adding support for new features.

► **CASE statement**

You can embed the `CASE` statement within an SQL statement, as opposed to fetching the data into the program and then comparing the data using the `CASE` statement.

Listing 11.9 shows a sample statement in which the sex of an employee is checked via the `CASE` statement.

```
SELECT name, age,
       CASE WHEN sex = 1 THEN @male
            WHEN sex = 2 THEN @female
            ELSE @others
       END AS group
FROM zemployee
ORDER BY group
INTO TABLE @DATA(it_employee).
```

Listing 11.9 Case Expression

► **Arithmetical operations**

The new SQL syntax allows you to perform arithmetical operations within the SQL statement. In versions before SAP NetWeaver 7.4, you'll have to get the

data into the program and then perform arithmetic operations on the data. However, with the new SQL syntax, you can perform arithmetic operations inside SQL statements.

In the following statement, we're performing an arithmetic operation inside a SQL statement—in this case, reading the result of multiplying the `fkimg` (quantity) and `netwr` (net value) of an item in the billing document:

```
SELECT SINGLE fkimg * netwr FROM vbrp WHERE vbeln EQ @p_vbeln  
AND posnr EQ @p_posnr INTO @DATA(total).
```

11.1.2 Logical Unit of Work

In the real world, a user completes a transaction by performing a sequence of actions that logically belong together—for example, booking a flight. This sequence of actions should be completed together as a single unit, and this unit is called a *logical unit of work* (LUW).

To ensure correctness and completeness of data, either all data belonging to an LUW should be committed to the database together or none of the data should be committed. For example, when you transfer an amount in financial accounting, one account should be debited and the other account credited. Once the amount is debited from one account and before it's posted to another account, the data is in an inconsistent form. If the database is updated immediately when the account is debited but, due to an error in the application, the credit account isn't posted, there will be data inconsistency because the amount isn't reflected in either account.

The data in a logical unit should be committed wholly and there should be a mechanism to roll back the data if the transaction wasn't completed successfully. To avoid such data inconsistency and ensure that data is always consistent at the end of each transaction, we implement the concepts of a database LUW and SAP LUW.

In the following subsections, we'll look at each LUW type.

Database LUW

A *database LUW* exists between each dialog step; it works as a single unit with an inseparable sequence of database operations that ends with a database commit. A database LUW is either executed fully or not at all. If there is any error within a

database LUW, none of the data within the LUW is committed to the database. This ensures that at the end of each database LUW, the data is always in a consistent form. In other words, the data is either modified at the end of the database LUW or it remains as it was before the beginning of the database LUW.

When you run a transaction, the program loads in the application server, which is processed by a work process. The work process sends a screen to the user in the presentation layer. Once the user makes an input and, for example, clicks a button, the request is sent to the work process in the application server. During this time, while the work process is processing the request, the screen is visible to the user but isn't ready for input. In other words, the screen is inactive. Once the work process has processed the request, it will send the next screen in the sequence, making the screen active for input.

The time between the screen becoming inactive and the screen becoming active again when the work process finishes processing is called a *dialog step* (see [Figure 11.1](#)). The database LUW begins each time a dialog step starts and ends when the commit work is performed.

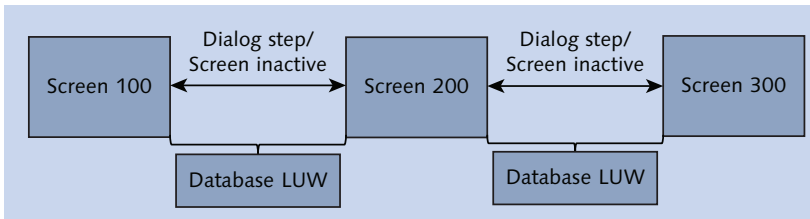


Figure 11.1 Dialog Steps

Database changes within a database LUW are not written to the database until the commit work is performed. This allows you to perform a roll back to reverse the changes. For example, say that you've executed an `INSERT` statement in the program to insert the header data of a document, but later in the code, some crucial data about the item details of the document are missing, so you can't update the item details.

Because the records aren't written to the database until the commit work is performed, the `INSERT` statement didn't update the database with the header record of the document immediately when the statement was executed. At this point, you can still roll back the data using the `ROLLBACK WORK` statement when the validation for item data fails.

The commit work is performed both implicitly and explicitly; it's performed *implicitly* in the following instances:

- ▶ When a dialog step is completed.
- ▶ When a remote-enabled function module is called in another work process (RFC call) from the program. This passes the control to another work process, so the commit work is performed implicitly.
- ▶ When the remote-enabled function module called in another work process ends. This results in control being returned to the calling work process.
- ▶ When a `WAIT` statement is used in the program. This interrupts the work process, and the control passes to another work process.
- ▶ When a message dialog is shown (error message, warning message, or informative message). This results in control being passed to the presentation layer.

The commit work can be performed *explicitly* in a program by calling the `DB_COMMIT` function module or using the `COMMIT WORK` ABAP statement.

Within a database LUW, a rollback is also performed both implicitly and explicitly. The rollback is performed implicitly when the program results in a runtime error or when a termination message (type `A` or `X`) is generated using the `MESSAGE` statement. The rollback can be triggered explicitly by using the `ROLLBACK WORK` ABAP statement.

A database LUW also consists of *database locks* to ensure no two users can edit the same data simultaneously. The lock is active during the duration of the database LUW and is automatically released at the end of the database LUW.

Each work process can only handle one database LUW. However, many database LUWs can be run in parallel in the system by different work processes.

SAP LUW

The lifetime of each database LUW is limited to a single dialog step—but what if your transaction extends over several dialog steps? Perhaps the transaction is spread over a series of screens called in sequence and the data spread over multiple dialog steps should be handled in one LUW; for example, the data should be saved when the user confirms the last screen. If you used `INSERT` or `UPDATE` statements in individual screens, the data would be committed to the database when the commit work is performed implicitly between each dialog step. There's no

way for you to roll back this data if the user cancels the transaction on the final screen.

To handle transactions that are typically spread across multiple dialog steps, we implement different *bundling techniques* to carry out the database changes in the final dialog step. Using these bundling techniques, we can ensure that the data of individual dialog steps isn't committed to the database within the database LUWs between the dialog steps. Bundling techniques involve using a function module or a subroutine that's executed in the last dialog step, as opposed to using the Open SQL statements in individual dialog steps, to update the data.

A *logical unit* consisting of multiple dialog steps for which changes are written to the database in a single database LUW (final database LUW) is called an SAP LUW. Whereas a database LUW spans a single dialog step, an SAP LUW spans multiple dialog steps, as shown in [Figure 11.2](#).

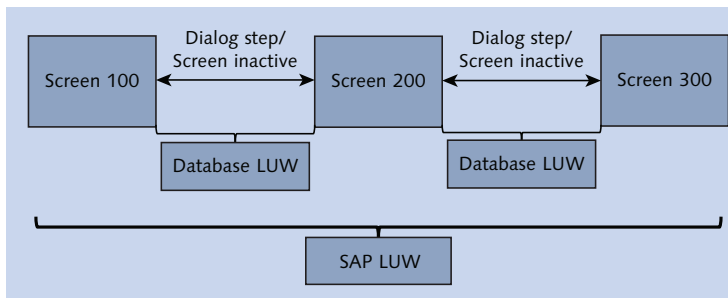


Figure 11.2 SAP LUW Dialog Steps

SAP LUW effectively extends the LUW across multiple dialog steps. This allows you to bundle all the database changes that occur in individual dialog steps so that they can be committed in the final database LUW using the `COMMIT WORK` statement when the transaction ends successfully. If any error occurs, we can use the `ROLLBACK WORK` statement to cancel all the bundled updates. In this way, either the data in a transaction is updated fully in the database or none of the data is updated.

The SAP LUW allows you to handle complex business processes and distribute your transaction across multiple work processes or even across multiple systems. The various bundling techniques we use to implement SAP LUW are discussed in the following subsections.

Bundling with Function Modules

To bundle database updates until the last step of an SAP LUW, a special function module called an *update function module* is used.

Update function modules are created in Function Builder by setting the PROCESSING TYPE to UPDATE MODULE in the function module attributes, as shown in Figure 11.3.

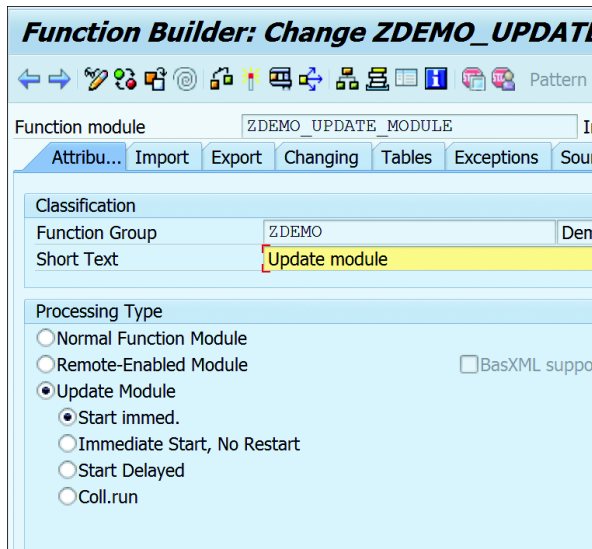


Figure 11.3 Update Function Module

There are two types of updates: *high-priority updates*, called V1, and *low-priority updates*, called V2. V1 updates are those that describe critical or primary changes, such as the creation of orders or changing material stock. V2 updates are those that describe less critical, secondary changes, such as statistical updates or results calculations. V1 updates always are processed before V2 updates. V1 updates are processed on the same application server in which the update is requested and are processed consecutively in a single-update work process; they can be rolled back and belong to a single database LUW. V2 updates are executed together in a separate LUW.

You can set the following attributes for the update function module:

- ▶ **START IMMEDIATELY**
Set this option for V1 updates within a shared SAP LUW. If there is any technical error during the update (e.g., a memory shortage), the update is rolled back and can be restarted by the update task.
- ▶ **IMMEDIATE START, NO RESTART**
Similar to the preceding option, but the update is not restarted in the case of a technical error.
- ▶ **START DELAYED**
Set this option for V2 updates. If there is any technical error, the update can be restarted by the update task.
- ▶ **COLL.RUN**
This flag is for internal SAP use. When this flag is set, the update function modules are executed collectively.

An update function module can't have exceptions or any exporting parameters, because the function module can't export anything to the program during an update. Also, all the parameters of the function module should be referenced to the ABAP Data Dictionary objects.

The Open SQL statements that modify the data are not written in the application program directly. Instead, the data that should be modified is passed to the update function module, and the Open SQL statements to modify the data in the database are used inside the function module. In other words, the update statements are written in the function module instead of the program. This update function module is then called at the location in the program at which you otherwise would have included Open SQL statements. The function module is called using the `IN UPDATE TASK` addition to the `CALL FUNCTION` statement (e.g., `CALL FUNCTION 'XYZ' IN UPDATE TASK`).

When a function module is called using the `IN UPDATE TASK` addition, it isn't executed immediately; instead, it's flagged for execution in a special work process called the *update work process*. The function module and its parameters are stored as a log entry in database table `VBLOG`. When the `COMMIT WORK` statement is executed in the program, the update work process takes care of updating the data, and the function module is executed by reading the log from table `VBLOG`.

For example, if the transaction flow involves three dialog steps and the update function module is called in the first dialog step, it isn't executed immediately; instead, a log entry is made in table `VBLOG`. In the final dialog step, if you use the `COMMIT WORK` statement, then the update work process will update the data in the database.

If for some reason in the final dialog step you decide not to update the data (due to a logical error), you can use the `ROLLBACK WORK` statement. When the `ROLLBACK WORK` statement is used, the log entry will be deleted from table `VBLOG` and no data will be updated to the database, because the function module won't be executed.

If due to any technical reasons the update work process can't complete the update, it will automatically perform a rollback and place the log entry back in table `VBLOG` (depending on the function module setting). It then sends a message to the user who originally started the dialog work process with the error information. After correcting the error, the `VBLOG` entries can be processed again.

This technique allows you to perform database updates asynchronously because updates are handled by a separate update work process. This in turn improves program performance because the dialog work process in which the program is running doesn't need to wait for the update to complete before executing the next step.

Bundling with Subroutines

A bundling technique can also be performed with subroutines using the `ON COMMIT` addition with the `PERFORM` statement. When you use the `PERFORM` statement to call a subroutine with the `ON COMMIT` addition (e.g., `PERFORM <xyz> ON COMMIT`), the subroutine isn't called immediately. The subroutine is called when the `COMMIT WORK` statement is executed in the program.

Unlike the update function module, the update task isn't assigned to the update work process; instead, it's handled by the same dialog work process. Because the system doesn't need to write log entries to table `VBLOG`, performance improves due to reduced database accesses. However, no parameters can be passed to this subroutine, so the program uses global variables and ABAP memory to perform the update, which can be risky and lead to inconsistencies.

Bundling with RFC

When you call a remote-enabled function module using `IN BACKGROUND TASK DESTINATION` with the `CALL FUNCTION` statement (e.g., `CALL FUNCTION 'ABC' IN BACKGROUND TASK DESTINATION 'QAS'`), the function module is registered for background execution in another SAP system specified in the *destination*. The destination contains the details of the remote system, such as remote system IP address and logon credentials. When the program encounters the `COMMIT WORK` statement, the remote function module is executed in background in the destination system.

If multiple remote function modules are called using the `IN BACKGROUND TASK DESTINATION` addition, all of them will be executed together in a single database LUW when the `COMMIT WORK` statement is executed in the program. The dialog work process doesn't need to wait for the update to complete, because the function modules are called asynchronously. This technique is useful if you want to update the same data in multiple systems.

So far, we've discussed various ways to store data persistently in the database. However, all of the techniques we discussed use procedural elements to save the data persistently. ABAP Object Services is a framework that provides various services to create and work with persistent objects. We'll discuss this further in the next section.

11.2 ABAP Object Services

As previously mentioned, ABAP Object Services is a framework for various services related to creating and working with persistent objects. *Persistent objects* allow you to use a purely object-oriented data model to store the data as opposed to using a relational data model. SAP provides a *persistence service* that acts as a software layer between persistent objects and the underlying database; in other words, the persistence service enables you to work with persistent data without having to use Open SQL statements in your program.

In this section, we'll discuss how to use ABAP Object Services for persistent objects. We'll begin with an overview of what a persistence service is, then we'll look at building persistent classes and working with persistent objects.

11.2.1 Persistence Service Overview

The persistence service is based on *object-relation mapping* (ORM) tools. Using ORM tools, the persistence details are encapsulated inside persistent classes by mapping the database tables/structures/views onto the object-oriented model. This in effect allows you to work with the data in the relational database in an object-oriented way. Classes with object-relational mapping are referred to as *persistent classes*, and the attributes that are mapped to the columns in the database table are called *persistent attributes*.

In other programming languages, such as Java, the ORM framework automatically generates the corresponding database tables and represents data as objects when you construct a class with attributes. However, SAP doesn't yet support such functionality. The persistence service is a software layer that intermediates between the persistent objects and the relational database (see [Figure 11.4](#)). The persistent objects themselves are transient in nature.

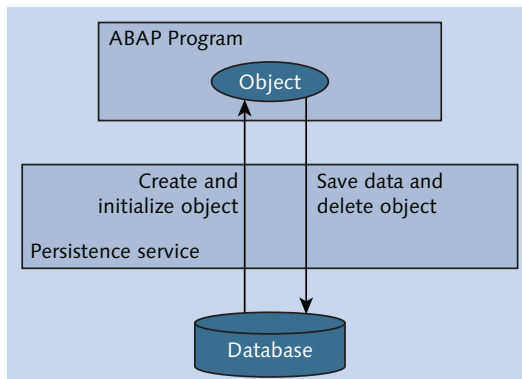


Figure 11.4 Persistence Service

The persistence service handles extracting the data from the database and initializing the attributes of the persistent object, and it works with the instances of persistent objects.

11.2.2 Building Persistent Classes

The steps to create a persistent class are similar to creating a regular class, except that you select `PERSISTENT CLASS` as the `CLASS TYPE`, as shown in [Figure 11.5](#). For customer objects, the name of the persistent class should start with `ZCA` or `YZA`.

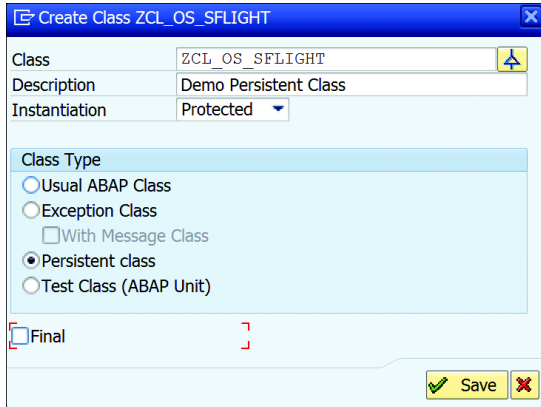


Figure 11.5 Creating Persistent Class

When the persistent class is created, it implements the `IF_OS_STATE` interface. This interface is implemented in all persistent classes to ensure all the persistent classes are accessed in a consistent way.

The Class Builder also creates two agent classes with the naming convention `CA_` and `CB_`. These agent classes are used to manage the persistent objects of the persistent class. In other words, these agent classes will have the methods that implement the actual Open SQL statements and take care of initializing the persistent objects.

For example, in Figure 11.5 we're creating a persistent class called `ZCL_OS_SFLIGHT`, and Class Builder will generate the `ZCA_OS_SFLIGHT` and `ZCB_OS_SFLIGHT` agent classes.

The `ZCB_OS_SFLIGHT` agent class is an abstract class, and the `ZCA_OS_SFLIGHT` class inherits from it. The `ZCB_OS_SFLIGHT` class is added as a friend to the `ZCL_OS_SFLIGHT` persistent class, as shown in [Figure 11.6](#).

At this point, the `ZCL_OS_SFLIGHT` class only has the methods from the `IF_OS_STATE` interface (see [Figure 11.7](#)). Class Builder adds additional methods once we perform persistence mapping with the database table, structure, or view. *Persistence mapping* is the process of mapping the database table fields with the attributes of the persistence class.

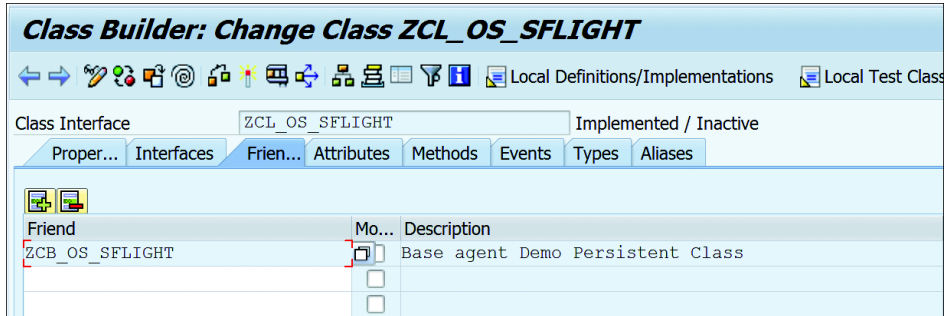


Figure 11.6 Abstract Agent Class Added as Friend

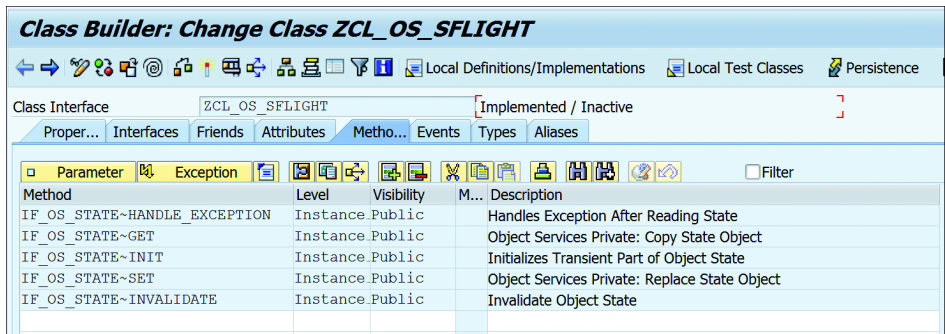


Figure 11.7 Methods of Base Class

To perform persistence mapping, click the PERSISTENCE button in the application toolbar, as shown in [Figure 11.7](#). This will open the mapping tool, as shown in [Figure 11.8](#). Provide the name of the table/structure to which you want to map the persistence class.

Class Builder will automatically create the attributes in the class for the fields of the table/structure mapped. It will also create setter and getter methods to manipulate these fields. The setter methods are not created for the key fields of the table because the key shouldn't be changed.

Once you provide the table/structure name, all the fields of the table will load in the bottom section, as shown in [Figure 11.9](#). You can double-click a field name to change some of the field's properties if required—for example, to make the field read-only.

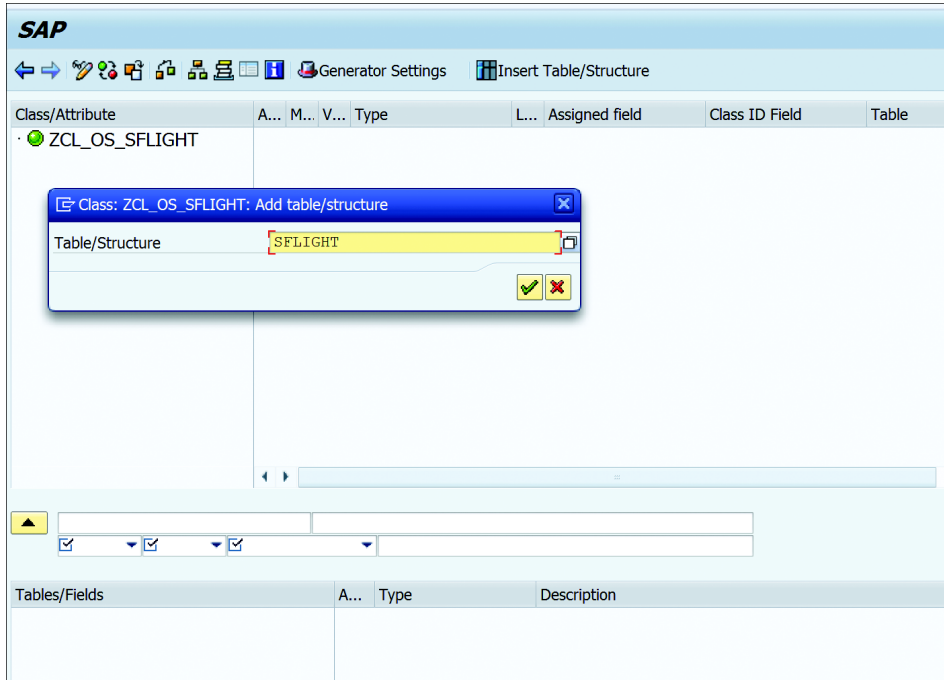


Figure 11.8 Mapping Tool

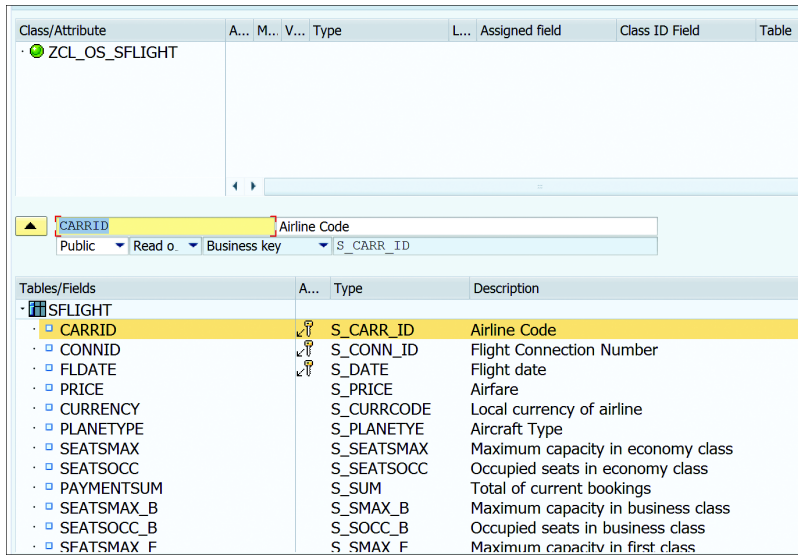


Figure 11.9 Fields of Table in Mapping Tool

Select each field and press `Enter` to add the field to the class. All the fields of the table should be added. If you want to use only some fields of the table, then create a projection view for the table and use the view name for mapping. Once all the fields are added, click `SAVE` and then `BACK`.

Note that all the fields are added as attributes, and the corresponding `SET` and `GET` methods are created (see [Figure 11.10](#)).



Figure 11.10 SET and GET methods

Now activate the class. If you haven't activated this class before, the system will show you a prompt asking whether the class activator should also be activated. Click `YES`.

11.2.3 Working with Persistent Objects

Working with persistent objects is fairly straightforward. The `CA` agent class implements a singleton design pattern, so we can't instantiate the object directly using the `CREATE OBJECT` statement in the program. A *singleton design pattern* ensures that only one instance of the object exists.

To get the reference of the agent class, we need to access the `agent` static attribute defined in the agent class. From there, we can work with the persistent data just like working with an object by calling the `get_persistent`, `create_persistent`, and `delete_persistent` methods to read, write, and delete a record in the database table.

[Listing 11.10](#) shows an example of reading a record by calling the `get_persistent` method of the agent class.

```
DATA : lr_sflight_agent TYPE REF TO zca_os_sflight,
      lr_sflight TYPE REF TO zcl_os_sflight,
      v_price TYPE s_price.
TRY.
  lr_sflight_agent = zca_os_sflight=>agent.
  lr_sflight = lr_sflight_agent->get_persistent(
    i_carrid = 'AA'
    i_connid = 0017
    i_fldate = '20150211'
  ).
  v_price = lr_sflight->get_price( ).
  WRITE v_price.
CATCH cx_os_object_not_found. " Object not Found
ENDTRY.
```

Listing 11.10 Working with Persistent Objects

In this section, we learn how to create persistent classes and work with persistent objects. In the next section, we'll look at working with files in the application server and the presentation layer.

11.3 File Interfaces

In addition to storing data in the database layer, you can save data in a persistent form in the application and presentation layers. Data is stored as sequential files in these layers—for example, as spreadsheets. In this section, you'll learn about working with files in the presentation application layers.

We'll first look at how sequential files can be saved to the application server directories from the ABAP program and how to read the saved files to be processed in ABAP programs in [Section 11.3.1](#). Then, in [Section 11.3.2](#), we'll discuss exporting and importing files from and to the presentation layer.

11.3.1 Working with Files in the Application Server

We use different ABAP statements to transfer data to a file stored in the application server directory. We typically perform two kinds of operations with files stored in the application server directory:

1. Read the file from the application server directory to an ABAP program (inbound interface)
2. Transfer the file from the ABAP program to the application server directory (outbound interface)

In an *inbound interface*, the data is read from the file and transferred to an internal table for further processing in the ABAP program. Conversely, in an *outbound interface*, the data from an internal table in the ABAP program is transferred to the file in the application server.

The following ABAP statements are used to read and write the files in the application server:

► OPEN DATASET

Opens the file for access. The system performs certain implicit authorization checks to ensure the user has the required permissions to access the file. Up to one hundred files can be opened per internal session. This statement will result in a catchable exception if the file is already opened in the program. The access type for which the file is to be opened should be supplied. The file can be opened for the following:

- For Input: Opens the file for reading.
- For Output: Opens the file for writing
- Appending: Opens the file for appending data. If the file exists, the file is opened and the cursor points to the end of the line to append new lines. If the file doesn't exist, a new file is created.
- Update: Opens the file for modifying the existing content.

The file can be opened in text mode or binary mode, and the encoding must be specified. The file should always be opened using this statement before any other operations can be performed.

The following statement shows an example of using the `OPEN DATASET` statement to open a file for writing:

```
OPEN DATASET v_file FOR OUTPUT IN TEXT MODE ENCODING DEFAULT.
```

Here, `v_file` contains the path of the file in the application server. `directory`.

► **TRANSFER**

If the file isn't opened for `INPUT`, data from a data object can be transferred to the file using this statement. Before this statement can be used, the file should be opened using the `OPEN DATASET` statement. If the file isn't opened, it will result in a catchable exception.

Listing 11.11 shows the sample code to transfer the records to file in the application server.

```
OPEN DATASET v_file FOR OUTPUT IN TEXT MODE ENCODING DEFAULT.
LOOP AT itab INTO wa.
TRANSFER wa TO v_file.
ENDLOOP.
CLOSE DATASET v_file.
```

Listing 11.11 Transferring Data to File

If the file is opened in text mode, the data object should be character type. If the file is opened in binary mode, only binary data should be transferred.

► **READ DATASET**

This statement transfers the data from the file to the program data object one line at a time. The file should be opened for input before using the `READ DATASET` statement. This statement is used in a `DO` loop to read all the lines of the file.

Listing 11.12 shows example code using the `READ DATASET` statement.

```
OPEN DATASET v_file FOR INPUT IN TEXT MODE ENCODING DEFAULT.
DO.
READ DATASET v_file into wa_makt.
IF sy-subrc IS NOT INITIAL.
EXIT.
ELSE.
APPEND wa to itab.
ENDIF.
```

```
ENDDO.  
CLOSE DATASET v_file.
```

Listing 11.12 Reading File

- ▶ **GET DATASET**
Gets the current cursor position in the file.
- ▶ **SET DATASET**
Sets the cursor in the file. The position is specified explicitly or can be set as the end of the file.
- ▶ **TRUNCATE DATASET**
Sets the end of the file.
- ▶ **CLOSE DATASET**
Closes the file. If the file isn't closed explicitly using this statement, it's closed automatically when the program is closed.
- ▶ **DELETE DATASET**
Deletes the file. The system performs implicit checks to ensure the user has the required authorizations to delete the file.

The file names are operating system-dependent because they're physical names. We recommend using platform-independent file names to make your programs portable.

To make your file names platform-independent, use logical file names and paths. These logical file names and paths are linked to physical file names and paths using special tables in customizing that can be maintained using Transaction FILE. The physical file name from a logical file name is generated using the `FILE_GET_NAME` function module.

Using logical file names and paths also enhances security.

11.3.2 Working with Files in the Presentation Layer

Sometimes, users may want to save data locally to their workstations or upload data to a program for processing. For example, a user may want to download the report output as a PDF file or upload a spreadsheet with material data to be posted in the database. In such scenarios, we can use the `CL_GUI_FRONTEND_SERVICES` global class to work with the files in the presentation layer.

Among other methods, the class provides the `GUI_UPLOAD` and `GUI_DOWNLOAD` static methods to upload or download data from or to the user's workstation. The class documentation available for these methods in Class Builder explains their usage clearly.

To download the file to the presentation layer, the file path, file type (binary or ASCII), and internal table that contains the data are passed to the `GUI_DOWNLOAD` method.

To upload a file, an internal table is defined with the number of fields matching the number of columns in the file. This table is then passed to the `GUI_UPLOAD` method along with the file path. The data will be uploaded from the file to the supplied internal table.

Listing 11.13 shows sample code using the `GUI_UPLOAD` method.

```
TYPES : BEGIN OF ty_file,
        field1 TYPE char20,
        field2 TYPE char30,
        field3 TYPE char10,
      END OF ty_file.
DATA : it_file TYPE STANDARD TABLE OF ty_file.
PARAMETERS p_file type string.

CALL METHOD cl_gui_frontend_services=>gui_upload
  EXPORTING
    filename           = p_file
    filetype           = 'ASC'
    has_field_separator = SPACE
  CHANGING
    data_tab           = it_file
  EXCEPTIONS
    file_open_error    = 1
    file_read_error    = 2
    no_batch           = 3
    gui_refuse_filetransfer = 4
    invalid_type       = 5
    no_authority       = 6
    unknown_error      = 7
    bad_data_format    = 8
    header_not_allowed = 9
    separator_not_allowed = 10
    header_too_long    = 11
    unknown_dp_error   = 12
    access_denied      = 13
    dp_out_of_memory   = 14
    disk_full          = 15
```

```

dp_timeout          = 16
not_supported_by_gui = 17
error_no_gui        = 18
others              = 19

```

```

IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.

```

Listing 11.13 GUI Upload

In this section, we looked at how to save data persistently in the application and presentation layers. In the next section, we'll discuss what data clusters are and how to export and import data by using them.

11.4 Data Clusters

Data clusters are a type of storage that can only be used by ABAP programs. A data cluster is a group of data objects that stored in a storage medium. The data of data objects can be stored in different media using data clusters, and this data can then be read by ABAP programs.

Data clusters are in an ABAP-specific format and can only be accessed using `IMPORT` and `EXPORT` statements. A data cluster consists of an administration section and a data section. The *administration section* stores administration information about the data cluster, and the *data section* contains the compressed data objects. The administrative information that is stored in a data cluster varies based on the storage medium.

The following media support data clusters:

- ▶ Byte string
- ▶ Internal table
- ▶ ABAP memory
- ▶ Database
- ▶ Cross-transaction application buffers

The following statements are used when working with data clusters:

- ▶ `EXPORT`
Allows data objects to be written to the storage medium.

- ▶ **IMPORT**
Allows data objects to be extracted from the storage medium.
- ▶ **DELETE**
Deletes any storage medium.
- ▶ **FREE**
Can be used to delete ABAP memory.
- ▶ **IMPORT DIRECTORY**
Generates a list of all data objects that have previously been exported to a database table.

A data cluster stored in a database table is stored persistently, so it can be imported later by ABAP programs. In this section, we'll discuss storing data clusters in a database table.

When storing data clusters in database tables, the table should have a specific structure. These structures are called *INDX structures* (see [Figure 11.11](#)), because they're based on the standard table `INDX` provided by SAP.

The table structure should satisfy the following conditions:

- ▶ The first field of a table should be the `MANDT` field if you want to store data objects specific to a client, and it should be a key field. You don't need to include the `MANDT` field if the data cluster is to be stored cross-client.
- ▶ The second field must be a key field named `RELID` of `TYPE CHAR` and `LENGTH 2`. It stores the two-character area specification supplied while exporting the data cluster. The area specification splits up the rows of the database table into several areas so that data clusters with the same ID can exist multiple times in the database table.
- ▶ The table should contain one key field of `TYPE CHAR` with any name to store the identifier specified in the `EXPORT` statement.
- ▶ The next field must be a key field named `SRTF2` of `TYPE INT4`. It contains the row numbers of a stored data cluster that can extend over several rows and is filled automatically by the system.
- ▶ Any number of components with any names and types may follow. The components will be passed with the `EXPORT` statement. The `IMPORT TO` statement reads these fields out again.

- ▶ The last two components must be named `CLUSTR` and `CLUSTD` and be of types `INT2` and `LRAW` of any length. In `CLUSTR`, the current length of field `CLUSTD` of each row is stored, whereas `CLUSTD` contains the actual data cluster.

Dictionary: Change Table

Transp. Table: ZDEMO_DC Active
 Short Description: INDX-like table

Attributes: Delivery and Maintenance, Field..., Entry help/check, Currency/Quantity Field

Field	Key	Ini...	Data element	Data Type	Length	Deci...
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3	0
RELID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	INDX RELID	CHAR	2	0
ID	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		CHAR	30	0
SRTF2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	INDX SRTF2	INT4	10	0
CLUSTR	<input type="checkbox"/>	<input type="checkbox"/>	INDX CLSTR	INT2	5	0
CLUSTD	<input type="checkbox"/>	<input type="checkbox"/>	INDX CLUST	LRAW	2886	0

Figure 11.11 INDX-Like Table

11.4.1 Exporting Data Clusters to Databases

The `EXPORT TO DATABASE` statement exports the data cluster to the special database that is of table `INDX`.

With the `EXPORT` statement, a two-character area specification and ID are supplied. The area specification splits up the rows of the database table into several areas so that data clusters with the same ID can exist multiple times in the database table.

[Listing 11.14](#) shows sample code using the `EXPORT` statement.

```
DATA: it_kna1 TYPE STANDARD TABLE OF kna1.
SELECT * FROM kna1 INTO TABLE it_kna1 UP TO 10 ROWS.
EXPORT tab = it_kna1
TO DATABASE zdemo_dc(SA)
ID 'TABLE'.
```

Listing 11.14 Exporting Data Cluster to Table

11.4.2 Importing Data Clusters

Using the `IMPORT` statement, the data cluster that was written to the database table `dbtab` in the area `ar` and under the ID specified in `id` using the `EXPORT` statement is imported.

[Listing 11.15](#) shows an example of importing data clusters using the `IMPORT` statement.

```
DATA: it_kna1 TYPE STANDARD TABLE OF kna1.
IMPORT tab = it_kna1
FROM DATABASE zdemo_dc(SA)
ID 'TABLE'.
```

Listing 11.15 Importing Data Clusters

11.5 Security Concepts

Database LUWs and the bundling techniques used with an SAP LUW ensure that data is consistent at the end of a transaction. Along with the lock objects used at the application level, SAP implicitly sets database locks when the Open SQL statements `SELECT`, `INSERT`, `UPDATE`, `MODIFY`, or `DELETE` are used. These lock concepts further ensure data consistency.

Data security is another vital component to consider. Ensuring that only authorized users have accesses to data is vital for business. For example, you wouldn't want an employee to see compensation benefits or personal data of other employees. Open SQL statements in themselves don't perform any authorization checks when they're executed; user authorizations should be checked at the application level before providing access to data.

To facilitate user authorization in application programs, we can use *authorization objects*. An authorization object consists of up to ten authorization fields. We use these authorization fields to check if the user has authorization for a certain activity in the system. For example, [Figure 11.12](#) shows the authorization object `S_CARRID` with couple of authorization fields—namely, `CARRID` and `ACTVT`. The `CARRID` field expects the airline code for which the authorization should be checked for the user.

The `ACTVT` fields expects the activity that should be checked, such as `CHANGE` or `DISPLAY`. Clicking the `PERMITTED ACTIVITIES` button shows you the available activities that can be checked. This allows you to check for various activities based on

the user actions in the application. For example, you may want to give display access to a certain user but not change access.

In the example in Figure 11.12, we can use the `S_CARRID` authorization object to check if the user has `CREATE` OR `GENERATE` authorization. This check can be performed before creating new records with the `INSERT` statement.

Similarly, we can check if the user has `DISPLAY` authorization before executing the `SELECT` statement to select the data.

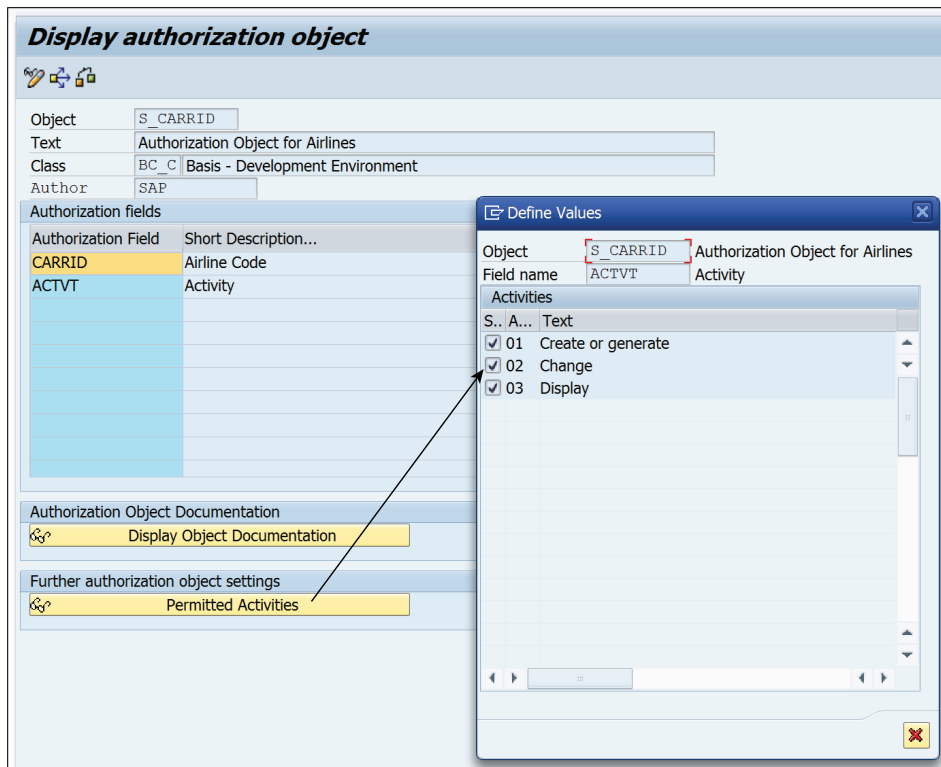


Figure 11.12 Authorization Object

The authorization object is called in the application program using the `AUTHORITY-CHECK OBJECT` statement. The `PATTERN` button in ABAP Editor can be used to insert the required syntax. Listing 11.16 shows sample code to check if the user has display authorization for a particular airline.

```

PARAMETERS p_carrid TYPE spfli-carrid.
AT SELECTION-SCREEN.
AUTHORITY-CHECK OBJECT 'S_CARRID'
ID 'CARRID' FIELD p_carrid
ID 'ACTVT' FIELD '03'.

IF sy-subrc <> 0.
  MESSAGE 'No Authorization for the Airline' TYPE 'E'.
ENDIF.

```

Listing 11.16 Checking for Authorization

Authorization objects are created by SAP Basis administrators, and the authorization profile is maintained in the user master record. The `AUTHORITY-CHECK` statement checks against the user profile to identify if the user has the required authorization. It's always a good practice to perform the authorization check early in the program.

11.6 Summary

In this chapter, we discussed various ways in which data can be stored persistently in the database and in files. We covered concepts such as database LUWs and SAP LUWs that are used to ensure data consistency.

We also discussed the ABAP Object Services framework, which is an object-oriented framework that allows you to work with relational databases without having to use Open SQL statements in the application program.

Even though the database is the most preferred persistence source, we often store application data in the other two layers. We discussed how data can be stored in and accessed from the application server as well as the user's workstation in the presentation layer.

We concluded this chapter with a discussion of persistent data security concepts and how to ensure users have the required privileges to work with data.

In the next chapter, we'll discuss dialog programming, which enables us to create screens, flow logic, menu bars, and ABAP code that can be linked to make an application.

Dialog programming involves creating a collection of screens, flow logic, menu bars, and ABAP code that when linked together builds an application. At the end of this chapter, you'll know how to create transactions and develop dialog programs.

12 Dialog Programming

Dialog programming is very different from the reporting program creation you've seen so far in this book. Whereas reports traditionally provide read-only access to a database, dialog programs allow a user to create and update data. For example, if the user needs an application to create a sales order in which he wants to capture information about the order and save it to the database, we need to develop a dialog program to enable the user to capture various information about the order, validate the data, and save it to the database. However, both report programs and dialog programs can work interchangeably, and both can be used to both read and update data.

Dialog programs are also known as *transactions*. Dialog programs involve various screens that are called to capture or show data to the user. These programs are accessed through the SAP GUI and form the user interface. Using dialog programs, the user can navigate between screens and interact with the SAP ERP system.

In this chapter, we'll discuss screen processing and how to create various screen elements using Screen Painter. As previously discussed in [Chapter 3](#), *Screen Painter* is an ABAP Workbench tool used to design dialogs in the SAP system. We'll also discuss how to create various screen objects (GUI statuses) such as menus, buttons on the application toolbar, and so on via the Menu Painter. GUI statuses extend the functionality of the application.

When the screen is processed, the ABAP runtime environment triggers specific screen events. We'll start our discussion in this chapter with an introduction to screen events in [Section 12.1](#), and we'll examine the different attributes of a screen in [Section 12.2](#).

A screen consists of different screen elements, which allow the user to interact with the program. The screen elements we'll discuss in this chapter are as follows:

- ▶ Basic screen elements (text fields, checkboxes, radio button, and push buttons): [Section 12.3](#)
- ▶ Input and output fields: [Section 12.4](#)
- ▶ List boxes: [Section 12.5](#)
- ▶ Tables controls: [Section 12.6](#)
- ▶ Tablestrip controls: [Section 12.7](#)
- ▶ Subscreens: [Section 12.8](#)

We will also provide instructions for working with screens in [Section 12.9](#). Even though there are many screen elements that allow us to design screens, they sometimes lack the necessary ingredients to provide a rich user experience, or more may be needed when customer requirements demand complex screen elements. To fill this gap, SAP provides the *Control Framework*, which helps us design better UIs and provide rich user experiences. You'll learn how to use the Control Framework in [Section 12.10](#).

Dialog programming is one of the topics that can become complicated easily, especially when you develop an application that consists of many screens. To grow more comfortable with dialog programming, you'll need to practice developing these applications. Therefore, [Section 12.11](#) concludes the chapter with a practice exercise.

12.1 Screen Events

Before you develop dialog programs, you first need to understand how dialogs are processed in SAP. As previously discussed, there are three types of screens that form SAP GUI. *Selection screens* and *list screens* are automatically created via specific keywords in ABAP programs. For example, a selection screen is generated by using the `PARAMETERS` or `SELECT-OPTIONS` keyword, and a list screen is created by using the `WRITE` keyword.

The third type of screen, called a *general screen* (or just a *screen*), is created manually via Screen Painter.

Figure 12.1 shows a general screen from Transaction VF03. As you can see in Figure 12.1, the general screen provides greater flexibility to design the screen with various screen elements, such as input fields, dropdown lists, tables, and more per your requirements, compared to selection screens or list screens.

Item	Description	Billed Quantity	SU	Net value	Material
10	Nose Landing Gear Assembly	1 EA		50,000.00	IAD-108000

Figure 12.1 General Screen with Various Screen Elements

Recall the architecture of the application server that we discussed in [Chapter 2](#): ABAP programs are processed by the work process in the application server. The work process consists of a *screen processor*, *ABAP processor*, and *database interface*, as shown in [Figure 12.2](#).

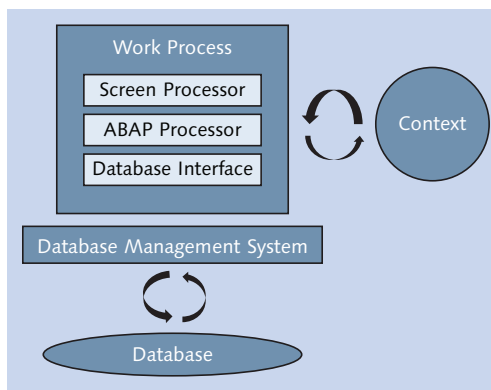


Figure 12.2 Components of Work Process

The screen processor is responsible for communication between the work process and SAP GUI. It processes the *screen flow logic* that makes up a screen and

calls different dialog modules. Dialog modules are maintained in the ABAP program between the `MODULE...ENDMODULE` statements and are called from the screen flow logic using the `MODULE` statement. The screen processor calls the dialog modules from the screen flow logic (see [Figure 12.3](#)) and ensures that the data is transferred from the screen to the flow logic.

In other words, the screen flow is controlled by the screen processor via various *screen events*. Similar to selection screen events and list events, which are called respectively when the selection screen and list screens are processed, the screen processor triggers different screen events when the general screen is processed.

Once the screen processor calls the dialog module from the screen flow logic, the control will be passed to the ABAP processor (see [Figure 12.2](#)), which takes care of executing the ABAP statements within the dialog module. If the ABAP processor encounters any statements that require interaction with the database, control is handed over to the database interface in the work process, which takes care of interacting with the underlying database.

[Figure 12.3](#) shows the screen flow logic in which screen events are maintained. When a screen event is triggered by the ABAP runtime environment, the dialog modules that are called within the event block are executed. The following are the four screen events that are triggered when a screen is processed:

► **Process before output (PBO)**

A PBO event is triggered before showing the screen to the user. We use this event to prepare the screen, such as by assigning default values or dynamically displaying/hiding certain fields on the screen.

► **Process after input (PAI)**

A PAI event is triggered after showing the screen to the user, when the user performs any action on the screen. This event allows you to respond to user actions on the screen.

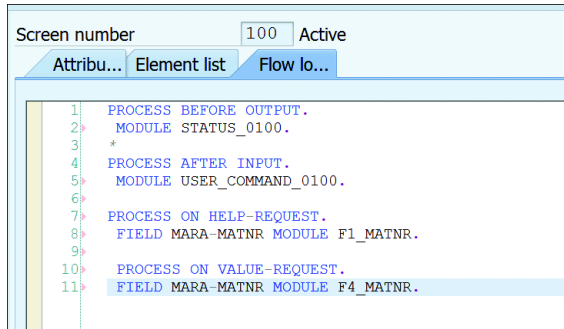
► **Process on help request (POH)**

A POH event is triggered when the user selects `[F1]` (or *field*) help for the screen field. We use this event to provide field help for the screen field.

► **Process on value request (POV)**

The POV event is triggered when the user selects `[F4]` (or *input*) help for the screen field. We use this event to provide input help for the screen field.

In [Figure 12.3](#), all four screen events listed are maintained with their respective event statements. Dialog modules are called from within the event block using the `MODULE` statement. The dialog modules that are called from the screen flow logic should be maintained between the `MODULE...ENDMODULE` statements in the main program to which the screen belongs (refer ahead [Figure 12.5](#)).



```
1 PROCESS BEFORE OUTPUT.
2 MODULE STATUS_0100.
3 *
4 PROCESS AFTER INPUT.
5 MODULE USER_COMMAND_0100.
6
7 PROCESS ON HELP-REQUEST.
8 FIELD MARA-MATNR MODULE F1_MATNR.
9
10 PROCESS ON VALUE-REQUEST.
11 FIELD MARA-MATNR MODULE F4_MATNR.
```

Figure 12.3 Screen Flow Logic

The PBO event is triggered before the screen is shown to the user, and the remaining events are triggered after the screen is shown. The PBO event allows you to prepare the screen before it's displayed to the user, and other events allow you to react to user actions on the screen. For example, if you want to disable a field on the screen or hide some of the screen elements dynamically, you can use the PBO event.

Similarly, you can capture user inputs and process the application in the PAI event. The POH and POV events allow us to extend an application's functionality by providing additional help for the screen fields. This helps the user understand what input is expected in each screen field or search for a valid input. The field help documentation along with input helps provides a consistent user experience for the user throughout the SAP system.

12.2 Screen Elements and Flow Logic

When we develop dialog programs, we work with ABAP Workbench tools to create different objects. For example, we use Screen Painter (Transaction SE51) to create and design a screen, and we use Menu Painter (Transaction SE41) to define the GUI status for the screen.

Dialog transactions are created using module pool programs. Unlike executable programs, module pool programs can't be executed directly using the program name. Module pool programs are always executed by assigning a transaction code to the program.

To assign a transaction code to the program, we use Transaction SE93. Instead of opening all of these various ABAP Workbench tools in different windows, we can use Object Navigator (Transaction SE80) to work with different ABAP Workbench tools in one place. Object Navigator provides an integrated development environment (IDE), as shown in [Figure 12.4](#).

Object Navigator consists of a navigation area on the left-hand side that groups various development objects of the program into different folders of a tree structure. On the right-hand side is the tool area, where the selected objects are opened using the respective ABAP Workbench tool. For example, if you expand the SCREENS folder from the navigation area shown in [Figure 12.4](#) and select a screen number, the screen will automatically open in Screen Painter in the tool area.

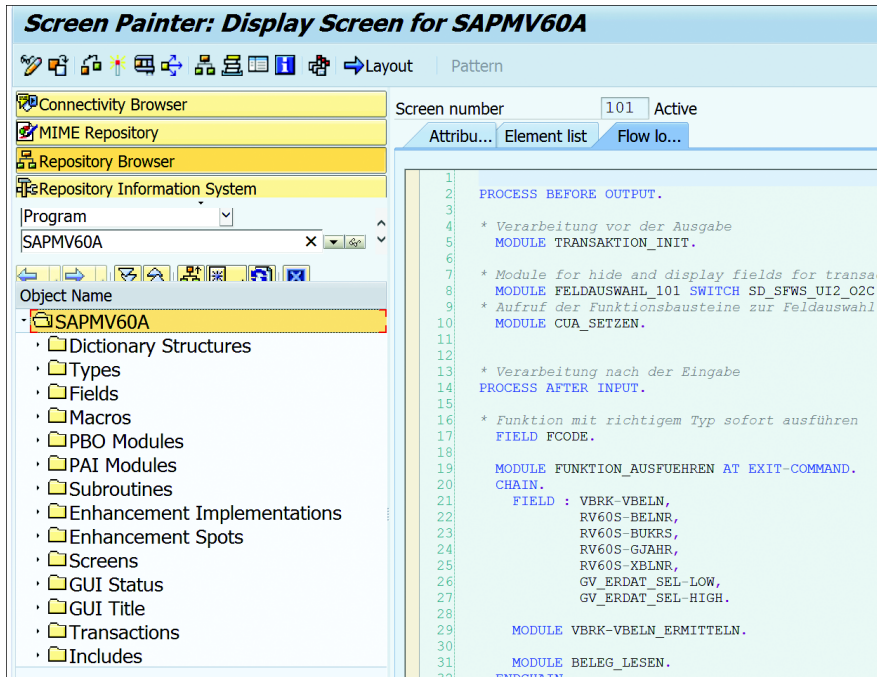


Figure 12.4 Object Navigator

In this section, we'll discuss the various components of a dialog program. Then, we'll explain the use of screens and the process of creating screens for ABAP programs. Finally, we'll provide steps for using and implementing screen elements.

12.2.1 Components of a Dialog Program

Dialog programs consist of various objects. These objects aren't executed on their own; instead, they're linked to a main program (dialog program) from which these objects are called based on the screen flow logic.

The various objects of a dialog program include the following:

► Screens

Screens are central to dialog programming. A dialog program may consist of many screens. These screens are created using Screen Painter, and each screen belongs to a program. In other words, each program defines its own screens, and one screen can't be shared with multiple programs. Each program screen is identified by a number between 1 and 9999. We'll walk through the step-by-step process of creating screens using Screen Painter in the next section.

A screen consists of screen elements and flow logic. *Flow logic* controls the screen flow by calling various modules in the main program to which the screen belongs. The various UI elements, such as input/output fields, checkboxes, radio buttons, dropdown list boxes, table controls, and so on, are called *screen elements*.

► GUI status

Each screen can load its own GUI status. A GUI status controls the *menu bar*, *standard toolbar*, and *application toolbar* of the screen and passes the user actions on those elements as function codes to the program.

The GUI status is defined independently of the screen, and each GUI status belongs to a program. Each screen of the program can load a different GUI status defined for the main program to which the screen belongs, or it can load an external GUI status defined for another program.

A GUI status is defined using Menu Painter. It's an independent component of a program. We'll discuss creating and using GUI statuses in [Section 12.9.2](#).

► ABAP program

All processing logic is maintained in the ABAP program in *dialog modules*, processing blocks that are maintained between `MODULE` and `ENDMODULE`. These

dialog modules are called from the screen flow logic for a screen event via the `MODULE` statement (see [Figure 12.5](#)).

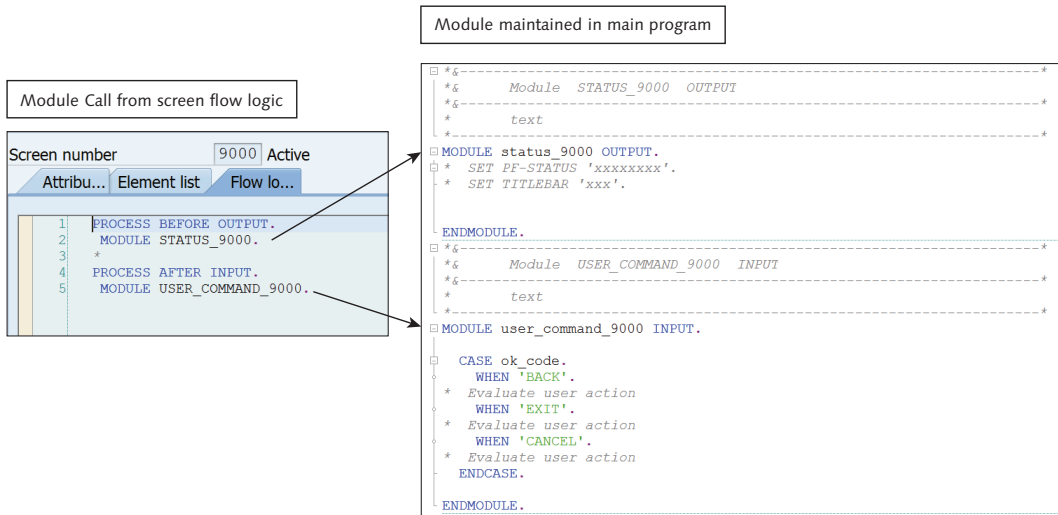


Figure 12.5 Dialog Modules

Dialog modules can't keep any local data, so any data declarations within a dialog module are added to the global declaration area of the program. Dialog modules can't be called externally from external program screens, nor do they support any parameter interface. Modules called from PAI end with `INPUT` and modules called from PBO end with `OUTPUT` (e.g., `MODULE xyz INPUT` or `MODULE abc OUTPUT`).

Data is transferred from the screen to the program during a PAI event and from the program to the screen during a PBO event. When we define a data object of the program with the same name as the screen field, the data is automatically transferred from screen to program in the PAI event and stored in the identically named data object of the program. Similarly, the data in a data object of the program is transferred to the identically named screen field in the PBO event.

For example, if we have an input field on the screen with the name `V_MATNR` in which the user can input a material number, then we need to define a data object in the program with the same name, `V_MATNR`, to store the value entered by the user.

Whatever value the user inputs in the `V_MATNR` screen field will be transferred to the program into the `V_MATNR` data object in a PAI event. Similarly, whatever data is contained in the `V_MATNR` data object of the program will be transferred to the `V_MATNR` screen field in PBO.

Each screen belongs to an ABAP program and can call modules of the same program. The purpose of the main program is to maintain different dialog modules that are called from the *screen flow logic*. For this purpose, a module pool program is used. The role of the program is limited to maintaining a pool of modules—hence the name *module pool*. Module pool programs are always executed through a transaction code to which the initial screen of the program is linked. When the transaction is called, it triggers the screen flow logic of the initial screen, from which other program screens can be called.

Because screens are also supported by executable programs and function groups, they can be used as main programs as well. With module pool programs, the first processing block called is always a dialog module within the PBO event of the initial screen.

With executable programs and function groups, the first processing block called could be a runtime event or a procedure call. The `CALL SCREEN` statement within these processing blocks then starts the screen sequence.

[Listing 12.1](#) shows example code in an executable program. In this example, when the program is executed, the ABAP runtime environment triggers the `START-OF-SELECTION` event and executes the `CALL SCREEN 100` statement. When the `CALL SCREEN` statement is executed, the screen processing starts by calling the PBO event of screen 100 and subsequently displaying the screen. At this point, the program behaves like a module pool program.

```
REPORT ZDEMO_PROGRAM.
DATA v_matnr TYPE matnr.

START-OF-SELECTION.
CALL SCREEN 100.
*&-----*
*&      Module STATUS_0100  OUTPUT
*&-----*
MODULE status_0100 OUTPUT.
*  SET PF-STATUS 'xxxxxxx'.
*  SET TITLEBAR 'xxx'.
ENDMODULE.
*&-----*
*&      Module USER_COMMAND_0100  INPUT
```

```

*&-----*
MODULE user_command_0100 INPUT.

ENDMODULE.

```

Listing 12.1 Starting Screen Processing in Report Program

► Transaction code

A dialog transaction links the program with its initial screen. When the transaction code is called, it starts the screen sequence by calling the flow logic of the initial screen. From the flow logic of the initial screen, other screens can be called. [Figure 12.6](#) shows Transaction VF01, which links the main program and the initial screen of the billing document application. As previously mentioned, transaction codes are created using Transaction SE93.

The screenshot shows the 'Display Dialog Transaction' configuration screen for transaction code VF01. The fields are as follows:

Transaction code	VF01
Package	VF
Transaction text	Create Billing Document
Program	SAPMV60A
Screen number	102
Authorization Object	

There is a 'Values' button next to the Authorization Object field. At the bottom, there is a checked checkbox for 'Maintenance of standard transaction variant allowed'.

Figure 12.6 Transaction VF01

► Screen flow logic

The *screen flow logic* controls the screen flow and consists of different screen events (as discussed in [Section 12.1](#)) from which dialog modules are called. You can only call dialog modules from the screen flow logic, and these dialog modules should be maintained in the main program of the screen.

► Dynpro

The screen along with its flow logic is called a *dynpro* (dynamic program). Each dynpro represents a dialog step of the program.

► Dialog step

A *dialog step* starts with the PAI of one screen and ends with the PBO of the next screen. Each dialog step comprises programming logic between two successive

screens. Individual dialog steps of a program can be executed by different work processes in the application server, and a program consists of multiple dialog steps.

During a dialog step, the screen remains inactive for input. For example, the screen is shown to the user in the presentation layer. When the user inputs data on the screen and presses `[Enter]`, this action initiates the dialog step by calling the PAI event of the screen. The user action is processed by the work process in the application server.

While the work process is processing the request, the screen is visible in the presentation layer but can't take any input (i.e., the screen is inactive). After the work process has completed processing the request, the same screen or another screen is sent back by the work process, completing the dialog step. Once the screen is sent back to presentation layer, the screen becomes active for user input. At this point, the work process is free to take another request from same user or a different user. [Figure 12.7](#) depicts this request-response cycle.

The number of work processes available in the system is often less than the number of users logged on to the system. Because a work process isn't dedicated to one user or program, many users can use the system simultaneously. The dispatcher is responsible for distributing all dialog steps among the available work processes in the application server.

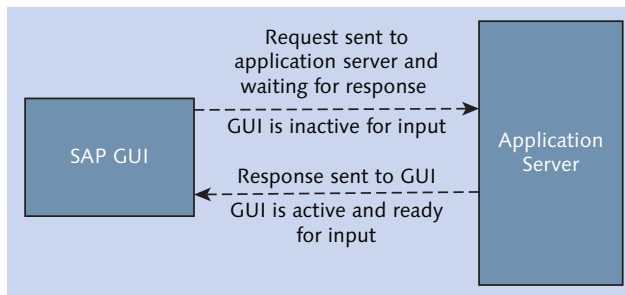


Figure 12.7 Request-Response Cycle

Now that we've looked at the different components of a dialog program, in the next section let's hone in on the use of screen components and how to create these screens.

12.2.2 Screens

A screen is a repository object that belongs to a program. Screens are dynamic programs (dynpros) that contain their own data objects. The data objects of the screen are called *screen fields* and are linked to the I/O fields on the screen (see [Section 12.4](#) for more on I/O fields). The system passes the data between the screen fields and identically named data objects of the program. This allows us to transfer data from the program to the screen or from the screen to the program.

The screen can be defined for executable programs, module pool programs, and function pools. The process of creating a screen is similar irrespective of the program type. In this chapter, we'll work with module pool programs.

Because a screen is a component of a program, we first need to create a program and then create a screen for a program. We'll use Object Navigator for the examples in this chapter, because it's helpful to work with all the objects in one place.

The following steps illustrate how to create a screen for a program:

1. Open the program in Object Navigator. Right-click the program name and select **CREATE • SCREEN**, as shown in [Figure 12.8](#).

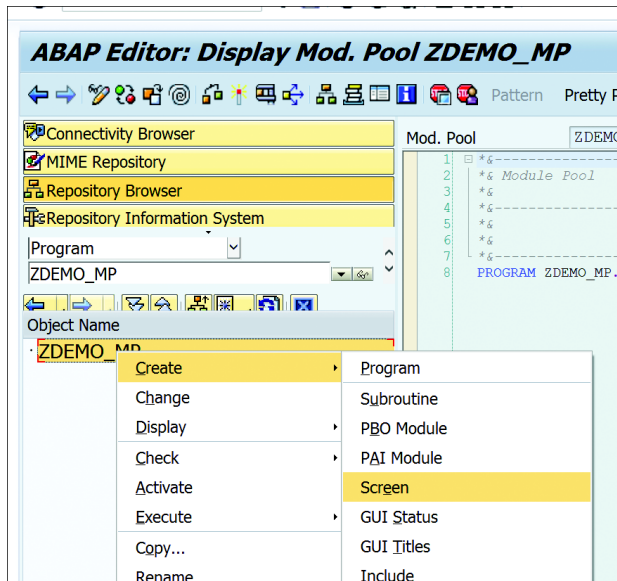


Figure 12.8 Create Screen Menu Path

- On the CREATE SCREEN dialog box that appears (see [Figure 12.9](#)), enter the SCREEN NUMBER and click the green checkmark. The screen number should be unique for the program and can be up to four digits. Screen numbers above 9000 are reserved for SAP customers. The initial screen of the program usually ends with 100.

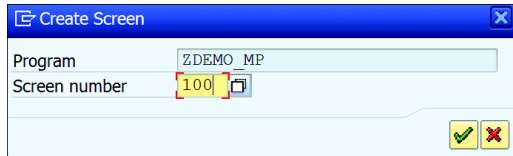


Figure 12.9 Create Screen Window

- The system will display the CHANGE SCREEN ATTRIBUTES screen (see [Figure 12.10](#)). This screen consists of three tabs: ATTRIBUTES, ELEMENT LIST, and FLOW LOGIC.

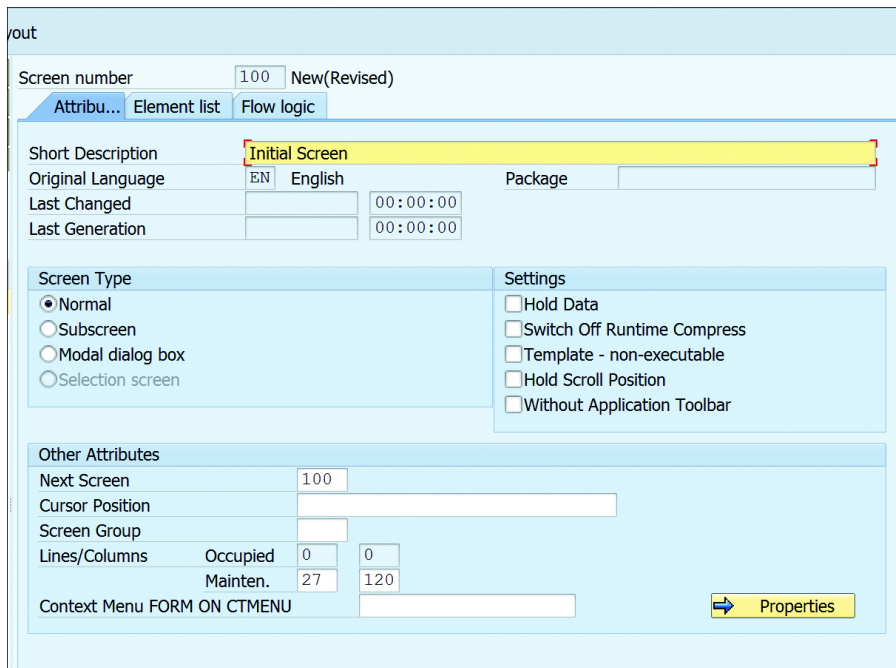


Figure 12.10 Change Screen Attributes

4. In the ATTRIBUTES tab, various screen attributes can be maintained:

▶ SCREEN TYPE

In this section, you define the type of screen you want to create. For the screen type, the SELECTION SCREEN option is grayed out, because selection screens aren't defined in Screen Painter but rather are generated using PARAMETERS or SELECT-OPTIONS statements in an ABAP program. This option is automatically selected for the generated selection screen. You can choose one of the following options:

- NORMAL: Defines a normal screen. This is the default option and is the type of the main screen. A normal screen acts as a main screen that can have a GUI status and can be called independently in the program or via a transaction code.
- SUBSCREEN: This option defines the screen as a subscreen. A subscreen can't be called individually; it's always loaded on the main screen.
- MODAL DIALOG BOX: This type of screen is used to display a pop-up screen.

▶ SETTINGS

Under this section, various settings can be maintained for the screen:

- HOLD DATA: If this option is selected, the screen fields will hold data if the user has enabled it under SYSTEM • USER PROFILE • HOLD DATA. This holds any entries made on the screen and will redisplay the data when the screen is processed.
- SWITCH OFF RUNTIME COMPRESSION: If this setting is enabled, the screen isn't compressed during runtime. That is, if compression is enabled, the system will automatically remove any lines in which fields were made invisible dynamically. If compression is turned off, then it will show empty lines in such scenarios.
- TEMPLATE—NON-EXECUTABLE: This setting is used by SAP internally. This is selected if the screen is to be used as a template for other screens and never executed.
- HOLD SCROLL POSITION: Enabling this setting will make the screen hold the scroll position. This is especially useful for large screens. When you come back to the screen after processing another screen, the system will hold the last scroll position.

- **WITHOUT APPLICATION TOOLBAR:** Enabling this setting will hide the application toolbar on the screen.
- ▶ **OTHER ATTRIBUTES**
- Under the **OTHER ATTRIBUTES** section, the following options can be set:
- **NEXT SCREEN:** You can input the next screen that should be called after PAI of the current screen. This setting can be overridden by using the **SET SCREEN** statement in one of the PAI modules of the current screen. If the same screen number is entered as the next screen, then the system will call the same screen PBO event after the PAI event and display the same screen.
 - **CURSOR POSITION:** Input the screen field where the cursor should be initially placed when the screen is called.
 - **SCREEN GROUP:** Several screens can be assigned to a screen group to perform any modifications consistently on all screens of the group. The screen group name can be accessed by using the **SY-DNGR** system field.
 - **LINES/COLUMNS:** This setting shows the screen size occupied by the screen elements. This information is useful when calling the screen as a modal dialog box.
 - **CONTEXT MENU FORM ON CTMENU:** You can assign a context menu for the screen in this field.
5. In the **ELEMENT LIST** tab, all the screen elements that can be maintained will be listed. [Figure 12.11](#) shows the **ELEMENT LIST** tab with all the screen elements belonging to the screen listed. The screen element details are automatically populated in the **ELEMENT LIST** tab when we create the screen elements using the layout editor. We'll discuss this tab further when we create screen elements in the next section.
6. The **FLOW LOGIC** tab (see [Figure 12.12](#)) allows you to maintain the screen flow logic. The screen events are maintained in the flow logic and are then called by the screen processor. Dialog modules can be called from the flow logic using the **MODULE** statement.

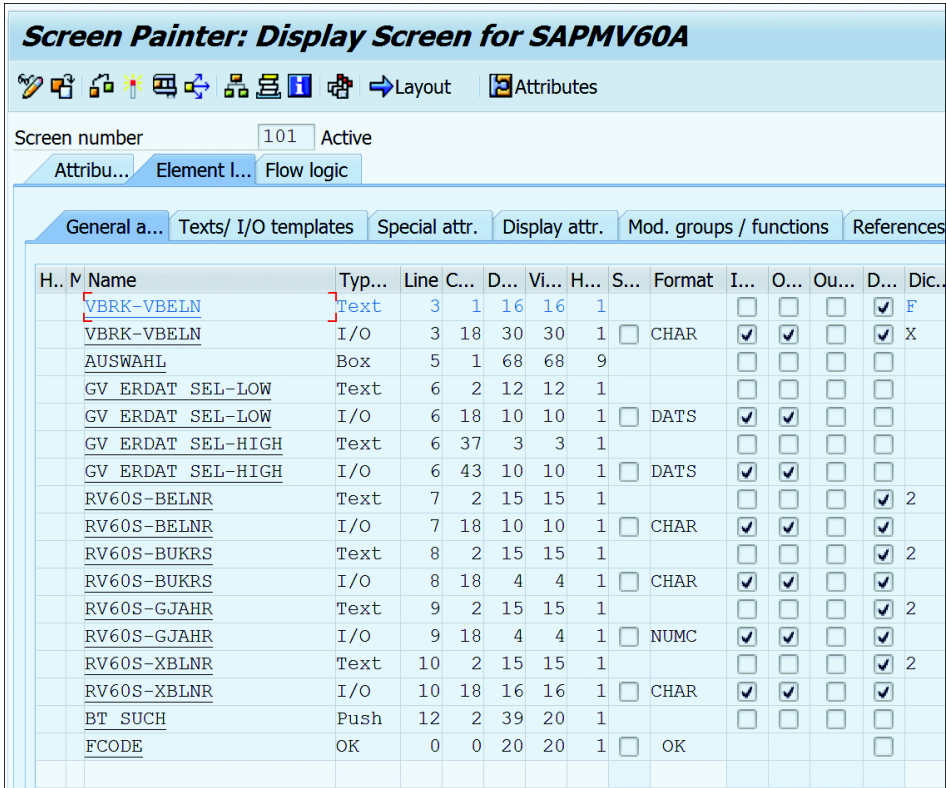


Figure 12.11 Element List Tab

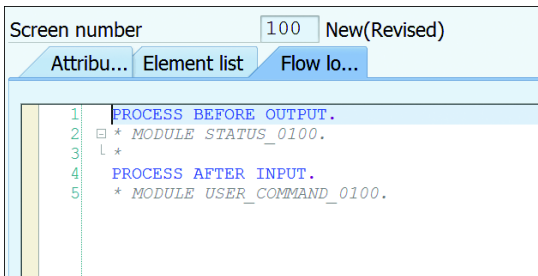


Figure 12.12 Flow Logic Tab

7. Once all the settings are maintained, activate the screen by clicking the ACTIVATE icon in the application toolbar.

12.2.3 Screen Elements

The various screen elements of a dialog program make up the user interface (UI) of the screen. Screen elements are created in graphical or alphanumeric mode in Screen Painter.

The graphical layout editor in Screen Painter allows you to create screen elements via drag and drop functionality. You can access the graphical layout editor by clicking the LAYOUT button in the application toolbar, as shown in [Figure 12.13](#).

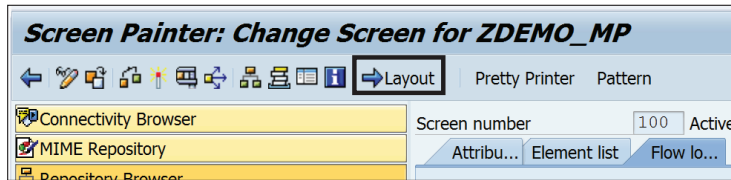


Figure 12.13 Layout Button

To open the editor in alphanumeric mode, from within Screen Painter follow the menu path UTILITIES • SETTINGS and select the SCREEN PAINTER tab. In the SCREEN PAINTER tab, deselect the GRAPHICAL LAYOUT EDITOR option, as shown in [Figure 12.14](#), and click the OK button (green checkmark). Once the GRAPHICAL LAYOUT EDITOR option is deselected, click the LAYOUT button shown in [Figure 12.13](#) to open the editor in alphanumeric mode.

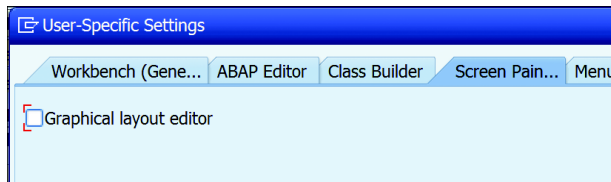


Figure 12.14 Deselecting Graphical Layout Editor

The graphical layout editor (see [Figure 12.15](#)) provides an easy-to-use interface in which all the screen elements can be added using drag and drop functionality.

The alphanumeric layout editor doesn't support drag and drop; it requires fields to be inserted manually. To insert a screen element, position the cursor on the editor where the screen element should be inserted and follow the menu path EDIT • CREATE ELEMENT, as shown in [Figure 12.16](#). You are then shown a list of screen elements that you can create via the context menu.

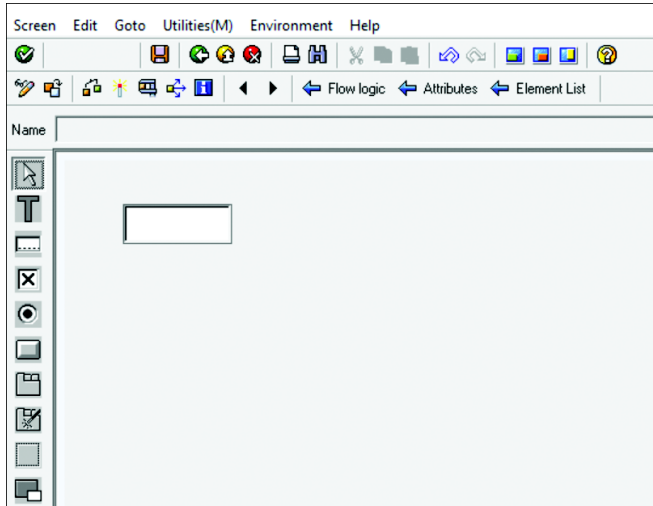


Figure 12.15 Graphical Layout Editor

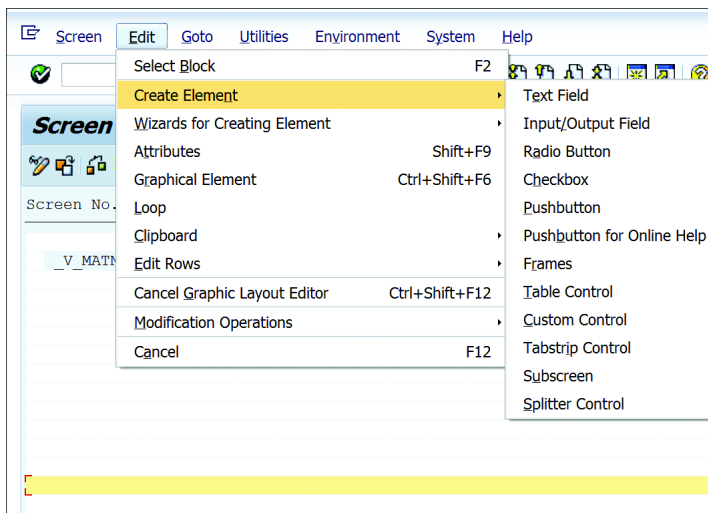


Figure 12.16 Alphanumeric Layout Editor

GUI Dependency

The graphical layout editor in Screen Painter is available only for SAP GUI for Windows and can be started in the SAP NetWeaver Application Server 6.10 with SAP GUI release 6.10 or later.

In SAP NetWeaver 7.0, the graphical layout editor can be started if you have SAP GUI release 6.20 or later, patch 56, or release 6.40, patch 8. In all other cases, the system starts the alphanumeric full-screen editor automatically.

Figure 12.17 shows the different screen elements that can be maintained; the basic screen elements include the following:

▶ **Text fields**

Used to provide labels for other fields and are not editable.

▶ **Dropdown list boxes**

A special type of input/output field that show a list of possible entries from which the user can select. The user won't be able to input a free value in this field; he must select from the available options.

▶ **Checkboxes**

Used to allow the user to select multiple options in a group. If a function code is assigned to a checkbox, the system will call the PAI event immediately; otherwise, the user can continue making other inputs and call the PAI by pressing or clicking a button.

▶ **Radio buttons**

Used to allow the user to select any one of the options in a group. Radio buttons always belong to a group. If a function code is assigned to a radio button/group, the system will call the PAI event immediately; otherwise, the user can continue making other inputs and call the PAI by pressing or clicking a button.

▶ **Push buttons**

Associated with a function code. When the user clicks the push button, the system triggers the PAI event and the function code is passed to the program.

▶ **Boxes**

Used to group different screen elements. This allows the screen elements to be arranged into various groups and provide greater visual appeal for the user. These are used purely for display purposes and can contain a header to show the group title.

Other, more specialized screen elements include the following:

▶ **Input/output (I/O) fields**

Used to enter or display data.

► **Table controls**

Used to display data in a tabular format—for example, to display all the line items of a document.

► **Tabstrip controls**

Allow you to create multiple tabs on a single screen for easy navigation. Each tab of the tabstrip control loads a subscreen.

► **Subscreen area**

You can create a subscreen area on a screen so that another screen can be displayed there as a subscreen at runtime.

► **Custom container**

A custom container area can be marked on a screen where a custom control can be displayed.

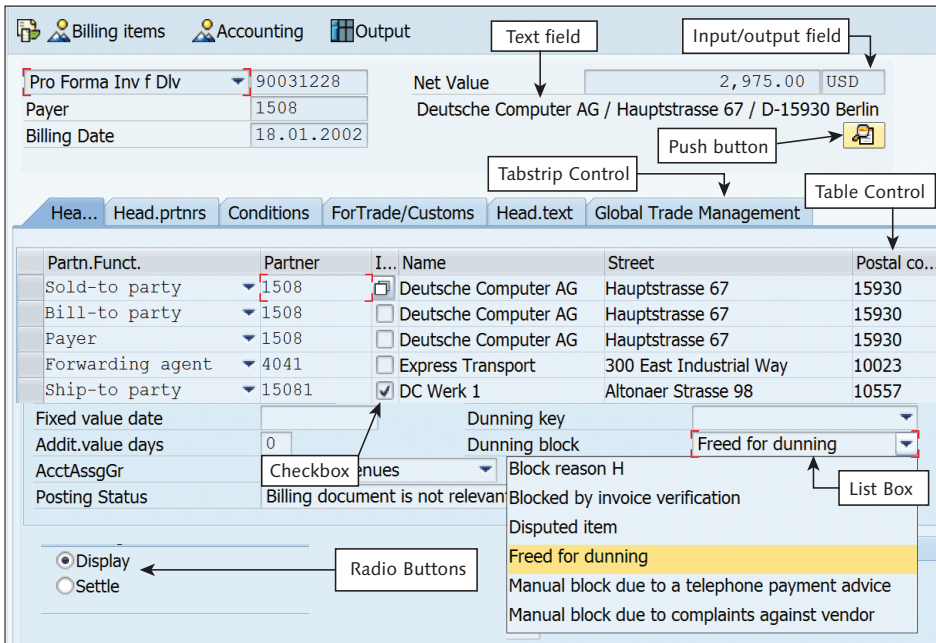


Figure 12.17 Screen Elements

Now that we've reviewed the various elements of dialog screens, in the next section we'll examine how to create the basic elements we've discussed: text fields, checkboxes, radio buttons, and push buttons. We'll look at the more complex elements in later sections.

12.3 Basic Screen Elements

Now that we have an understanding of the screen elements that can be created on a screen, let's go ahead and create some of the basic elements.

As previously discussed, to create screen elements we call the layout editor by clicking the LAYOUT button (see [Figure 12.13](#)) in Screen Painter. The graphical layout editor is part of the SAP GUI, and you can use it if you're using a supported SAP GUI version on the Windows platform (SAP NetWeaver Application Server 6.10 with SAP GUI release 6.10 or later). If you aren't using a supported platform, the system will open the alphanumeric editor. Both editors offer the same functionality but different interfaces. The graphical layout editor makes it easy to create screen elements using drag and drop.

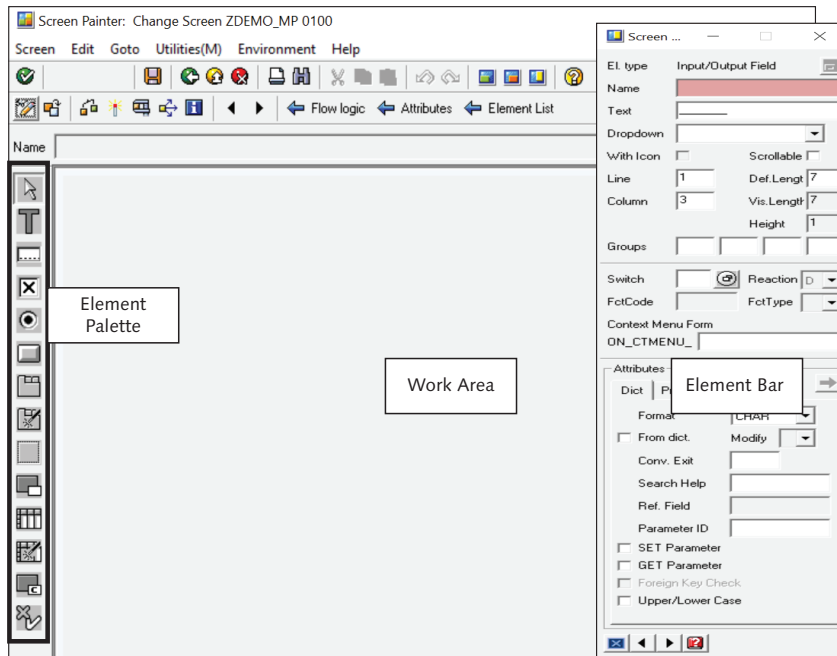


Figure 12.18 Graphical Layout Editor

The graphical layout editor has three components, as shown in [Figure 12.18](#):

- ▶ **Element palette**
Allows you to create all the screen elements.

► **Work area**

The area in which you design the screen by adding various screen elements.

► **Element bar**

Shows the major attributes associated with a screen element when you double-click it.

In this section, we'll walk through the steps for creating text fields, checkboxes, radio buttons, and push buttons via the graphical layout editor in Screen Painter.

12.3.1 Text Fields

As previously mentioned, *text fields* provide labels for other fields and are not editable. To create a text field, follow these steps:

1. Select the element from the element palette and drag it to the work area, as shown in [Figure 12.19](#).

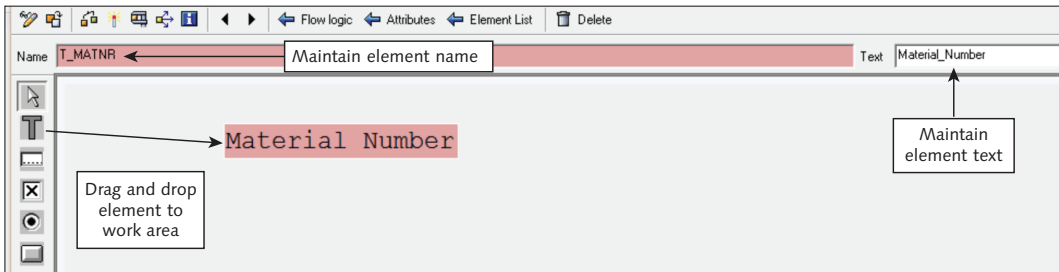


Figure 12.19 Creating a Text Field

2. Adjust the length and height of the element if required.
3. Maintain a name for the screen element.
4. Maintain text for the screen element. This text will be displayed inside the element.
5. Double-click the screen element to open the element bar in order to maintain further attributes. If the text element is displayed as a label for an I/O field, select AS LABEL ON LEFT/AS LABEL ON RIGHT in the DISPLAY tab under ATTRIBUTES.

12.3.2 Checkboxes and Radio Buttons

To create a checkbox or radio button, the corresponding program/dictionary field should be of type C and length 1. When the checkbox or radio button is selected, the corresponding program data object will be passed X. We can identify if the checkbox or radio button is selected by checking the corresponding program field for X or a space.

In the SCREEN PAINTER: DICT./PROGRAM FIELDS window, select the checkbox (CHKB) or radio button (RADB) option under the COPY AS column (see [Figure 12.20](#)). These options will be enabled only if the field is of TYPE C and LENGTH 1.

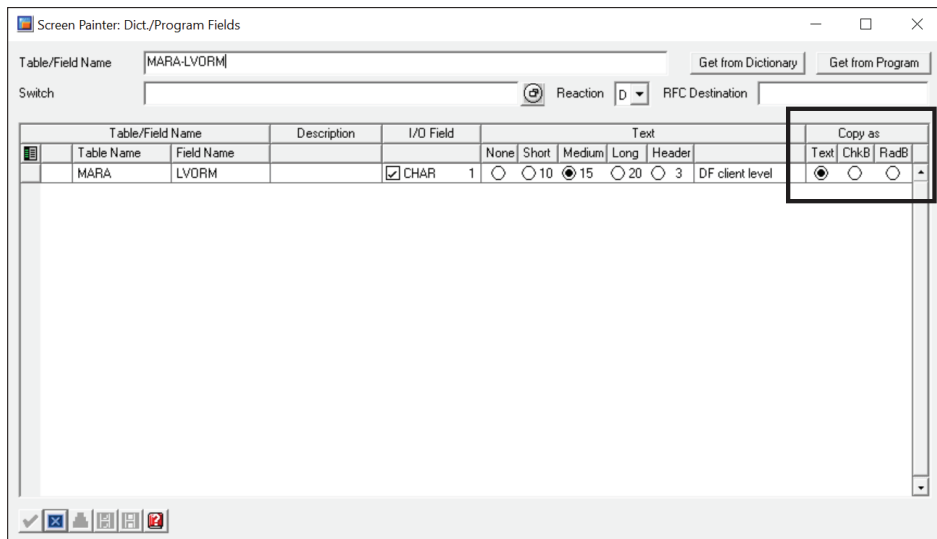


Figure 12.20 Copying Checkbox/Radio Button

The checkbox and radio button elements can also be created by selecting them from the element palette. The radio buttons should be grouped so that only one of the buttons in the group can be selected at a time. To group the radio buttons, select all the buttons that should be part of the group (by holding the **[Ctrl]** key and clicking each element), right-click, and select RADIO BUTTON GROUP • DEFINE from the context menu, as shown in [Figure 12.21](#).

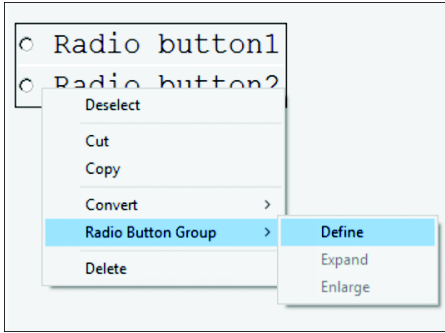


Figure 12.21 Defining Radio Button Group

Optionally, the checkboxes and radio buttons can be assigned a function code in the element bar, which will then trigger a PAI event the moment the checkbox or radio button is selected. This is useful if you want enable/disable fields dynamically when a checkbox or radio button is selected, for example.

To assign the function code, double-click the screen element to open the element bar and enter the function code (up to twenty characters long) in the FCTCODE field (see [Figure 12.22](#)).

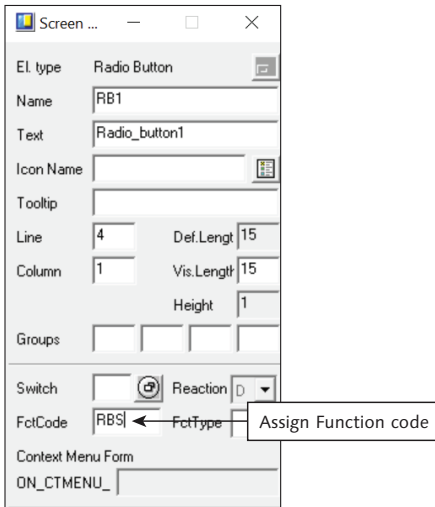


Figure 12.22 Assigning Function Code

12.3.3 Push Button

Before SAP ERP release 4.6, push buttons were the only UI elements that triggered PAI when the user clicked on them. However, checkboxes, radio buttons, and list boxes can now also trigger PAI. Each push button is assigned a function code that is passed to the program when PAI is triggered. You can maintain text for the push button statically in the attributes, or it can be set dynamically in the program.

To set the push button text dynamically, you need to define a data object in the program with the same name as the push button. The text that is filled in this data object will be displayed as the push button text. An icon can also be assigned to a push button in the element bar.

Double-click the screen element to open the element bar. Maintain the push button name in the NAME field, the text in the TEXT field, select the icon in the ICON NAME field, and assign a function code in the FCTCODE field, as shown in [Figure 12.23](#).

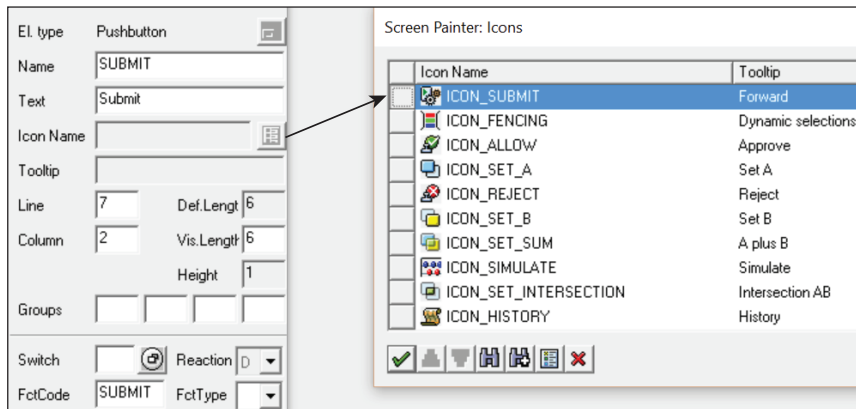


Figure 12.23 Assigning Icon

The function code can be accessed in the program using the `sy-ucomm` system field. It is also transferred to the `ok_code` field if the `ok_code` field is assigned in the screen element list. The screen element list consists of a field of type `OK`, as shown in [Figure 12.24](#). Once the `ok_code` field is assigned in the element list, it should be declared in the program as type `sy-ucomm`.

Some user actions may trigger PAI but not pass any function code (e.g., if the user presses the `[Enter]` key on the keyboard). In such cases, the `sy-ucomm` field or the `ok_code` field retains the function code of the previous action. To ensure the previous actions of the user are not performed when the function code isn't sent, it's good practice to clear the `ok_code` field after performing the related task for the user action.

Screen number: 100 Active(Revised)

Attribu... Element l... Flow logic

General a... Texts/ I/O templates Special attr. Display attr. Mod. groups

H..	M	Name	Typ...	Line	C...	D...	Vi...	H...	S...	Format
		MAKT-MATNR	Text	3	3	20	18	1		
		MAKT-MATNR	I/O	3	24	18	18	1	<input type="checkbox"/>	CHAR
		MAKT-SPRAS	Text	4	3	15	15	1		
		MAKT-SPRAS	I/O	4	19	2	2	1	<input type="checkbox"/>	LANG
		MAKT-MAKTX	Text	5	3	15	15	1		
		MAKT-MAKTX	I/O	5	19	40	40	1	<input type="checkbox"/>	CHAR
		OK CODE	OK	0	0	20	20	1	<input type="checkbox"/>	OK

Figure 12.24 Assigning `ok_code` Field in Element List

This section discussed the steps for creating the most basic screen elements of a dialog program. In the next section, we'll look at one of the more specialized elements: I/O fields.

12.4 Input/Output Fields

As previously discussed, I/O fields are used to enter or display data. To create an I/O field, you can drag and drop the I/O field from the element palette on to the work area, similarly to text fields, as shown in [Figure 12.25](#).

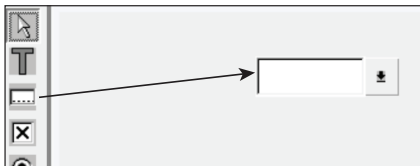


Figure 12.25 I/O Field

However, if the screen element has corresponding fields in your program or in ABAP Data Dictionary, you can perform the following steps to add an I/O field:

1. Select the DICTIONARY/PROGRAM FIELDS icon from the toolbar, as shown in [Figure 12.26](#).
2. Enter the table name or field name in the TABLE/FIELD NAME field and click GET FROM DICTIONARY or GET FROM PROGRAM, depending on from where you want to pick the field from.
3. Select the field and click CONTINUE.
4. Once you move your cursor in the work area, you'll see the cursor with the block of fields selected. Click the position in the work area where you want to display the screen element(s).

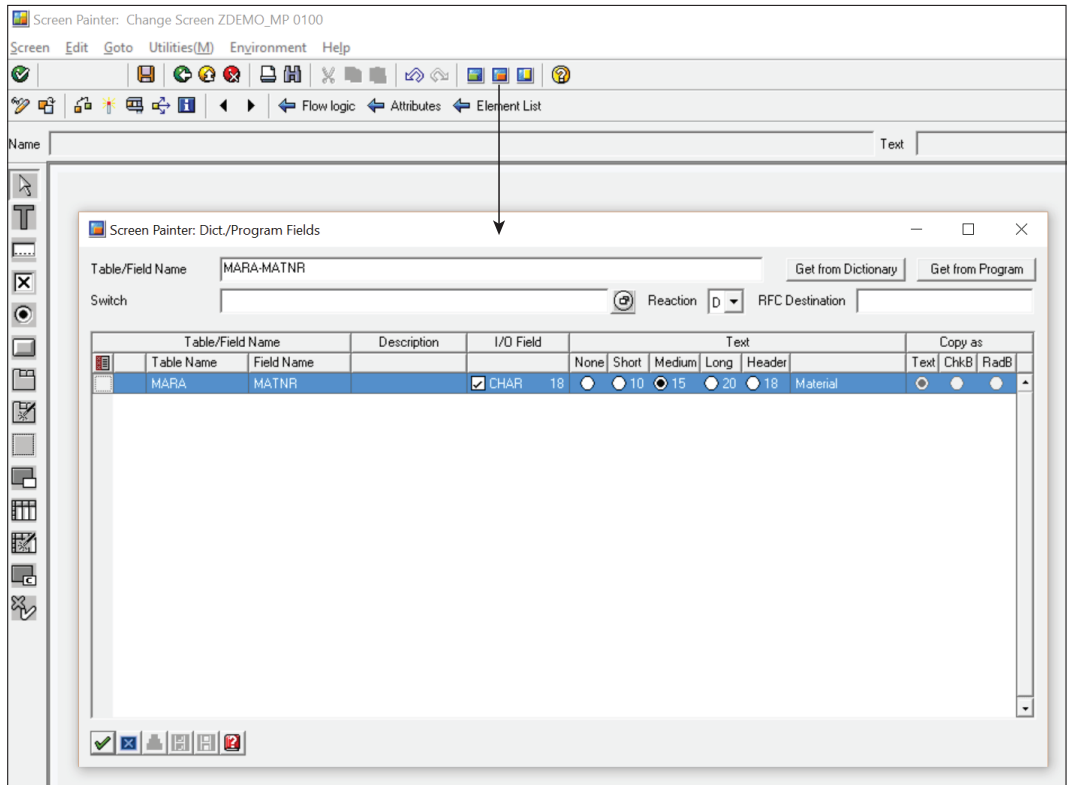


Figure 12.26 Using the Dictionary/Program Fields Button to Add I/O Elements

If you inserted the fields from ABAP Data Dictionary, then the ABAP Data Dictionary table should be declared in the program to facilitate data transfer between the program and the screen.

The screen field properties, such as conversion exit, foreign key check, and so on, are automatically inherited from the table field if they're maintained for the field in ABAP Data Dictionary.

12.5 List Box

List boxes are special I/O fields that display possible entries in a dropdown list. These fields are useful if you want the user to select only from available options. The user can't enter a value in the list box directly. If a function code is assigned to the list box, it will trigger PAI the moment a value is selected from the list box.

A list box consists of two columns: key and value. You can define the list box to show both key and value or just the value. When the user selects a value in the list box, the corresponding program data object will be passed the key of the value. This is useful when you want to show the external format of the data to the user but use the internal format internally.

To set the I/O field as a list box, double-click the screen element and select LISTBOX or LISTBOX WITH KEY in the DROPDOWN field in the field attributes, as shown in [Figure 12.27](#). You can populate the values in the list box either from ABAP Data Dictionary (from the search help or value range) or from the flow logic using the POV event. This is the recommended method, though you can also populate the list box by setting the FROM PROGRAM attribute for the value list (see [Figure 12.27](#)). In such cases, the list box is populated by passing an internal table to the `VRM_SET_VALUES` function module called in a PBO module.

Element Attributes

Each screen element can be set with additional attributes by double-clicking it. The following URL provides comprehensive documentation for various attributes that can be set for different screen elements:

https://help.sap.com/saphelp_nw74/helpdata/en/d1/801c2d454211d189710000e8322d00/content.htm.

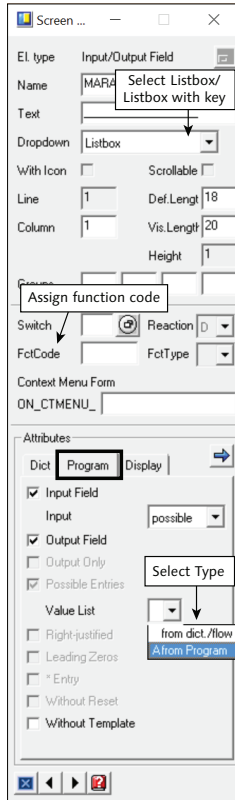


Figure 12.27 List Box

12.6 Table Controls

Table controls are used to display data in a tabular format. You can use a table control when you want to display multiple rows of data (such as order items). [Figure 12.28](#) shows the various elements of a table control.

Item	Description	Billed Quantity	SU	Net value
10	Sony Sunny01	5	ST	8,150.00
20	Sony Xa1	5	ST	9,440.00
30	Highscreen MS 1775P	4	ST	9,400.00
40	MAG DX 15F/Fe	4	ST	5,848.00

Figure 12.28 Table Control

Table controls can be created on a screen either directly or by using a wizard. In the following subsections, we'll look at each of these options.

12.6.1 Create a Table Control without a Wizard

This section walks through the steps for creating a table control without the use of a wizard.

Proceed as follows:

1. Select the table control element from the element palette and drag it to the work area, as shown in [Figure 12.29](#). Provide a name for the table control.

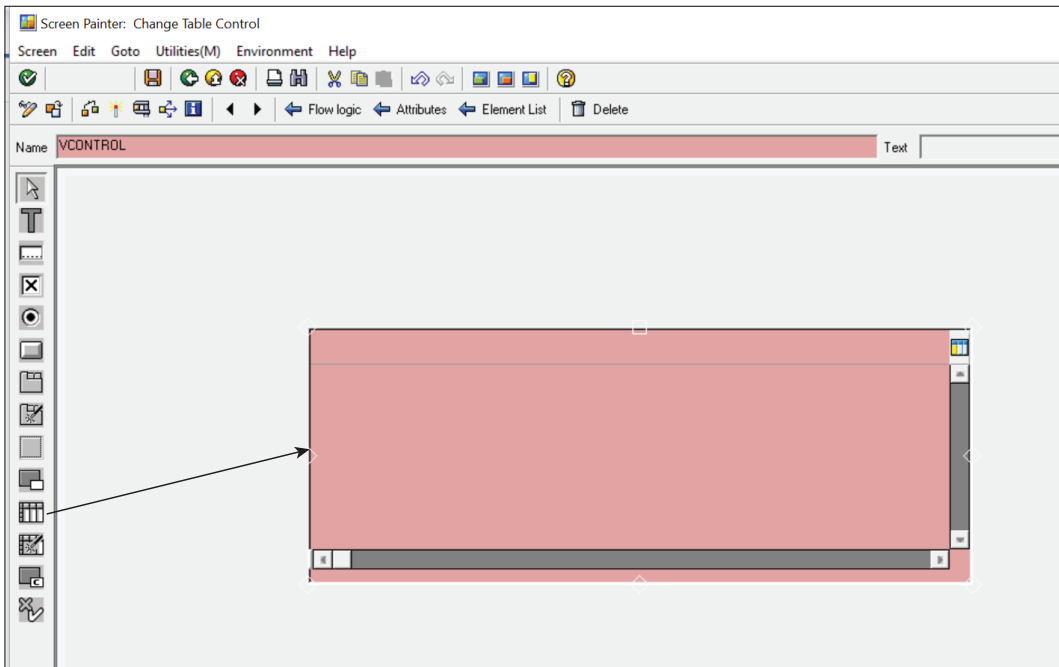


Figure 12.29 Creating Table Control

2. Click the **DICTIONARY/PROGRAM FIELDS** icon on the toolbar (see [Figure 12.26](#)) to select the fields for the table control. Enter the ABAP Data Dictionary table name or internal table name and select either the **GET FROM DICTIONARY** button or the **GET FROM PROGRAM** button, respectively. Select the fields that should be added to the table control and click **CONTINUE**.

3. Move the cursor over the table control and click to add the columns. You can repeat this process to select additional columns from different tables. At this point, the table control should look like [Figure 12.30](#).

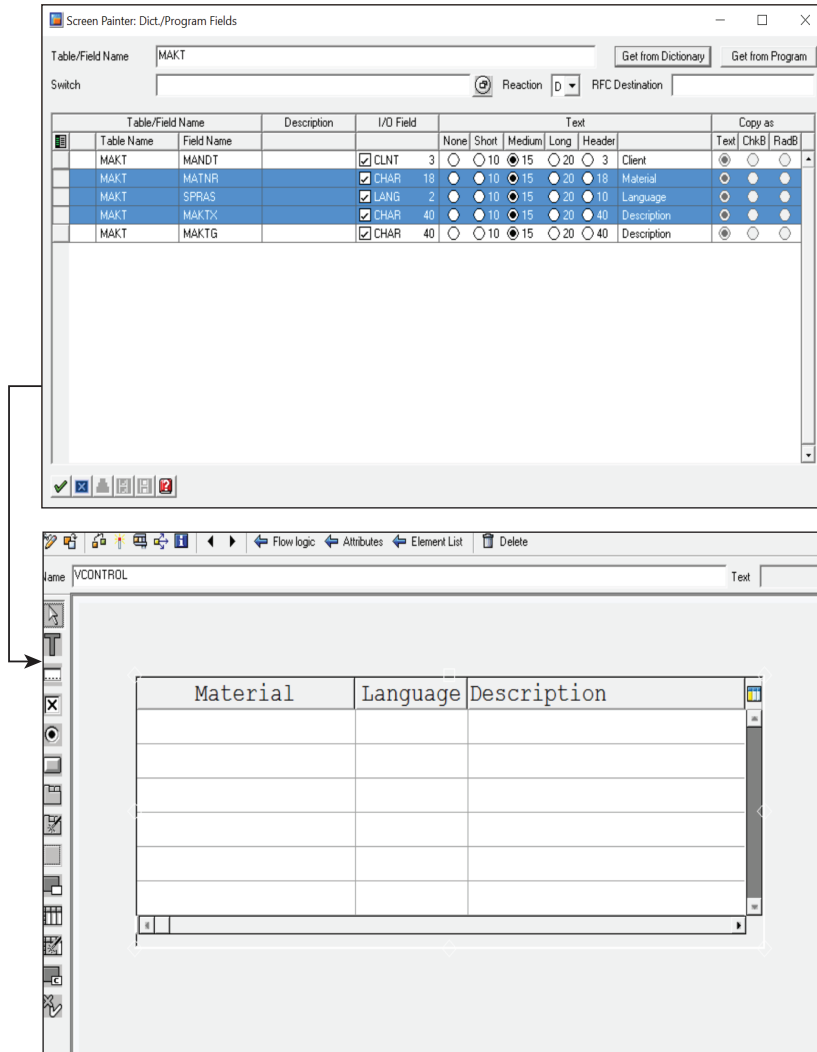


Figure 12.30 Adding Table Fields

4. You can maintain additional attributes, such as separators, rows, and column selection modes, as shown in [Figure 12.31](#).

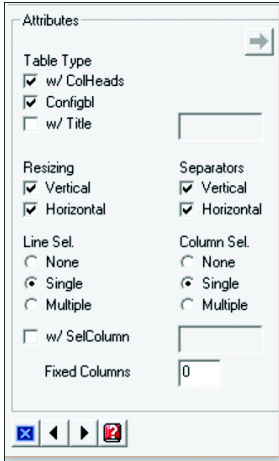


Figure 12.31 Table Control Attributes

Table 12.1 describes the attributes shown in Figure 12.31.

Attribute	Description
W/ COLHEADS	Allows you to create column headings with text fields or I/O fields.
W/ TITLE	Allows you to create a table tile with text fields or I/O fields.
CONFIGBL	Allows the table control settings (such as column width) to be saved to a file at runtime. The system will read this file whenever the table is displayed.
RESIZING	Allows the table to be resized vertically and horizontally.
SEPARATORS	Adds vertical separators between columns and horizontal separators between rows.
LINE SEL.	Allows the table row to be selected. You can set it to select a single row, multiple rows, or none.
COLUMN SEL.	Allows the table column to be selected. You can set it to select a single row, multiple rows, or none.
W/ SELCOLUMN	Specifies if the line selection column should appear in the table. If this option is selected, you should provide a name for the column, and the column will be stored internally as a checkbox.
FIXED COLUMNS	Allows you to display a fixed number of columns by excluding one or more columns from horizontal scrolling. Enter the number of columns you want to fix (the count starts from the left and moves to the right).

Table 12.1 Attribute Meanings

5. The table control displays multiple rows of data, but only one row of data is transported between the program and screen at once. To transport the data between the table control and the program's internal table, a loop statement should be maintained in the flow logic of the screen. The loop statement is maintained in both PBO and PAI. In a PBO loop, the records are transported from the program to the screen, and in a PAI loop the records are transported from the screen to the program, allowing you to capture user inputs.

In PBO, the loop statement is specified using the following syntax:

```
LOOP AT itab INTO wa WITH CONTROL tbl_ctrl.
  ENDLLOOP.
```

Here, `itab` is the internal table name, `wa` is the work area, and `tbl_ctrl` is the table control name. You can't call any modules within this loop. The name of the work area should match the structure name used in table control. If the table control columns are inserted from an ABAP Data Dictionary table, then the same should be defined using the `TABLES` statement in the ABAP program and used as the work area in the loop.

In PAI, the syntax for a loop statement is as follows:

```
LOOP AT itab.
  MODULE mod.
  ENDLLOOP.
```

In a PAI loop, modules can be called to process user inputs and update the internal table. If the screen consists of more than one table control, then the loop statement should be maintained for each table control.

In this example, we've named our table control `VCONTROL` and inserted the fields from table `MAKT`. [Figure 12.32](#) shows the flow logic code maintained for the table control.

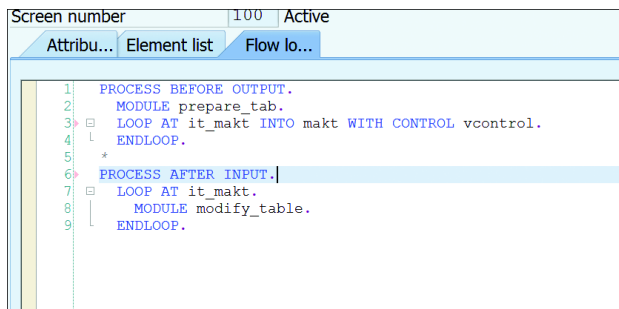


Figure 12.32 Flow Logic for Table Control

6. The table control should be defined in the ABAP program using the `CONTROLS` statement as shown:

```
CONTROLS tbl_ctrl TYPE TABLEVIEW USING SCREEN dynnr.
```

Here, `tbl_ctrl` is the name of the table control, and `dynnr` is the screen number of the screen for which the table control is defined.

The `CONTROLS` statement defines the table control `tbl_ctrl` as a structure of type `CXTAB_CONTROL`. We can control the table control using the components of structure `CXTAB_CONTROL`. For example, the `CURRENT_LINE` component contains the current row being processed.

[Listing 12.2](#) shows the code in the main program for this example.

```
TABLES makt.
DATA : it_makt TYPE STANDARD TABLE OF makt.
CONTROLS vcontrol TYPE TABLEVIEW USING SCREEN 100.

MODULE prepare_tab OUTPUT.
  IF it_makt IS INITIAL.
    SELECT * FROM makt INTO TABLE it_makt UP TO 10 ROWS.
  ENDIF.
ENDMODULE.

MODULE modify_table INPUT.
MODIFY it_makt FROM makt INDEX vcontrol-current_line.
ENDMODULE.
```

Listing 12.2 Main Program Code for Table Control

We've looked at the steps for creating a table control without the use of a wizard, so now we'll look at how the wizard provided by SAP can speed up this process.

12.6.2 Create a Table Control with a Wizard

SAP provides a wizard to quickly create the table control; with the wizard, all the required steps and code is autogenerated.

Follow these steps to create the table control with the wizard:

1. In the graphical layout editor, select `TABLE CONTROL WITH WIZARD` and drag it onto the work area to start the wizard, as shown in [Figure 12.33](#).

The wizard will walk you through multiple steps, as shown in [Figure 12.34](#). The wizard provides a detailed explanation for each step; before you begin, read the explanation for each step and then click `CONTINUE`.

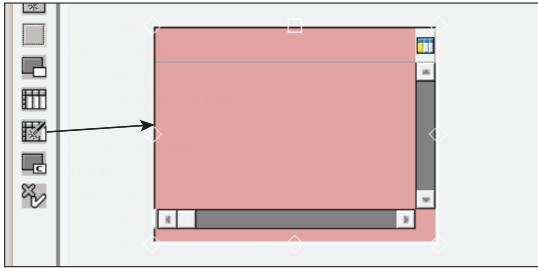


Figure 12.33 Table Control with Wizard in Work Area

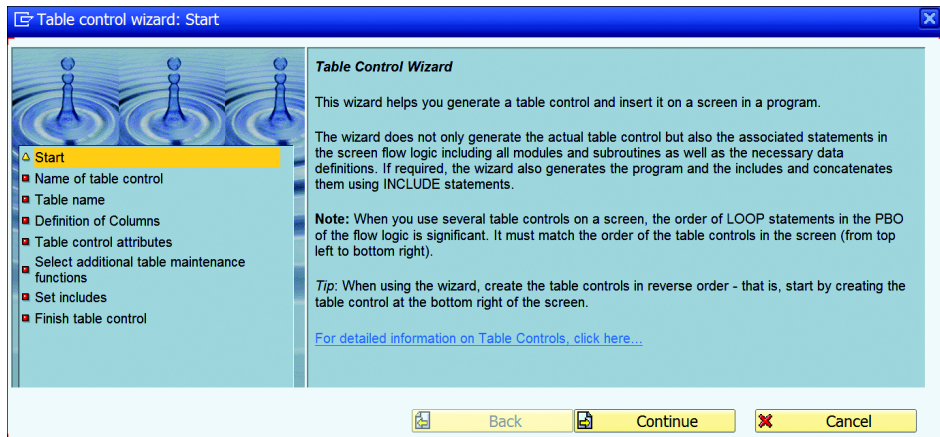


Figure 12.34 Table Control Wizard: Start

2. In the next step, provide the name of the table control. If you have multiple table controls on the screen, each table control should have a unique name. Click CONTINUE.
3. On the next screen, provide the name of the table, either from ABAP Data Dictionary or an internal table name from the program. If the internal table is without a header line, provide the name of the work area as well. Click CONTINUE.
4. Select the fields of the table control. Click CONTINUE.
5. Set the preferred attributes for the table control. Click CONTINUE.
6. You can set additional table maintenance functions, such as scroll. Click CONTINUE.

7. Provide the include names where the code should be generated. You can enter the program name to generate all the code directly in the main program. To maintain legibility, we recommend creating separate include programs for PBO modules, PAI modules, and subroutines. Click **CONTINUE**.
8. Click **COMPLETE** to complete the wizard and generate the table control and the associated code. Activate the screen and the program.

12.7 Tabstrip Controls

Tabstrip controls allow you to create multiple tabs on a screen. If the screen has many UI elements, you can segregate them into multiple tabs for a better user experience.

A tabstrip control area is marked on the main screen, and each tab is linked to a subscreen. When a tab is selected, the respective subscreen is loaded with all the elements maintained on the subscreen. The elements of the tab are not maintained on the main screen where the tabstrip control is created, but are instead maintained on the subscreen to which the tab is linked. You need to have a minimum of two tabs to create a tabstrip control.

The following steps walk you through creating a tabstrip control with the wizard:

1. From the element palette, select **TABSTRIP CONTROL WITH WIZARD** and drag it onto the work area where you want to have the tabstrip control (see [Figure 12.35](#)).

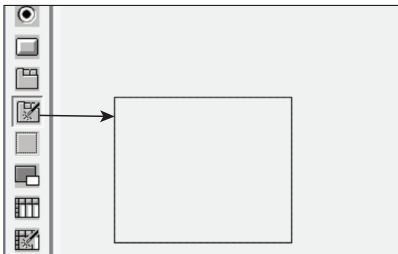


Figure 12.35 Tabstrip Control with Wizard

This will start the wizard, as shown in [Figure 12.36](#). Read the notes for the step and click **CONTINUE**.

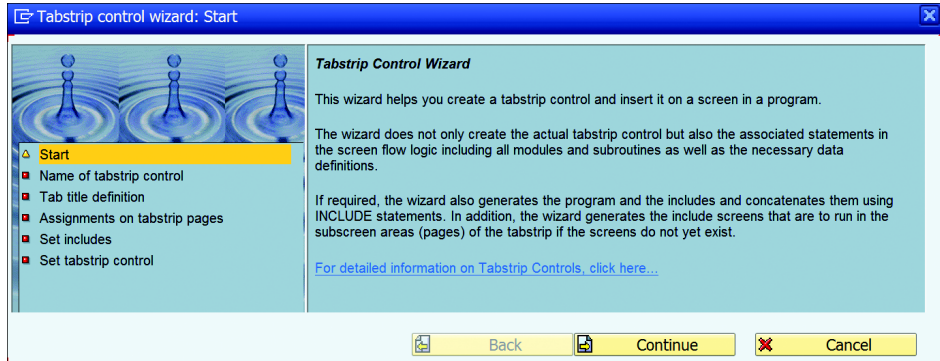


Figure 12.36 Tabstrip Control Wizard

2. In the next step, provide a name for the tabstrip control. You can also set the scrolling behavior. The documentation for the steps explains the significance of each option. Click CONTINUE.
3. Next, maintain the tab titles. You must define at least two tabs. This title will be displayed for the tab. Click CONTINUE.
4. The system will automatically propose the screen number for the subscreens that should be generated. The number of subscreens generated depends on the number of tabs defined in the previous step. A function code for each tab is also proposed. You can accept the defaults or modify them. Click CONTINUE.
5. Provide the include names where the code should be generated. Click CONTINUE.
6. The wizard will display a summary of all the actions that will be performed. Click COMPLETE to generate the screens and the corresponding code.
7. Activate the screen and the program, along with the newly generated subscreens. To maintain screen elements, open the subscreen corresponding to each tab and create elements using the graphical layout editor.

12.8 Subscreens

Subscreens are independent screens that are always embedded on another screen. Subscreens are created in Screen Painter, similarly to normal screens, but with the subscreen attribute selected, as shown in [Figure 12.37](#).

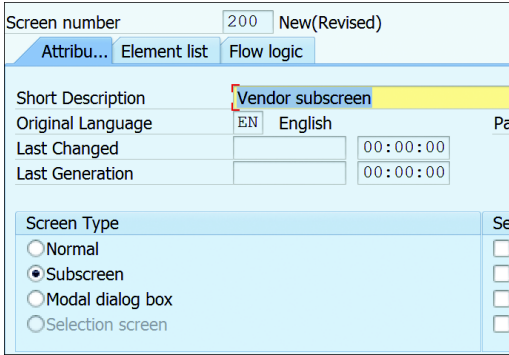


Figure 12.37 Subscreen

A subscreen can't have its own GUI status, and the flow logic of the subscreen will be embedded into the flow logic of the main screen. A subscreen can embed another subscreen.

To embed a subscreen, a subscreen area is marked on the main screen (see [Figure 12.38](#)), where the subscreen will be loaded at runtime using the `CALL SUBSCREEN` statement.



Figure 12.38 Subscreen Area

One main screen can have many subscreen areas. Each subscreen area of the main screen should be assigned a unique name ([Figure 12.38](#)).

To create a subscreen area on the main screen, select the subscreen area from the element palette and drag it to the work area where you want the subscreen to load. Provide a unique name for the subscreen area. To load the subscreen in the subscreen area, under PBO event, use the `CALL SUBSCREEN` statement in the screen flow logic of the main screen where the subscreen area is defined.

The syntax is as follows:

```
CALL SUBSCREEN sub INCLUDING prog dynnr.
```

Here, `sub` is the subscreen area name, `prog` is the program name to which the subscreen belongs, and `dynnr` is the subscreen number.

To access the contents of the screen elements of the subscreen in the main screen flow logic, use the `CALL SUBSCREEN` statement in PAI of the main screen:

```
CALL SUBSCREEN sub.
```

All the screen elements that you can define on a normal screen can be defined on a subscreen. You can't define an `OK_CODE` field for a subscreen, because the subscreen uses the main screen's GUI status. Also, you can't set a screen title for the subscreen.

Now that we've reviewed the various screens elements, in the next section we'll discuss how to work with screens.

12.9 Working with Screens

Thus far, you've learned how to create various screen elements on the screen. In this section, we'll shift our focus to the screen flow logic. We'll discuss the various statements that can be used in screen flow logic to call dialog modules, how to maintain and call the GUI status for the screen, and how to set the title for the screen. We'll also look at how screen fields can be modified dynamically and how field help and input help can be set for the screen fields.

A single dialog program can have multiple screens that can be called in sequence. We'll look at how different screens can be called in sequence, and then we'll conclude the section by walking through the steps to assign a transaction code to the program.

12.9.1 Screen Flow Logic

Let's look at the statements that can be used in the screen flow logic and how to call dialog modules. *Screen flow logic* allows you to use a limited set of ABAP statements, and it's divided into different sections of screen events. The screen events maintained in the flow logic are as follows:

- ▶ PROCESS BEFORE OUTPUT
- ▶ PROCESS AFTER INPUT
- ▶ PROCESS ON HELP-REQUEST
- ▶ PROCESS ON VALUE-REQUEST

The screen flow logic must contain the PROCESS BEFORE OUTPUT and PROCESS AFTER INPUT events at a minimum; the PROCESS ON HELP-REQUEST and PROCESS ON VALUE-REQUEST events are optional. You only need to maintain the POH and POV events if you want to handle the `[F1]` help and `[F4]` help for the screen fields.

Within each event block, you can call dialog modules. Some of the statements that you can use within the event block of the flow logic include the following:

- ▶ **MODULE**
This statement calls the dialog module maintained in the main program. This statement can be used in all screen events.
- ▶ **FIELD**
This statement controls the data transport from the screen field to the ABAP program. If this statement isn't used, the data of the field is automatically transferred to the program at the beginning of the PAI event. When the FIELD statement is used, the corresponding screen field value is transferred when the system executes this statement. The FIELD statement is used in conjunction with the MODULE statement to restrict the data transfer until a particular module is executed.

For example, in [Listing 12.3](#), the data of the screen field `v_matnr` isn't transferred until the system executes the `FIELD v_matnr MODULE get_data` statement. This effectively restricts the data transfer of the screen field to all the modules that are called before the FIELD statement. In our example, the data of the `v_matnr` field will not be available in the `user_command` module that is called before the `get_data` module. The FIELD statement is also used to validate input fields.

When an error message is shown within a module that is called with the `FIELD` statement, the corresponding field will be input-enabled for the user to correct the input; otherwise, the error message disables all the input fields on the screen and the user won't be able to correct wrong input. Always handle an error message inside a module that's called with the `FIELD` statement.

```
PROCESS AFTER INPUT.
MODULE user_command.
FIELD v_matnr MODULE get_data.
```

Listing 12.3 `FIELD` Statement

The `FIELD` statement supports `ON INPUT` and `ON REQUEST` (e.g., `FIELD v_matnr MODULE get_data ON INPUT`). `ON INPUT` restricts the module call if the corresponding field is empty, whereas `ON REQUEST` restricts the module call if the value in the corresponding field hasn't changed compared to the last PBO event. If none of these additions are used, the module is always called in each PAI event. These additions are useful when you're performing steps inside a module that need not be repeated if the field value is empty or if the field value hasn't changed (e.g., when querying the database table to validate the input).

`ON INPUT` and `ON REQUEST` can be used only in PAI events.

► **CHAIN**

The purpose of the `CHAIN` statement is similar to the `FIELD` statement, except that the `FIELD` statement is used for a single field and the `CHAIN` statement is used for multiple fields. For example, if you want to restrict the data transfer of multiple related fields, use the `CHAIN` statement as shown in [Listing 12.4](#). The `CHAIN` statement is concluded with `ENDCHAIN`.

```
PROCESS AFTER INPUT.
MODULE user_command.
CHAIN.
FIELD v_matnr.
FIELD v_spras.
MODULE get_data.
ENDCHAIN.
```

Listing 12.4 `CHAIN` Statement

The corresponding `ON INPUT` and `ON REQUEST` options for the `CHAIN` statement are `ON CHAIN-INPUT` and `ON CHAIN-REQUEST` (e.g., `MODULE get_data ON CHAIN-INPUT`). The `ON CHAIN-INPUT` and `ON CHAIN-REQUEST` additions can be used only in PAI events.

► **CALL**

This statement is used in the flow logic to call a subscreen. This statement can't be used with `LOOP...ENDLOOP` or `CHAIN...ENDCHAIN` statements.

► **LOOP**

This statement is used to loop the table control. For example, the `LOOP WITH CONTROL` statement is used to loop the table control.

Within the dialog module maintained in the ABAP program, you can use all valid ABAP statements.

12.9.2 GUI Status

Each screen can load its own GUI status. The GUI status is set in the PBO module of the screen using the `SET PF-STAUS` statement. The GUI status is defined using Menu Painter.

Follow these steps to define a GUI status:

1. Right-click the program name in Object Navigator and select **CREATE • GUI STATUS**.
2. In the **CREATE STATUS** window, as shown in [Figure 12.39](#), enter the status name and a description. For **STATUS TYPE**, select **NORMAL SCREEN**. If you're creating the GUI status for a model dialog box, you should select the **DIALOG BOX** option. If you're creating a context menu (displayed on right-click), you should select the **CONTEXT MENU** option. Click **CONTINUE**.

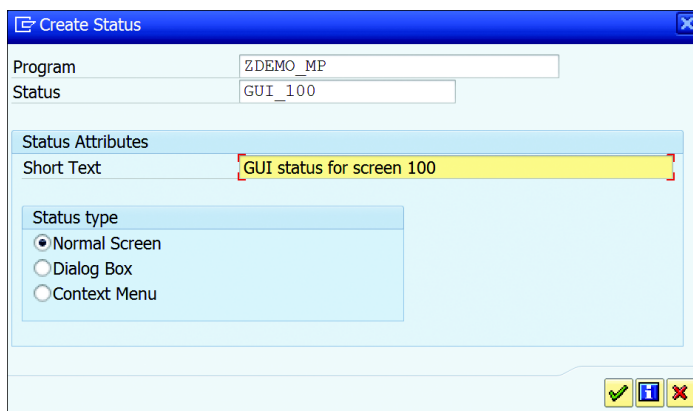


Figure 12.39 Create Status

- On the MAINTAIN STATUS screen, you can define the status for the menu bar, application toolbar, standard toolbar, and function keys. Expand each status to maintain the function code, as shown in [Figure 12.40](#). You can define new objects for the menu bar and application toolbar, but for the standard toolbar you can only enable or disable the standard buttons by maintaining a function code for each button.

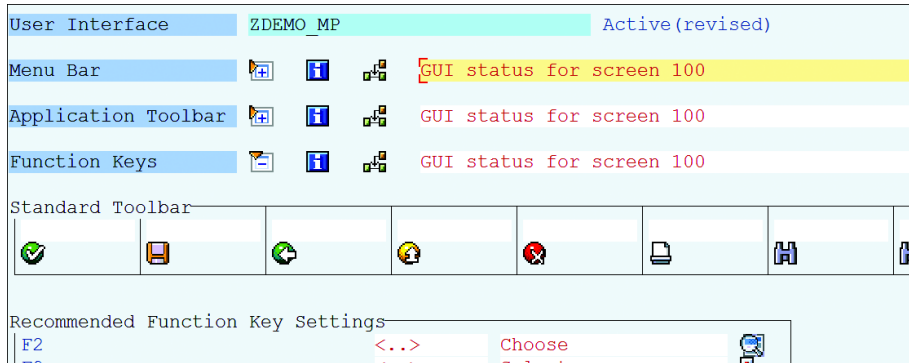


Figure 12.40 Maintaining Status

- To create an application toolbar button, expand the APPLICATION TOOLBAR status (see [Figure 12.41](#)). Provide a function code in the available item positions and press `[Enter]`.

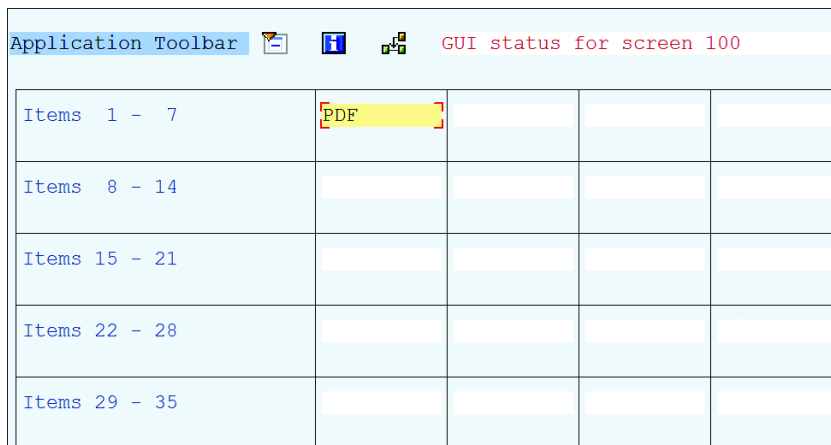


Figure 12.41 Application Toolbar

- In the ENTER FUNCTION TEXT popup window that appears, select STATIC TEXT or DYNAMIC TEXT and click CONTINUE. If you select DYNAMIC TEXT, the system will prompt you to provide a data object or dictionary field that will contain the text. If you select STATIC TEXT, it will show a window in which you can maintain the details manually (see [Figure 12.42](#)). Click CONTINUE to set a function key for the status.

Function code	PDF
Function text	Download PDF
Icon name	ICON_PDF
Info. text	Download as PDF

Figure 12.42 Maintaining Static Text for Status Button

- Once all the GUI statuses are defined, activate them.
- To load the GUI status for a screen, use the `SET PF-STATUS <'status_name'>` statement in the PBO module of the screen as shown:

```
MODULE status_100 OUTPUT.
SET PF-STATUS 'GUI100'.
ENDMODULE.
```

- If you want to dynamically hide the status button on the screen, you can use the `EXCLUDING` addition with the `SET PF-STATUS` statement:

```
SET PF-STATUS 'GUI100' EXCLUDING fcode.
```

Here, `fcode` is a function code that should be excluded from the screen. You can also supply an internal table with a single field of type `sy-ucomm`. All the function codes maintained in this internal table will be excluded on the screen.

12.9.3 GUI Title

You can set a title for the screen to be displayed in the title bar. The title is loaded in a PBO module using the `SET TITLEBAR title_code` statement.

To create a title, proceed with the following steps:

1. Right-click the program name in Object Navigator and select CREATE • GUI TITLES.
2. The CREATE TITLE window will open (see [Figure 12.43](#)). Maintain a title code and the TITLE text. Click CONTINUE.

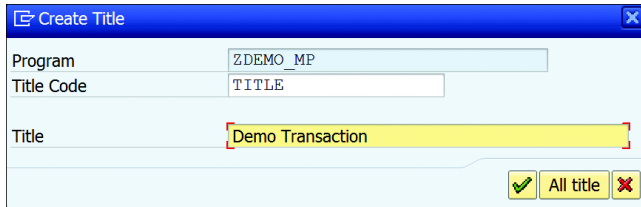


Figure 12.43 Create Title

3. Once the title is created and activated, you can load the title as shown in [Listing 12.5](#).

```
MODULE status_100 OUTPUT.
SET TITLEBAR 'TITLE'.
ENDMODULE.
```

Listing 12.5 Set Title

12.9.4 Modifying Screen Fields Dynamically

The attributes of screen fields are set statically at design time. However, it's possible to modify some of these attributes at runtime. The screen field attributes can be accessed using a predefined `SCREEN` structure. We can access each screen element by looping the screen structure as shown:

```
LOOP AT SCREEN.
MODIFY SCREEN,
ENDLOOP.
```

The screen structure (see [Figure 12.44](#)) has various attributes that can be modified at runtime in the ABAP program. The `name` component holds the names of the screen fields that can be checked to apply the changes to specific screen fields. If the screen fields are assigned to a group, the group name can be accessed by checking the `group1`, `group2`, `group3`, and `group4` fields. By grouping the screen fields, we can apply the changes consistently to all fields of the group.

Dictionary: Display Structure

← → ↻ ↺ ↻ ↺ ↻ ↺ ↻ ↺ ↻ ↺ Hierarchy Display

Structure: SCREEN Active

Short Description: Structure Description for the System Data Object SCREEN

Attribu... Compon... Entry help/check Currency/quantity fields

Predefined Type 1 / 17

Component	Typing Method	Component Type	Data Type	Length	Deci...	Short Descript
NAME	Types		CHAR	132	0	
GROUP1	Types		CHAR	3	0	
GROUP2	Types		CHAR	3	0	
GROUP3	Types		CHAR	3	0	
GROUP4	Types		CHAR	3	0	
REQUIRED	Types		CHAR	1	0	
INPUT	Types		CHAR	1	0	
OUTPUT	Types		CHAR	1	0	
INTENSIFIED	Types		CHAR	1	0	
INVISIBLE	Types		CHAR	1	0	
LENGTH	Types		RAW	1	0	
ACTIVE	Types		CHAR	1	0	
DISPLAY 3D	Types		CHAR	1	0	
VALUE HELP	Types		CHAR	1	0	
REQUEST	Types		CHAR	1	0	
VALUES IN COMBO	Types		CHAR	1	0	
COLOR	Types		INT4	10	0	

Figure 12.44 Screen Structure

For example, if we want to dynamically set a screen field as a required field, then we can use the SCREEN structure, as shown in [Listing 12.6](#). The screen field attributes should be modified in a PBO module just before showing the screen.

```
DATA wa_screen TYPE screen.
MODULE modify_fields OUTPUT.
LOOP AT SCREEN INTO WA_SCREEN.
IF WA_SCREEN-NAME = 'V_MATNR'.
WA_SCREEN-REQUIRED = abap_true.
MODIFY SCREEN FROM WA_SCREEN.
ENDIF.
ENDLOOP.
ENDMODULE.
```

Listing 12.6 Modifying Screen Field Attribute

[Table 12.2](#) explains the components of the SCREEN structure.

Field Name	Possible Values	Meaning
NAME	Name of screen element	Name of the screen field
GROUP1 to GROUP4	id1 to id4	Modification group to apply common settings to related fields that belong to the same group
REQUIRED	0, 1, 2	Mandatory or recommended field
INPUT	0, 1	Input-ready field
OUTPUT	0, 1	Read-only/display field
INTENSIFIED	0, 1	Highlighted field
INVISIBLE	0, 1	Invisible element
LENGTH	Visible length of field	Field length
ACTIVE	0, 1	Active field
DISPLAY_3D	0, 1	Display frame
VALUE_HELP	0, 1, 2	Input help button
REQUEST	0, 1	Input exists
VALUES_IN_COMBO	0, 1	Dropdown list box

Table 12.2 Components of Screen Structure

The cursor position can be set statically in the screen attributes. To dynamically set the cursor position, you can use the statement `SET CURSOR`, as shown:

```
SET CURSOR FIELD field_name.
```

Here, `field_name` is the name of the screen field. The `SET HOLD DATA ON|OFF` statement can be used to dynamically set the `HOLD DATA` flag for the screen.

12.9.5 Field Help and Input Help

The *field help* `[F1]` and *input help* `[F4]` for the screen field can be shown directly if they're maintained for the ABAP Data Dictionary table field and the screen field is inserted by selecting from the ABAP Data Dictionary table field (for example, data element documentation for `[F1]` help or search help for `[F4]` help). However, you can also handle this in the program by maintaining the `POH` event for field help and the `POV` event for input help.

In the following subsections, we'll look at how to handle the field help and input help settings.

Field Help

To handle the field help, the event `PROCESS ON HELP-REQUEST` should be maintained in the screen flow logic. Within this event, a module can be called for the field using the statement `FIELD field_name MODULE mod_name`. Here, `field_name` is the name of the field for which the field help should be called, and `mod_name` is the dialog module called in the program—for example:

```
PROCESS ON HELP-REQUEST.
FIELD MATNR MODULE F1_MATNR.
```

Inside the module called for field help, you can call the `HELP_OBJECT_SHOW_FOR_FIELD` or `HELP_OBJECT_SHOW` function modules to load the field help. The function module `HELP_OBJECT_SHOW_FOR_FIELD` displays the data element documentation maintained for any table or structure component of ABAP Data Dictionary. Using the function module `HELP_OBJECT_SHOW`, you can display an SAPscripts document.

Input Help

To handle input help in the program, you need to use the `PROCESS ON VALUE-REQUEST` event and call the dialog module using the `FIELD` statement, similarly to the `POH` event—for example:

```
PROCESS ON VALUE-REQUEST.
FIELD MATNR MODULE F4_MATNR.
```

Within the module called for the field, you can call the `F4IF_FIELD_VALUE_REQUEST` or `F4IF_INT_TABLE_VALUE_REQUEST` function module to pass the value list.

12.9.6 Screen Sequence

Dialog transactions often involve calling multiple screens in sequence. When a transaction is called, it calls the initial screen of the program. The next screen attribute of the screen can be set statically to call the next screen after the current screen is processed (i.e., after the current screen's `PAI`). Often, the next screen called in the screen sequence depends on the user actions on the current screen.

The screen sequence can be dynamically handled by using statements such as `SET SCREEN`, `CALL SCREEN`, and `LEAVE SCREEN`.

In the following subsections, we discuss each of these screen sequence statements.

SET SCREEN

The `SET SCREEN` statement overrides the next screen attribute set for the screen at design time. When you call the next screen in the screen sequence using the `SET SCREEN` statement, it will call the next screen after executing all the modules in the current screen's PAI.

CALL SCREEN

The `CALL SCREEN` statement stacks the current screen and calls the new screen immediately without executing further code in the current screen. When the screen sequence ends, the control will return to the statement immediately after the `CALL SCREEN` statement. For example, in screen 100, if you use the `CALL SCREEN 110` statement, it will call screen 110 immediately. If the next screen for screen 110 is set as screen 120, then after the PAI of screen 110, the PBO of screen 120 will be called.

If the statement `LEAVE TO SCREEN 0` is used in screen 120, then the system will end the screen sequence and return to screen 100 and execute the statement immediately after the `CALL SCREEN` statement, which started the screen sequence. The `CALL SCREEN` statement interrupts the current screen sequence and starts a new screen sequence. The maximum number of screens that can be stacked is fifty, but it's not recommended to stack more than forty screens in a sequence because the system uses additional screen sequences internally—for example, to show messages.

LEAVE SCREEN

The `LEAVE SCREEN` statement leaves the current screen immediately and starts the next screen. The next screen could be a static next screen set in the screen attributes or a dynamic next screen set using the `SET SCREEN` statement. The `LEAVE TO SCREEN dynnr` statement leaves the current screen and goes to the one specified by `dynnr`. With the `LEAVE SCREEN` statement, the screens aren't stacked, because this

statement doesn't start a new screen sequence; instead, it calls the next screen in the current sequence immediately. The screen sequence ends when you leave to screen 0.

Note

A modal dialog box (pop-up screen) is called using the syntax `CALL SCREEN STARTING AT XX YY ENDING AT XX YY`. Here, `XX` and `YY` are two-digit numbers specifying the coordinates used for the position on screen.

12.9.7 Assigning Transaction Codes

As we discussed earlier, module pool programs can't be executed directly by program name. To execute a module pool program, we need to create a transaction code for the program, to which the initial screen of the program is assigned.

To create a transaction code for the program, proceed with the following steps:

1. Right-click the program name in Object Navigator and select `CREATE • TRANSACTION`, as shown in [Figure 12.45](#).

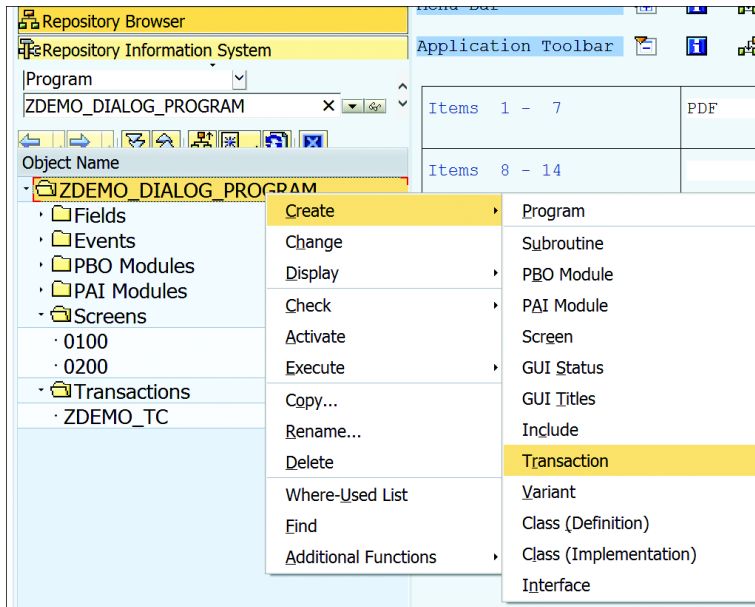


Figure 12.45 Create Transaction

2. In the CREATE TRANSACTION window (see [Figure 12.46](#)), input a TRANSACTION CODE and SHORT TEXT. Select the PROGRAM AND SCREEN (DIALOG TRANSACTION) radio button and click CONTINUE. The other options on this screen allow you to create the following types of transaction codes:
- ▶ PROGRAM AND SELECTION SCREEN (REPORT PROGRAM): This option is used if the program has a selection screen that should be called as the initial screen of the program.
 - ▶ METHOD OF A CLASS (OO TRANSACTION): A method of a class can be linked to a transaction code and can then be called directly using the transaction code.
 - ▶ TRANSACTION WITH VARIANT (VARIANT TRANSACTION): This option allows you to create a transaction code for a report program that can be started with a selection screen variant directly.
 - ▶ TRANSACTION CODE WITH PARAMETERS (PARAMETER TRANSACTION): This option allows you to create a transaction code that can support parameters to be passed. For example, you want to pass certain default values to the input fields of the initial screen of the transaction.

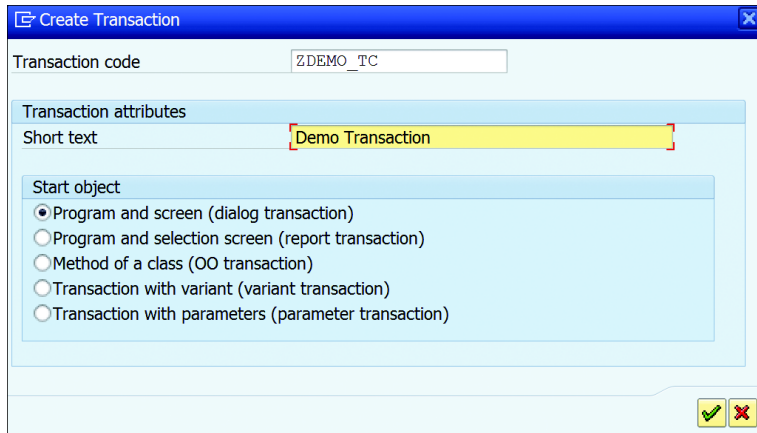


Figure 12.46 Creating Transactions

3. The transaction code should be in the customer namespace (starting with Z or Y). Choose a transaction code that's short and easy to remember, because users will always run the program using the transaction code. Click CONTINUE (green checkmark) to confirm.

- On the next screen, input the program name and the initial screen of the program that should be called when the transaction is executed. Optionally, you can assign an authorization object to restrict the access to certain users. Select the GUI SUPPORT for the transaction, as shown in [Figure 12.47](#). For example, uncheck the checkbox for a particular GUI if any of the screen elements you're using aren't supported by that GUI.

The screenshot shows a SAP transaction maintenance form with the following fields and values:

- Transaction code: ZDEMO_TC
- Package: (empty)
- Transaction text: Demo Transaction
- Program: ZDEMO_MP
- Screen number: 100
- Authorization Object: (empty)
- Values button: (yellow button with grid icon)
- Checkbox: Maintenance of standard transaction variant allowed
- Classification section:
 - Transaction classification:
 - Professional User Transaction
 - Easy Web Transaction
 - Pervasive enabled
 - Service: (empty)
- GUI support section:
 - SAPGUI for HTML
 - SAPGUI for Java
 - SAPGUI for Windows

Figure 12.47 Maintaining Transaction Attributes

- Once the transaction code is assigned to the program, you can execute the program by entering the transaction code in the command bar.

So far, we've discussed various screen elements and how they can be processed. However, these screen elements may not always cover your business requirements. To meet more complex requirements and provide a better user experience, SAP introduced the Control Framework, which we'll discuss next.

12.10 Control Framework

When we design transactions using Screen Painter, the list of available screen elements is limited to I/O fields, labels, checkboxes, and so on. This provides a

limitation on the possible UI elements that can be used on the screen. Many desktop applications that users are accustomed to may provide better UI experiences than what was traditionally possible with SAP screens. Understanding this gap, SAP introduced the Control Framework with SAP ERP 4.5.

The Control Framework allows you to create complex screens that are intuitive and fluid. It enables you to create custom controls on the user's client from your ABAP programs. This framework involves two types of components: server-side components and client-side components.

On the server side, the framework implements a series of ABAP objects, which are collectively called *ABAP Objects Control Framework*. On the client side, the framework is managed by a component called the *Automation Controller* (or *Automation Client*). The Automation Client manages all the custom control instances and takes care of the communication between the custom controls in the presentation layer and the ABAP Objects Control Framework in the application server.

In the following subsections, we'll discuss how to use container controls and then how to implement custom controls using the Control Framework.

12.10.1 Using Container Controls

Custom controls are integrated in the SAP GUI client, which implements either Active X controls or JavaBeans depending on the GUI client version. Custom controls allow you to create many complex controls, such as ALV Grid Controls, SAP calendar controls, SAP picture controls, and so on. Each custom control is exposed as an ABAP object of the corresponding control type.

A custom control runs inside a container control. Each container control is exposed as an ABAP object that acts as a parent to the custom control. Different container controls are described in [Table 12.3](#).

Container Control	Class	Description
Custom container	CL_GUI_CUSTOM_CONTAINER	The custom container is defined on the screen using Screen Painter. The custom control is displayed inside the container. One screen can have multiple containers.

Table 12.3 Container Controls

Container Control	Class	Description
Dialog box container	CL_GUI_DIALOGBOX_CONTAINER	The dialog box container allows you to display controls in a modal dialog box or full-screen.
Docking container	CL_GUI DOCKING_CONTAINER	The docking container allows you to attach a control to any of the four edges of a screen as a resizable screen area. You can also detach it so that it becomes an independent modal dialog box.
Splitter container	CL_GUI_SPLITTER_CONTAINER	The splitter container allows you to display more than one control by splitting the screen area into cells.

Table 12.3 Container Controls (Cont.)

12.10.2 Implementing Custom Controls

Implementing a custom control is straightforward. We'll use the example of the SAP picture control to explain how a control is implemented in an ABAP program. The steps should be similar for other controls with the exception of using the specific container and controller class for the chosen control.

To implement a controller, we first create a container on screen. Then, instantiate the parent container and pass the parent container reference to the custom control. From there, we can use various methods of the custom control class to work with the custom control.

Follow these steps to create a SAP picture control:

1. Create a custom container in the work area of the graphical layout editor by selecting the CUSTOM CONTROL button from the element palette and dragging it to the work area, as shown in [Figure 12.48](#). Provide a name for this container. In our example, the container name is CUSTOM.

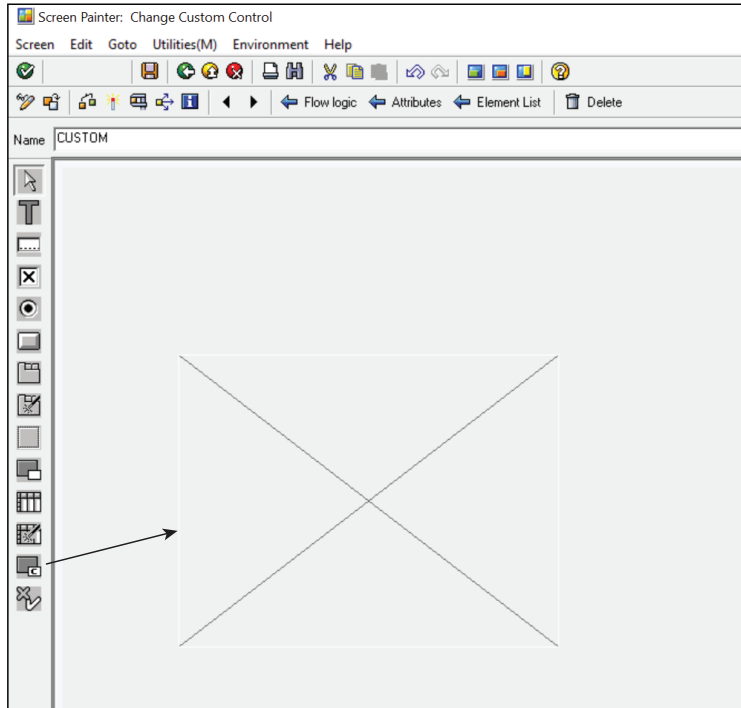


Figure 12.48 Custom Container

2. In the main program, define two reference objects for the container and the control. In our example, the reference object for the container should have a `CL_GUI_CUSTOM_CONTAINER` static type, because we're using a custom container. The reference object for the control should have a `CL_GUI_PICTURE` static type, because we're using the SAP picture controller.
3. In the PBO module of the screen, instantiate the container by passing the container name to the constructor. Once the container is instantiated, instantiate the controller by passing the container object to the constructor.
4. Once the controller is instantiated, you can call the `load_picture_from_url` method to pass the URL of the image that should be displayed in the container of the screen. [Listing 12.7](#) shows the code in the main program.

```
*Declare reference objects for container and controller
DATA : container TYPE REF TO CL_GUI_CUSTOM_CONTAINER,
      controller TYPE REF TO CL_GUI_PICTURE.
```

```

MODULE load_container OUTPUT.
*Instantiate the container
  CREATE OBJECT container
  EXPORTING
    container_name = 'CUSTOM'.
*Instantiate the controller
  CREATE OBJECT controller
  EXPORTING
    parent = container.
*Load image
  controller->load_picture_from_url(
    url = 'http://www.domain.com/image.jpg'
  ).
ENDMODULE.

```

Listing 12.7 Code to Load Image in Container

Note

Your program can react to events raised by the custom controller by implementing suitable event handler methods.

Now that we've covered dialog programming in detail, let's take a look at a practical example to practice developing these applications.

12.11 Practice

Dialog programming involves processing multiple screens and reacting to user actions on the screen. Even though the application development is fairly simple, it can become complicated easily when many screens are involved. Hands-on experience is important to feel comfortable with dialog programming.

In this section, we'll look at a practice example for dialog programming. We'll start by laying out the instructions for this example:

► Scenario

A user wants to capture materials that are loaded to a container. Design and create an application with the functionality indicated in the Functional Requirements bullet point.

► Tables

Create a Z table with the fields listed in [Table 12.4](#). You can use any field names.

Field Name	Field Type
CONTAINER NUMBER	Type C, length 6
MATERIAL NUMBER	Type C, length 18
MATERIAL DESCRIPTION	Type C, length 40
QUANTITY	Type P, decimals 2
SHIFT INDICATOR	Type C, length 1
DATE	Type D
USER NAME	Type C, length 20
DELETION INDICATOR	Type C, length 1

Table 12.4 Required Fields of Table

► **Functional requirements**

The application should provide the following functionality:

- All the fields except the field to input the container number should be input-disabled.
- Provide a DISPLAY/CHANGE toggle button in the application toolbar to toggle between display and change modes.
- Provide a table control as shown in [Figure 12.49](#). Note that the SHIFT field is a list box and the DELETE field is a checkbox.
- Enable the SAVE button in the standard toolbar to let the user save the data.

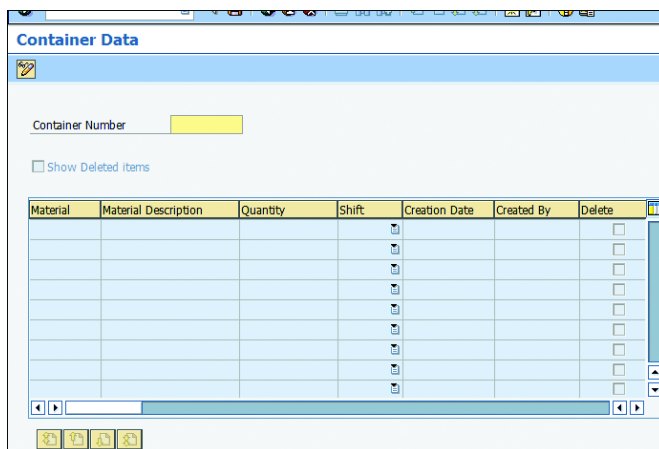


Figure 12.49 Initial Screen of Transaction

► End product

The application should eventually look like [Figure 12.49](#).

The following subsections explain the functionality of the application.

12.11.1 Application Flow

Once the user inputs the container number and presses , check if the data for that container exists in your Z table.

If the data exists, pull out the data for the container and display it in the table control. The table control should still be input-disabled. At this point, if the user wants to change or delete items, he can click the DISPLAY/CHANGE button to enter the change mode and make the necessary changes. He can toggle back to display mode by clicking the same button. [Figure 12.50](#) shows the application in change mode.

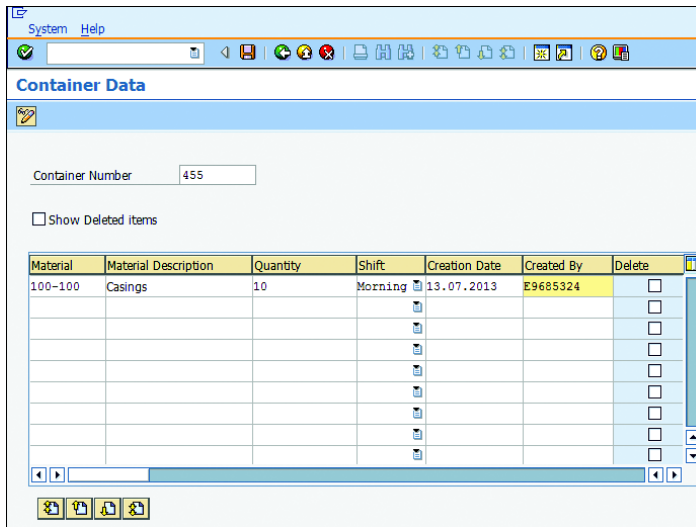


Figure 12.50 Change Mode

If there is no data for the given container in your Z table, display a pop-up message, as shown in [Figure 12.51](#).

If the user selects OK (the green checkmark), make the table control input-enabled and let the user create items. If he selects CANCEL, simply keep the table control input disabled and let the user input a new container number.

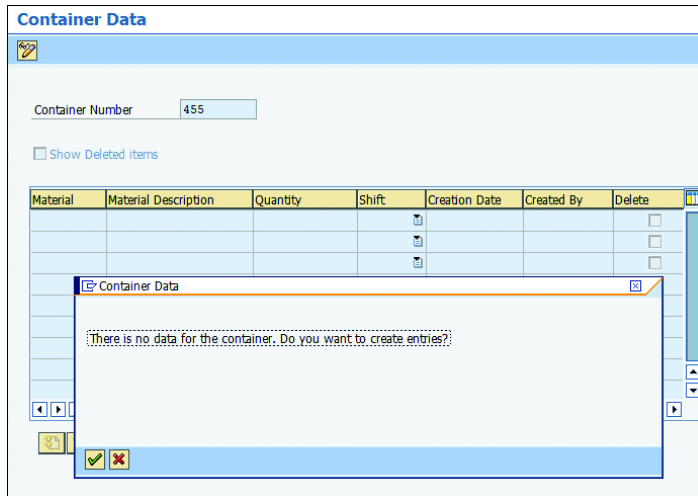


Figure 12.51 Modal Window Showing Message

At this point, clicking the DISPLAY/CHANGE toggle button should not allow the user to enter change mode. In short, the user should enter change mode only to edit existing container data.

12.11.2 Delete Functionality

If the user chooses to delete an item from the container, he can select the DELETE checkbox and click SAVE. This entry should be removed from the table control immediately (unless the SHOW DELETED ITEMS checkbox is selected). The deleted items should appear in the table control only if the SHOW DELETED ITEMS checkbox above the table control is selected.

The SHIFT dropdown list box should have three options from which the user can select: MORNING, AFTERNOON, and EVENING.

12.11.3 Validations and Autofills

The following are the validations that should be performed for the screen fields:

- ▶ MATERIAL NUMBER should be validated against table MARA.
- ▶ The MATERIAL DESCRIPTIONS column should always be input-disabled and filled automatically based on the material number entered.

- ▶ The `SAVE` button should be enabled only when in change mode.
- ▶ The `SHOW DELETED ITEMS` checkbox should be enabled only if the container has any deleted items.
- ▶ `CREATION DATE` should be filled with the current date automatically
- ▶ `CREATED BY` should be filled with the current user ID automatically.
- ▶ When the `SHOW DELETED ITEMS` checkbox is selected, all the items including the deleted ones should be shown, not just the deleted ones. The items should appear/disappear the moment the checkbox is selected or deselected; this change shouldn't require an `Enter` action.
- ▶ The application shouldn't allow the same material to be added twice.
- ▶ If the user tries to add the same material twice, show a pop-up informing him that the material already exists and asking if he wants to update the quantity of the existing material.
- ▶ The `ON SAVE` action should validate none of the items that have zero quantity.

12.12 Summary

In this chapter, we discussed how to develop dialog applications. You've seen how to work with Screen Painter to design screens in SAP, and we discussed various screen elements that can be used in SAP screens, from simple elements like I/O fields to more complex elements like table controls and custom controls.

You should now understand how screen processing is controlled by the ABAP runtime environment by calling various screen events. We also discussed how dialog modules can be called from the screen flow logic.

The concepts discussed in this chapter should help you get started with dialog programming. Of course, it takes hands-on experience to feel comfortable with dialog programming and confident to handle business requirements. We highly recommend that you complete the assignment given in the last section before proceeding further in this book. This assignment should help you feel more comfortable following along with the remainder of the topics in this book.

We'll continue our discussion of screens in the following chapters. Now that you have a good understanding of the general screen, we'll explore list screens in the next chapter.

List screens are one of the most basic ways of developing reports in ABAP. This chapter explains how to use list screens for report development.

13 List Screens

List screens, also known as *classical lists*, are one of the three types of screens in the SAP GUI. The other two type of screens, as previously discussed, are selection screens and general (dialog) screens. List screens are obsolete and have been replaced with the SAP List Viewer (ALV) for live applications, but because many old programs might still use list screens, understanding them is important to work with such programs.

In ABAP, we can create different types of programs, and not all program types support screens. Before we talk more about list screens, let's discuss the different types of ABAP programs that you can create, whether they support screens, and the various processing blocks that they support. We've already worked with some of these program types while discussing the various topics covered so far in this book.

We'll start our discussion in this chapter by looking at the various types of ABAP programs in [Section 13.1](#), which should serve as a quick recap of the different program types used so far in this book. Different program types behave differently during execution, and we'll discuss program execution in [Section 13.2](#). Note that one program can call other programs during execution. In [Section 13.3](#), we'll look at how memory is managed when a program runs and we'll explore various ways in which different programs can exchange data during runtime.

ABAP is an event-driven programming language. We discussed various events in [Chapter 7](#), but in [Section 13.4](#) we'll look at events specific to list screens. Classical lists are typically used to generate reports or issue print output; they can be used for creating normal reports (basic lists) and drilldown reports (detail lists). We'll discuss basic and detail lists in [Section 13.5](#). We'll look at classical reports that use

basic lists in [Section 13.6](#), and finally at interactive reports that use detail lists in [Section 13.7](#). We'll take what we've learned and put it to practice in [Section 13.8](#).

13.1 Program Types

When you created your first program in [Chapter 4](#), you saw that the program type attribute in the PROGRAM ATTRIBUTES window is a dropdown list with various options (see [Figure 13.1](#)).

The PROGRAM ATTRIBUTES window can be called using the menu path GOTO • ATTRIBUTES from the ABAP Editor screen. The various types of programs include executable programs, module pools, function groups, class pools, interface pools, subroutine pools, type groups, and include programs.

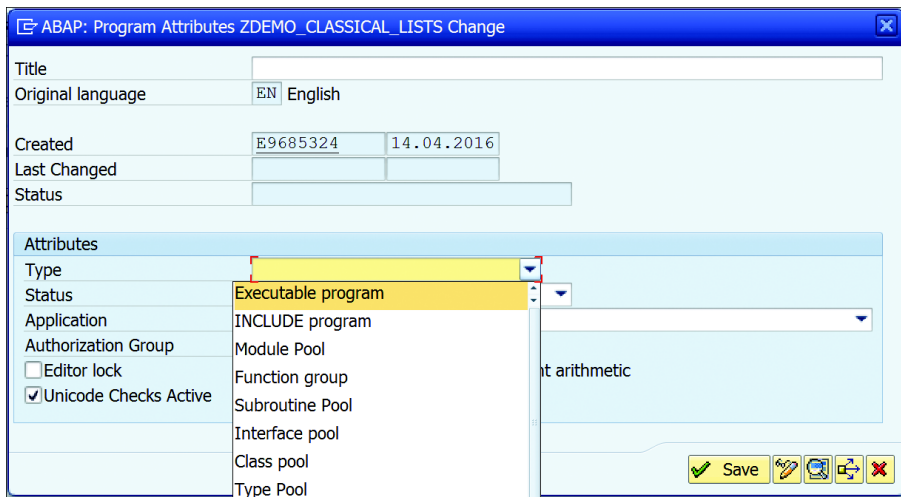


Figure 13.1 Program Types in ABAP Editor

When you run an ABAP program, you call its processing blocks in a specific sequence. The program type attribute needs to be set while creating the program. The program type determines which processing blocks the program can contain, how the program is executed and handled by the ABAP runtime environment, and whether the program can define its own screens.

In this section, we'll look at the different program types.

13.1.1 Executable Programs

Traditionally, *executable programs* are used for developing reports due to their sequence of execution. This is why executable programs are introduced with a `REPORT` statement in the code. For example, if you create an executable program with the name `ZEXAMPLE_PROGRAM`, you'll see that the first statement inserted in ABAP Editor is `REPORT ZEXAMPLE_PROGRAM`.

An executable program can contain any ABAP processing blocks, except function modules. They can also contain any number of local classes. Executable programs support screens, but it's not mandatory to have a screen defined for the program to run. You don't need to manually design a screen to work with executable programs; you can use selection screens and list screens, which are generated by specific statements in the code (such as `PARAMETERS` or `SELECT-OPTIONS` for selection screens and `WRITE` for list screens).

A selection screen allows the user to specify selection criteria, based on which the data from the database can be processed and shown as output on the list screen. This process flow makes selection screens suitable for developing reports.

Executable programs are executed with `SUBMIT` statements. Users can execute programs directly by entering the program name in Transaction SA38 or using the menu path `SYSTEM • SERVICES • REPORTING`. Even when using the transaction or menu path, the system uses the `SUBMIT` statement internally to execute the program.

An executable program can also be executed directly from ABAP Editor (Transaction SE38). However, end users may not have the required authorizations to open ABAP Editor to execute the program, because ABAP Editor is an ABAP Workbench tool and requires development authorizations to access.

Transaction SA38 provides a simple interface to run any executable program by entering the program name, as shown in [Figure 13.2](#). Optionally, a transaction code can be assigned to an executable program, allowing the program to be executed directly using the transaction code. These transactions are called *report transactions*.

Executable programs are the only programs that can be used for background processing. Executable programs are created using either ABAP Editor or Object Navigator (Transaction SE80).

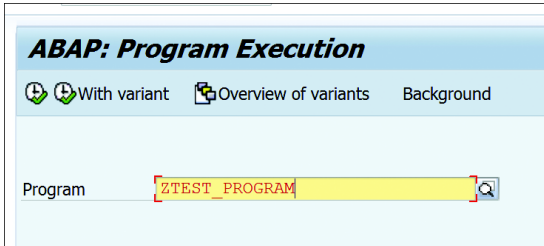


Figure 13.2 Initial Screen of Transaction SA38

13.1.2 Module Pool Programs

Module pool programs are used for dialog-oriented applications that contain a large number of screens. They're created using ABAP Editor or Object Navigator and are introduced with the `PROGRAM` statement.

A module pool program can contain all the possible processing blocks in ABAP, except function modules and reporting events (`INITIALIZATION`, `START-OF-SELECTION`, `GET`, `END-OF-SELECTION`). Module pool programs can contain as many local classes as required.

Unlike executable programs, module pool programs can't be executed directly using the program name. Instead, they're executed using a transaction code. It's mandatory for a module pool program to have at least one screen defined; that is, you can't execute a module pool program without having a screen defined for the program.

To execute a module pool program, a transaction code is created for the program, which links to the initial screen of the program when the transaction is executed. From this initial screen flow logic, dialog modules are called. The execution of module pool programs is controlled by screen flow logic. The screens for the module pool program are created using the Screen Painter tool. We'll discuss the module pool program flow further in [Section 13.2.2](#).

13.1.3 Function Groups

Function groups are introduced using the `FUNCTION-POOL` statement and are the only program type that can contain function modules. A function group can use all processing blocks in ABAP, except reporting events. They can also contain local classes.

Function groups are created using the Function Builder tool (Transaction SE37) or Object Navigator. They can be executed by a transaction code, but are generally loaded by calling the function modules of the function group. When a function module is called, the entire function group with all its function modules is loaded into the program memory. For this reason, it's important to keep only related function modules in a function group.

Similar to executable programs and module pools, a function group can have its own screens defined, including selection screens and list screens. The screens of a function group are called from the function module belonging to the group.

13.1.4 Class Pools

Class pools are created in the Class Builder tool (Transaction SE24) and are introduced with the `CLASS-POOL` statement. They act as a container to hold exactly one global class and as many local classes as required. They can't have any other processing blocks, except methods. A class pool can't have any screens defined.

Note

Executable programs, module pools, and function groups are the only three types of programs that support screens.

A class pool is loaded when the global class is called from programs. A transaction code can be created for a method of the global class so that an object of the class can be created implicitly when the method is used.

13.1.5 Interface Pools

Interface pools are created in Class Builder and are introduced with the `INTERFACE-POOL` statement. Interface pools act as a container for global interfaces that can be implemented in any global or local class. Interface pools can't contain any screens and are loaded when the interface is used in a class.

We covered the steps to create interface pools in [Chapter 8](#).

13.1.6 Subroutine Pools

Subroutine pools are used to maintain subroutines. Apart from subroutines, they can contain only a `LOAD-OF-PROGRAM` event. Subroutine pools are created in ABAP

Editor and are introduced with the `PROGRAM` statement. They can't contain any screens and are loaded when the subroutine is called from an external program.

13.1.7 Type Pools

Type pools (or type groups) are created in ABAP Data Dictionary (Transaction SE11) and are introduced with the `TYPE-POOL` statement. A type pool is used to define data types that have global visibility.

Type pools are predecessors of the general type definitions in ABAP Data Dictionary. Type pools are obsolete; we no longer use type pools to define global data types. One drawback of type groups is their missing semantic attributes; in other words, except for any comments in the code, semantic attributes can't be maintained for data types or constants defined in a type group, unlike other ABAP Data Dictionary objects, such as data elements.

In older systems, the declarations in a type group are made visible in other programs using the `TYPE-POOLS` statement.

13.1.8 Include Programs

Include programs provide library functions through which the source code maintained in an include program can be inserted in other programs using the `INCLUDE` statement. They aren't standalone programs on their own and can't be executed independently. They're used to organize the program source code into smaller editable units. They don't have introductory statement, because they don't represent stand-alone compilation units with memory of their own.

Now that you have an understanding of the various types of programs, let's discuss how these programs are executed.

13.2 Program Execution

Programs are either executed by the user directly or called by other programs. Of all the available program types, only executable programs and module pool programs are executed by the user directly; all the other programs are loaded when called from other already running programs. For example, a user will run an executable program, and if a functional module is called within this executable pro-

gram, then the function group is loaded into memory. The user doesn't execute the function group directly.

An ABAP program is made of various processing blocks, and executing an ABAP program is all about calling these processing blocks in a certain order. The execution flow of the program is entirely controlled by the runtime system. When the program is executed, the first processing block is always called from outside the program by the runtime environment, and the rest of the sequence depends on user actions.

For example, if you execute a report program, the runtime environment first calls the `INITIALIZATION` event and, if a selection screen is defined, prepares the selection screen. Based on the user actions on the selection screen, other processing blocks are called. The program type determines which of the processing blocks is first called.

In this section, we'll look at implementing both executable programs and module pool programs, the only program types executed by the user. We'll then discuss how to internally call the other program types we've discussed thus far.

13.2.1 Executable Program Flow

Executable programs are executed by entering the program name in Transaction SA38 (see [Figure 13.2](#)). From a user perspective, executable programs are the only programs that can be executed by their program names. However, often users need not know the program names, because the programs are either assigned an easy-to-remember transaction code (e.g., Transaction MRKO) or linked to menu items on the screen. From a technical perspective, executable programs are the only programs that can be executed using a `SUBMIT` statement—for example, `SUBMIT ZEXAMPLE_PROGRAM`.

When an executable program is run, the ABAP runtime environment triggers a predefined sequence of events. The sequence of events triggered by the ABAP runtime environment is as follows:

1. LOAD-OF-PROGRAM
2. INITIALIZATION
3. AT SELECTION-SCREEN
4. START-OF-SELECTION

5. GET

6. END-OF-SELECTION

The selection screen events are triggered only if the program has a selection screen defined using the `PARAMETERS` or `SELECT-OPTIONS` statement in the program. Similarly, list events are triggered if the program generates a list screen (see [Section 13.4](#) for more on list events). A list is generated automatically at the end of program execution. User actions on the list trigger list events.

The sequence of events for executable programs makes them suitable for developing reports. *Reports* provide read-only access to the database; the data from the database can be read based on the selection criteria entered by the user and shown as a report output on a list screen.

For example, when the user executes a report, we can initialize the program variables under the `INITIALIZATION` event, validate the selection screen inputs under the `AT SELECTION-SCREEN` event, select and process the data under the `START-OF-SELECTION` event, and finally generate the report output under the `END-OF-SELECTION` event.

We can use the `GET` event to process the data from a *logical database*, an ABAP program that transfers the data from multiple logically related database tables to ABAP programs in an efficient way. In the past, logical databases could only be used in executable programs using the `GET` event, but now they can be used in other programs as well without the need to use `GET` event.

13.2.2 Module Pool Program Flow

Module pool programs are executed differently than executable programs. To execute a module pool program, we need to define at least one screen and create a transaction code. The transaction code links the program with one of its screens, which becomes the initial screen. From this initial screen, dialog modules and other screens are called.

The sequence of events triggered for module pool programs is as follows:

1. LOAD-OF-PROGRAM
2. PROCESS BEFORE OUTPUT (PBO)
3. PROCESS AFTER INPUT (PAI)

The ABAP runtime environment first calls the PBO event of the initial screen and displays the screen to the user. User actions on the screen trigger PAI. After PAI, the ABAP runtime environment triggers the same screen's PBO or another screen's PBO immediately, before presenting the screen again, depending on whether the same screen is called or a different screen is called. Additional events such as PROCESS ON HELP-REQUEST (POH) and PROCESS ON VALUE-REQUEST (POV) are called if the user selects `[F1]` or `[F4]` help respectively for screen fields.

With this program flow, the role of the ABAP program is to maintain dialog modules that are called from the screen—hence the name *module pool program*.

Module pool programs (also called *dialog programs*) are used to create transactions that allow the user to modify the data in the system. Most of the standard SAP transactions that are used by users to carry out day-to-day activities such as creating sales orders (Transaction VA01) and purchase orders (Transaction ME21n) are dialog programs.

13.2.3 Calling Programs Internally

As mentioned previously, only executable programs and module pool programs are called by the user through a screen; all other program types are called internally using specific statements (e.g., using `CALL FUNCTION` to call a function module that loads the function group).

Even executable programs and module pools can be called internally from other programs that are already running. Executable programs are called internally using the `SUBMIT` statement with the program name (e.g., `SUBMIT ZDEMO_PROGRAM.`), whereas module pool programs are called using `CALL TRANSACTION` (e.g., `CALL TRANSACTION MM01.`) or `LEAVE TO TRANSACTION` (e.g., `LEAVE TO TRANSACTION MM01.`) statement.

A program call can be distinguished by whether the calling program is terminated when the new program is called or the called program is embedded in the calling program. The `SUBMIT` and `LEAVE TO TRANSACTION` statements terminate the calling program and load the new program in memory, while `SUBMIT AND RETURN` (e.g., `SUBMIT ZDEMO_PROGRAM AND RETURN`) and `CALL TRANSACTION` stack the current program with all its data before loading the called program into memory.

Once the called program's execution ends, the calling program is loaded back into memory with all its data and continues execution of the statement immediately

after the call location. This allows us to go back to the previous program in the call sequence.

Now that you have an understanding of how programs are executed, let's take a look at how memory is organized in ABAP programs.

13.3 Memory Organization

When you log onto an SAP system, a session opens in the application server and occupies memory. You can open up to sixteen sessions (depending on the restrictions set by your SAP Basis administrator), with each session occupying its own memory space in the application server and linked to its own window in SAP GUI. ABAP programs run within the session (window), and each session is independent of the others.

When you run an executable program or transaction (i.e., programs linked to a transaction code), the system creates an internal session within the main session, as shown in [Figure 13.3](#).

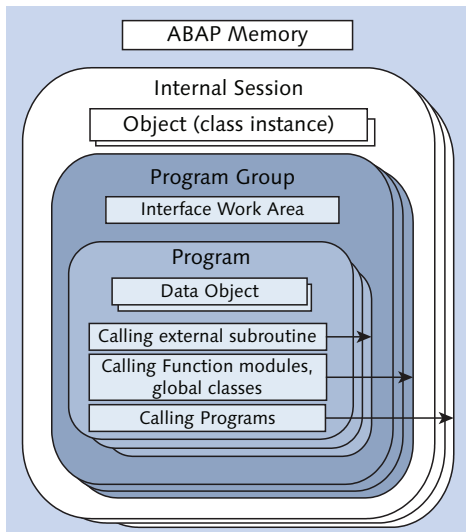


Figure 13.3 Internal Session

Multiple programs can be called within an internal *session* and are organized into *program groups*. The first program that is called in the internal session creates a

main program group. This program will be the main program of the main program group. When a function module or global class is called from the main program, it creates an additional program group. The main program of the additional group will be the first program that created the additional group.

When a program is called that isn't a function group or class pool (e.g., external subroutine call), it won't create an additional program group; instead, it's loaded into the program group of the user. It's not the program type that determines the assignment of a program to a program group; it's the introductory statement. For example, subroutine pool programs use the `PROGRAM` introductory statement. However, if the introductory statement is changed to `FUNCTION-POOL` without changing the program type in the program attributes, it will create an additional program group.

When a new program is called using the `CALL TRANSACTION` or `SUMBIT AND RETURN` statement, it creates a new internal session. The new internal session creates a new program group, with the called program being the main program of the program group. If a function module or global class is called in this new session, it will create an additional program group.

If a new program is called in the new session using the `CALL TRANSACTION` or `SUBMIT AND RETURN` statement, it will again create a new internal session. Every internal session has its own program group. You can have up to nine internal sessions in a main session. If the call sequence exceeds nine internal sessions, it will terminate all internal sessions and lead to a runtime error.

When a function module is called, the system will check if the function group to which the function module belongs already exists in the additional program group. If it exists, it won't load the function group again; instead, the function module is called directly from the additional program group, which can then access the global data of the function group. If the function group doesn't exist in the additional program group, then the function group is loaded with all its function modules, and the global data of the function group is initialized.

If the same functional module is called in a new internal session, it creates an additional program group (if not yet created in new session) if the function group doesn't exist in the additional program group and initializes all its global data. The data accessed by the function module in the new internal session will be different than the data accessed by the function module in the previous internal session.

For example, let's say two programs A and B call function modules FM1 and FM2, which belong to function group FG1. Program A is executed, which creates an internal session 1 in the main session, creating a main program group. When this program A calls the function module FM1, the function group FG1 is loaded to the internal session 1, creating an additional program group. Let's then assume FM1 sets some global data in the function group FG1. If program A calls function module FM2 next, the function group FG1 isn't reloaded, because it already exists in the additional program group, and FM2 can work with the global data set by FM1.

At this point, if program A calls the second program B using a `SUBMIT AND RETURN` statement, it will create a new internal session 2 with a main program group. The main program of this main program group will be program B. If program B calls the function module FM2, then the function group FG1 is loaded into the internal session 2, creating an additional program group because the function group exists in the additional program group of internal session 1 and not internal session 2. When function group FG1 is loaded in internal session 2, all its global data is reset, so function module FM2 called in program B will be working with the global data of the function group FG1 loaded in session 2. Therefore, the data accessed by FM2 in internal session 1 will be different from the data accessed by FM2 in internal session 2.

All the internal sessions of the main session have access to the ABAP memory of the main session. Programs in the call sequence can exchange data using ABAP memory with `IMPORT` and `EXPORT` statements. For example, the calling program can export data to ABAP memory using the `EXPORT` statement before calling a new program. Once the new program is loaded into the internal session, it can import the data from ABAP memory using the `IMPORT` statement, as shown in [Listing 13.1](#).

```
*Code in calling program
DATA v_matnr TYPE matnr VALUE 'ABCD'.
EXPORT v_matnr TO MEMORY ID 'MAT'.
CALL TRANSACTION ZXXX.
*Code in called program
DATA v_matnr TYPE matnr.
IMPORT v_matnr FROM MEMORY ID 'MAT'.
```

Listing 13.1 Using `IMPORT/EXPORT` Statements to Access ABAP Memory

ABAP memory can be used to exchange data between programs running in the same main session. Each main session has its own ABAP memory that can't be

accessed from external sessions. To exchange data between programs running in different main sessions of the user session, we use SAP memory.

Figure 13.4 shows the memory organization of ABAP programs.

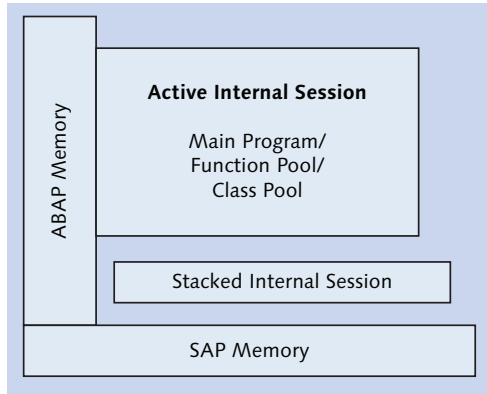


Figure 13.4 Memory Organization of ABAP Programs

All the main sessions are linked to SAP memory in the user session. Programs running in the internal session can access the SAP memory of the user session. This allows you to exchange data between programs running in different windows of SAP GUI (different main sessions) for the user session. For example, one program running in main session 1 can store data in SAP memory, and another program running in main session 2 can read the data stored by the program running in main session 1. Data can be stored in SAP memory using the `SET PARAMETER ID` statement and accessed using the `GET PARAMETER ID` statement. Input fields of the screen can be linked to SAP memory to establish default values.

In Listing 13.2, we created a `ZPROGRAM1` program in which we use the `SET PARAMETER ID` statement to save the value of the `v_matnr` field in SAP memory and assigning the value "MAT" as the memory ID. Once this value is stored in SAP memory, any program running in the user session can access this value by referring to the memory ID "MAT".

```
REPORT ZPROGRAM1.
DATA v_matnr TYPE matnr VALUE '100'.
START-OF-SELECTION.
SET PARAMETER ID 'MAT' FIELD v_matnr.
```

Listing 13.2 Storing Value in SAP Memory

In [Listing 13.3](#), we created program ZPROGRAM2 where we use the `GET PARAMETER ID` statement to read the value from SAP memory using the same memory ID assigned in [Listing 13.2](#).

Once the `GET PARAMETER ID` statement is executed, the value stored in the "MAT" memory ID will be transferred to the `v_matnr` field irrespective of whether the ZPROGRAM2 program is running in the same session as the ZPROGRAM1 program or in a different session.

```
REPORT ZPROGRAM2.
DATA v_matnr TYPE matnr.
START-OF-SELECTION.
GET PARAMETER ID 'MAT' FIELD v_matnr.
```

Listing 13.3 Reading Value from SAP Memory

Now that you have an understanding of memory organization, let's shift our attention to how list screens are processed by programs.

13.4 List Events

You've seen that executable programs, module pools, and function groups are the three types of programs that can contain screens. List screens are typically used with executable programs to generate report output.

List screens aren't created manually; they're automatically generated using the `WRITE`, `SKIP`, or `ULINE` statement. In executable programs, the list screen is presented automatically at the end of program execution. In module pool programs, the `LEAVE TO LIST-PROCESSING` statement will present the generated list.

Unlike selection screens and general screens that consist of named screen elements (e.g., input fields, radio buttons, checkboxes, etc.) and data exchanged with ABAP programs with identically named fields, list screens provide a freely definable area to output data.

The runtime environment triggers list events when a list screen is processed. *List events* allow you to prepare a list and react to user actions on the screen.

The following are the different types of list events:

- ▶ TOP-OF-PAGE
- ▶ END-OF-PAGE

- ▶ AT LINE-SELECTION
- ▶ AT USER-COMMAND

List events can be categorized as those that are triggered before the screen is shown and those that are triggered after the screen is shown. `TOP-OF-PAGE` and `END-OF-PAGE` are triggered before the list screen is displayed to the user, whereas `AT LINE-SELECTION` and `AT USER-COMMAND` are triggered by user actions on the displayed list.

In this section, we'll look at the different types of list events.

13.4.1 TOP-OF-PAGE

The `TOP-OF-PAGE` event is triggered when a list is first prepared or when a new page starts. For example, when the runtime system encounters the first statement that generates the list (`WRITE`, `SKIP`, or `ULINE`), it will trigger the `TOP-OF-PAGE` event and execute the code under the event. Once the code under the `TOP-OF-PAGE` event is processed, control will return to the statement that generated the list, and the statement will be executed.

For example, in [Listing 13.4](#), when the runtime system encounters the first `WRITE` statement under the `END-OF-SELECTION` event, before executing the `WRITE` statement control will be passed to the `TOP-OF-PAGE` event and the statements under it will be executed. Once the last of the statements in the `TOP-OF-PAGE` event is executed, the control will return to the `WRITE` statement that triggered the `TOP-OF-PAGE` event, and it will be executed. `NO STANDARD PAGE HEADING` for the `REPORT` statement disables the program title being displayed as the default page heading.

```
REPORT ZDEMO_CLASSICAL_LISTS NO STANDARD PAGE HEADING.
TYPES : BEGIN OF ty_sflight,
         CARRID TYPE S_CARR_ID,
         CONNID TYPE S_CONN_ID,
         FLDATE TYPE S_DATE,
         PRICE TYPE S_PRICE,
       END OF ty_sflight.
DATA : it_sflight TYPE STANDARD TABLE OF ty_sflight,
       wa_sflight TYPE ty_sflight.
SELECT-OPTIONS : s_carrid FOR wa_sflight-carrid.
START-OF-SELECTION.
SELECT carrid
       connid
       fldate
       price
```

```

        FROM sflight
        INTO TABLE it_sflight
        WHERE carrid IN s_carrid.
END-OF-SELECTION.
LOOP AT it_sflight INTO wa_sflight.
  AT FIRST.
    WRITE : / 'Airline Code',
            20 'Flight Connection No.',
            50 'Flight Date',
            70 'Airfare'.
  ENDAT.
  WRITE : / wa_sflight-carrid,
          20 wa_sflight-connid,
          50 wa_sflight-fldate,
          70 wa_sflight-price LEFT-JUSTIFIED.
ENDLOOP.
TOP-OF-PAGE.
WRITE : 'Report Generated On:',
        sy-uzeit.
ULINE. SKIP.

```

Listing 13.4 TOP-OF-PAGE Event

The statements under the TOP-OF-PAGE event allow you to print page headings, as shown in [Figure 13.5](#).

Report Generated On: 11:57:07			
Airline Code	Flight Connection No.	Flight Date	Airfare
AA	0017	01.05.2013	422.94
AA	0017	29.05.2013	422.94
AA	0017	26.06.2013	422.94
AA	0017	24.07.2013	422.94
AA	0017	21.08.2013	422.94
AA	0017	18.09.2013	422.94
AA	0017	16.10.2013	422.94
AA	0017	13.11.2013	422.94
AA	0017	11.12.2013	422.94
AA	0017	08.01.2014	422.94
AA	0017	05.02.2014	422.94
AA	0017	05.03.2014	422.94
AA	0017	02.04.2014	422.94
AA	0017	30.04.2014	422.94
AA	0017	28.05.2014	422.94

Figure 13.5 List Output with TOP-OF-PAGE

13.4.2 END-OF-PAGE

The END-OF-PAGE event is triggered when the page ends. The statements in this event allow you to maintain the page footer. To trigger the END-OF-PAGE event, the line count for the page should be set and a certain number of lines reserved for the footer in the program introductory statement. The line count is set using the LINE-COUNT statement in the program introductory statement—for example, REPORT ZDEMO_CLASSICAL_LISTS. Here, the line count is any numeric value and the line count for the footer is specified in parenthesis.

During the program execution, once the list page reaches the number of lines per page, the runtime system calls the END-OF-PAGE event and executes the code under it. After the END-OF-PAGE event, the system starts a new page to print the new records, which will trigger the TOP-OF-PAGE event.

Listing 13.5 shows sample code using the END-OF-PAGE event. In this example, we're using the LINE-COUNT 10(2) addition to maintain the line count per page. Here, 10(2) indicates that the total number of lines per page is ten, out of which two lines are reserved for the footer. Therefore, in this example, the END-OF-PAGE event will trigger after printing eight lines per page.

```
REPORT ZDEMO_CLASSICAL_LISTS NO STANDARD PAGE HEADING LINE-COUNT 10(2).

TYPES: BEGIN OF ty_sflight,
        carrid TYPE s_carr_id,
        connid TYPE s_conn_id,
        fldate TYPE s_date,
      END OF ty_sflight.

DATA : it_sflight TYPE STANDARD TABLE OF ty_sflight,
      wa_sflight TYPE ty_sflight.
PARAMETERS p_carrid TYPE sflight-carrid.

START-OF-SELECTION.
SELECT carrid connid fldate FROM sflight INTO TABLE it_
sflight WHERE carrid EQ p_carrid.

END-OF-SELECTION.
LOOP AT it_sflight INTO wa_sflight.
WRITE : / wa_sflight-carrid,
        wa_sflight-connid,
        wa_sflight-fldate.
ENDLOOP.

TOP-OF-PAGE.
```

```
WRITE: 'Report generated by:', sy-uname.
```

```
END-OF-PAGE.
```

```
WRITE: 'End of current page'.
```

Listing 13.5 Sample Code for END-OF-PAGE Event

13.4.3 AT LINE-SELECTION

An AT LINE-SELECTION event is triggered when the user double-clicks any record of the displayed list. The AT LINE-SELECTION event allows you to provide drill-down functionality in reports, making them interactive. The GET CURSOR statement (for more information, see [Section 13.7.3](#)) can be used to identify the column and row on which the user has double-clicked, as shown in [Listing 13.6](#).

```
DATA : column TYPE string,
      row TYPE I.
AT LINE-SELECTION.
GET CURSOR FIELD column.
GET CURSOR LINE row.
```

Listing 13.6 Using GET CURSOR

In [Listing 13.6](#), the column name will be transferred to the `column` field and the row number will be transferred to the `row` field.

13.4.4 AT USER-COMMAND

When the list screen is displayed, the list processor calls the special container screen (screen number 120). This container screen temporarily replaces the existing screen of the calling program. The container screen will inherit the same position, size, and GUI status of the screen it replaces. In executable programs, the container screen will replace the selection screen and uses a standard GUI status.

To override the standard GUI status with your own GUI status in order to add functionality, you can load a GUI status (using SET PF-STATUS) before the list is displayed. When the user selects any of the GUI buttons or menu items of the custom GUI status, the runtime environment will call the AT USER-COMMAND event. Under this event, we can maintain the functionality to perform various actions based on the button selected by the user on the list screen.

The function code of the GUI menu item or button is transferred to the `sy-ucomm` system field, which can be checked to identify the user action. [Figure 13.6](#) shows

a custom GUI status, with a PDF icon in the application toolbar. The function code set for this icon is PDF.

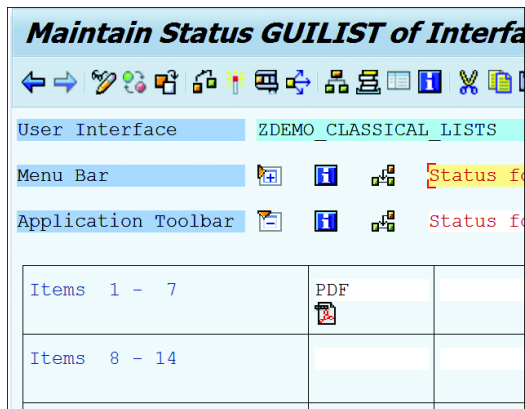


Figure 13.6 Custom GUI Status with PDF Icon

[Figure 13.7](#) shows the PDF icon displayed on the list screen when the code in [Listing 13.7](#) is executed. In [Listing 13.7](#), the GUI status is loaded under the END-OF-SELECTION event, before the list is displayed. Under the AT USER-COMMAND event, the user action is identified by checking the `sy-ucomm` system field.

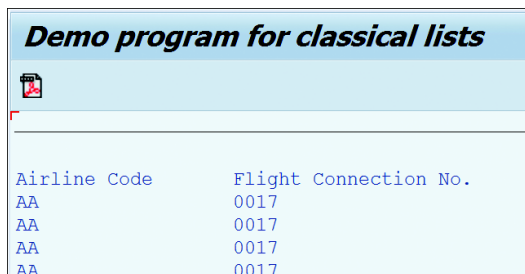


Figure 13.7 PDF Button Displayed on List Screen

```
END-OF-SELECTION.
SET PF-STATUS 'GUILIST'.
WRITE '---output---'.
AT USER-COMMAND.
IF sy-ucomm EQ 'PDF'.
ENDIF.
```

Listing 13.7 Checking `sy-ucomm` under AT USER-COMMAND

This section looked at the different list events used for list screens. In the next section, we'll look at two different kinds of lists.

13.5 Basic Lists and Detail Lists

When the list screen is displayed, the list processor has control over the program. The initial list screen is called the *basic list*, and the user actions on the basic list can create additional lists if any output statements are executed. These additional lists are called *detail lists* (or *secondary lists*).

One basic list can have up to nineteen detail lists. For example, if we use a WRITE statement under the AT LINE-SELECTION event, it creates a detail list when the user double-clicks the displayed list. When a detail list is created, it temporarily replaces the previous list on the container screen. The basic list and detail lists form a *list system*, which can have up to twenty levels. This list system allows you to create drilldown reports in which header data is shown on the basic list and the user can click the displayed header record to see the item data of the record.

Drilldown reports are also useful to divide program output into multiple levels and allow the user to navigate between different levels. For example, if we print salary details of all employees in the report output, the user can click on the employee ID to see more information about a particular employee. After checking the employee details, the user can click the BACK button to return to the main list. This allows us to break the report output into multiple levels without having to display all the information in one screen.

The code in [Listing 13.8](#) shows the basic information about the flights from table SFLIGHT on a basic list. When the user double-clicks a record in the basic list, a secondary list is generated that shows the flight schedule from table SPFLI.

```
REPORT zdemo_classical_lists NO STANDARD PAGE HEADING.
```

```
TYPES: BEGIN OF ty_sflight,
        carrid TYPE s_carr_id,
        connid TYPE s_conn_id,
        fldate TYPE s_date,
      END OF ty_sflight.
```

```
DATA : it_sflight TYPE STANDARD TABLE OF ty_sflight,
      wa_sflight TYPE ty_sflight.
PARAMETERS p_carrid TYPE sflight-carrid.
```

```

START-OF-SELECTION.
  PERFORM get_data.

END-OF-SELECTION.
  PERFORM output.

AT LINE-SELECTION.
  PERFORM detail_list.
*&-----*
*&      Form  GET_DATA
*&-----*
FORM get_data .
  SELECT carrid connid fldate FROM sflight INTO TABLE it_
sflight WHERE carrid EQ p_carrid.
ENDFORM.
*&-----*
*&      Form  OUTPUT
*&-----*
FORM output .
  LOOP AT it_sflight INTO wa_sflight.
    WRITE : / wa_sflight-carrid,
            wa_sflight-connid,
            wa_sflight-fldate.

    HIDE: wa_sflight-carrid,
          wa_sflight-connid.
  ENDLOOP.
ENDFORM.
*&-----*
*&      Form  DETAIL_LIST
*&-----*
FORM detail_list .
  TYPES: BEGIN OF ty_spfli,
         carrid   TYPE spfli-carrid,
         connid   TYPE spfli-connid,
         countryfr TYPE spfli-countryfr,
         cityfrom TYPE spfli-cityfrom,
         airpfrom TYPE spfli-airpfrom,
         fltime   TYPE spfli-fltime,
         deptime  TYPE spfli-deptime,
  END OF ty_spfli.

  DATA : lt_spfli TYPE STANDARD TABLE OF ty_spfli,
         lw_spfli TYPE ty_spfli.

  SELECT carrid
         connid
         countryfr
         cityfrom

```

```

airpfrom
fltime
deptime
    FROM spfli
    INTO TABLE lt_spfli
    WHERE carrid EQ wa_sflight-carrid
    AND connid EQ wa_sflight-connid.

LOOP AT lt_spfli INTO lw_spfli.
  WRITE : / lw_spfli-carrid,
        lw_spfli-connid,
        lw_spfli-countryfr,
        lw_spfli-cityfrom,
        lw_spfli-airpfrom,
        lw_spfli-fltime,
        lw_spfli-deptime.

ENDLOOP.
ENDFORM.

```

Listing 13.8 Drilldown List for Flights

Figure 13.8 shows the output generated by the code in Listing 13.8. In this example, we're generating the secondary lists via WRITE statements under the AT LINE-SELECTION event.

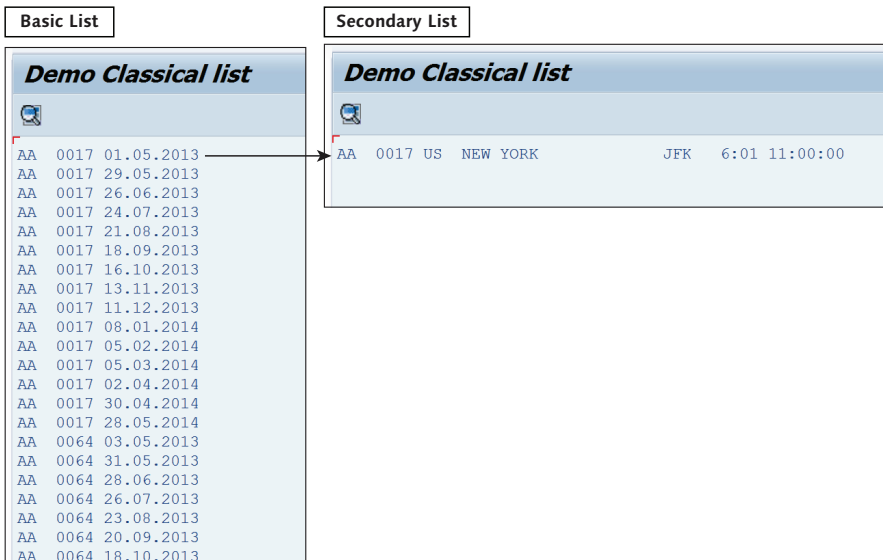


Figure 13.8 Basic and Secondary Lists

Figure 13.9 shows the representation of a list system. The main screen contains the container in which the basic list is displayed initially. When the secondary list is generated, the basic list stacks itself and the secondary list is displayed in the container. If further secondary lists are generated, the current list will be stacked, and the new list will be displayed in the container. The system field `SY-LSIND` contains the list number currently displayed in the container.

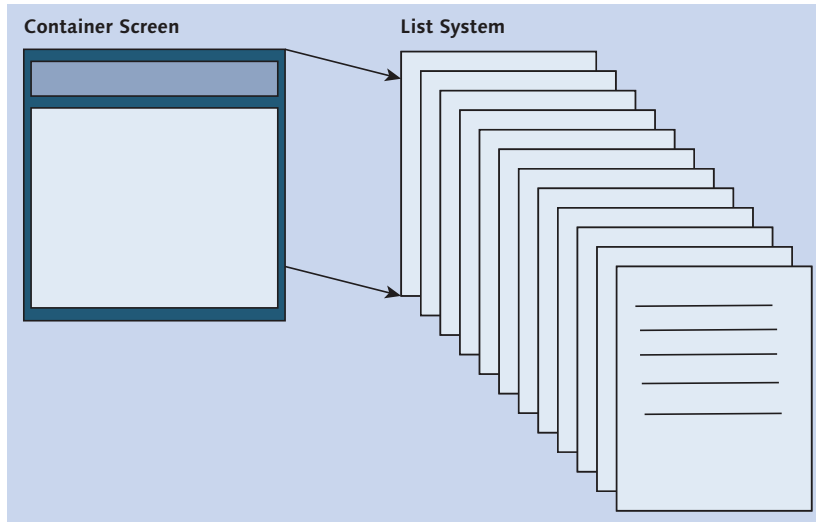


Figure 13.9 List System

From a general screen, you can call the list processor by using the `LEAVE TO LIST-PROCESSING` statement. You can display up to twenty lists in the list system. However, when the screen is processed, the list system is always initialized, which means that the basic list is called first and other lists are generated based on the user actions on the screen.

If a new screen is called from within the list system (e.g., using `CALL SCREEN`), the list system of the original screen is retained and the new screen is called. At the end of processing the new screen, the program returns to the last list level that was displayed.

The `TOP-OF-PAGE DURING LINE-SELECTION` event is executed before a detail list is called. This allows us to maintain page headings on detail lists separately from the basic list. The `SY-LSIND` system field contains the list number, which can be used to distinguish between the detail lists.

Now that you understand both basic and detail lists, let's take a look at both classical and interactive reports.

13.6 Classical Reports

Reports that use list screens to display report output are called *classical reports*. Classical reports have been replaced with ALV and are seldom used. Classical reports can be normal reports or interactive reports. *Normal reports* use only the basic list to display the report output. In other words, a normal report doesn't have any interactive elements. On the other hand, *interactive reports* can have interactive elements that react to user actions on the displayed list.

In a typical classical report, the user is presented with a selection screen to input selection criteria. Once the selection criteria are validated, the data from the database is fetched into internal tables and processed as per program requirements. The processed data is displayed on the list screen using `WRITE` statements within the internal table loop. [Listing 13.8](#) provides an example of a typical report.

13.7 Interactive Reports

Interactive reports are those reports that react to user actions on the list screen, adding an element of interactivity to the report. The `AT LINE-SELECTION` and `AT USER-COMMAND` events allow you to create interactive reports. Typically, interactive reports are designed in such a way that summarized information is displayed on the basic list and the user can then click on the displayed line to branch out to more detailed information on subsequent lists.

Various techniques are used to identify the line that the user selected on the list and pass this data to subsequent processing blocks. Some of the important ABAP statements used in interactive reports are discussed in the following subsections.

13.7.1 HIDE

The `HIDE` statement is one of the fundamental statements used with interactive reports. It's used when creating a basic list to define a *hide area* where the information that needs to be passed to subsequent lists is defined. Once a record is

written to the basic list using the `WRITE` statement, the `HIDE` statement is used immediately to store the record in the hide area. When the user selects the record from the list, it's automatically copied to the data object in the program for which the value was written in the output.

For example, say we need to develop a report to show all the billing documents created in a given date range. The requirement specifies that the user should be presented with a selection screen to input the date range.

On clicking `EXECUTE`, the report should display all the billing documents created in the given date range. The basic list should display the header data of each document, and the user can double-click the document number to see the item details of the selected document.

Listing 13.9 shows the code for this requirement.

```
REPORT zdemo_billing_report NO STANDARD PAGE HEADING.
```

```
TYPES: BEGIN OF ty_vbrk,
        vbeln TYPE vbeln_vf,
        fkdat TYPE fkdat,
        netwr TYPE netwr,
        kunrg TYPE kunrg,
      END OF ty_vbrk.
```

```
TYPES: BEGIN OF ty_vbrp,
        vbeln TYPE vbeln,
        posnr TYPE posnr_vf,
        arktx TYPE arktx,
        fking TYPE fking,
        vrkme TYPE vrkme,
        netwr TYPE netwr_fp,
        matnr TYPE matnr,
        mwsbp TYPE mwsbp,
      END OF ty_vbrp.
```

```
DATA: it_vbrp TYPE TABLE OF ty_vbrp,
      wa_vbrp TYPE ty_vbrp,
      it_vbrk TYPE TABLE OF ty_vbrk,
      wa_vbrk TYPE ty_vbrk.
```

```
SELECT-OPTIONS: s_fkdat FOR wa_vbrk-fkdat.
```

```
START-OF-SELECTION.
  SELECT  vbeln
         fkdat
         netwr
```

```

        kunrg
        FROM vbrk
        INTO TABLE it_vbrk
        WHERE fkdat IN s_fkdat.

```

```

END-OF-SELECTION.
LOOP AT it_vbrk INTO wa_vbrk.
  AT FIRST.
    FORMAT COLOR COL_HEADING.
    WRITE : 5 'BILLING DOCUMENT',
    30 'PAYER',
    50 'NET VALUE',
    80 'BILLING DATE'.
    FORMAT COLOR OFF.
  ENDAT.
  FORMAT COLOR COL_NORMAL.
  WRITE : /5 wa_vbrk-vbeln,
    30 wa_vbrk-kunrg,
    50 wa_vbrk-netwr LEFT-JUSTIFIED,
    80 wa_vbrk-fkdat.
  HIDE wa_vbrk-vbeln.
ENDLOOP.
FORMAT COLOR OFF.

```

```

AT LINE-SELECTION.

SELECT vbeln
       posnr
       arktx
       fking
       vrkme
       netwr
       matnr
       mwsbp
       FROM vbrp
       INTO TABLE it_vbrp
WHERE vbeln EQ wa_vbrk-vbeln.
LOOP AT it_vbrp INTO wa_vbrp.
  AT FIRST.
    FORMAT COLOR COL_HEADING.
    WRITE AT: /5 'ITEM',
    20 'DESCRIPTION',
    56 'BILLED QUANTITY',
    75 'SU',
    96 'NET VALUE',
    115 'MATERIAL',
    130 'TAX AMOUNT'.
    FORMAT COLOR OFF.
  ENDAT.

```

```

FORMAT COLOR COL_NORMAL.
WRITE AT: /5 wa_vbrp-posnr,
20 wa_vbrp-arktx,
55 wa_vbrp-fking,
75 wa_vbrp-vrkme,
85 wa_vbrp-netwr,
115 wa_vbrp-matnr,
130 wa_vbrp-mwsbp LEFT-JUSTIFIED.
FORMAT COLOR OFF.
ENDLOOP.
TOP-OF-PAGE.
WRITE 'Billing Document Header Data'.
TOP-OF-PAGE DURING LINE-SELECTION.
WRITE 'Billing Document Item Data'.

```

Listing 13.9 Billing Document Report

In [Listing 13.9](#), initially the document header details are selected from table `VBRK` for the given data range and displayed in the basic list. To display the data in basic list, we loop the data in internal table `it_vbrk` and use the `WRITE` statement to print the record. We used the `HIDE` statement before `ENDLOOP` to store the document number in the hide area, which will be transferred to the `AT LINE-SELECTION` processing block when the user double-clicks the document number in the basic list.

The hide area is the area in the list buffer in which global variables can be stored for every row of a screen list via the `HIDE` statement. Without using the `HIDE` statement, there's no way to identify the document number on which the user double-clicked in order to fetch subsequent item details for the document inside the `AT LINE-SELECTION` event. When the user double-clicks the document number, its value will be transferred to the original `wa_vbrk-vbeln` field in the program. We use this field to select the data from the item table `VBRP`.

We print the page heading for the basic list under the `TOP-OF-PAGE` event and the page heading for the detail list under the `TOP-OF-PAGE DURING LINE-SELECTION` event.

[Figure 13.10](#) shows the expected output in the basic list and [Figure 13.11](#) shows the expected output in the detail list for the given requirement. The detail list is called when the user double-clicks the document number displayed in the basic list.

Billing Document Header Data				
BILLING DOCUMENT	PAYER	NET VALUE	BILLING DATE	
90039748	300760	2,500.00	24.04.2015	
90039749	300760	24,148.00	24.04.2015	
90039750	300760	24,148.00	24.04.2015	
90039751	300760	24,148.00	24.04.2015	
90039752	300760	24,148.00	24.04.2015	
90039753	300760	24,148.00	24.04.2015	
90039754	301111	31,822.00	24.04.2015	
90039755	301111	31,822.00	24.04.2015	
90039756	301111	31,822.00	24.04.2015	
90039757	301111	31,822.00	24.04.2015	
90039758	301111	31,822.00	24.04.2015	

Figure 13.10 Basic List for Code in Listing 13.9

Billing Document Item Data						
ITEM	DESCRIPTION	BILLED QUANTITY	SU	NET VALUE	MATERIAL	TAX AMOUNT
000010	Biopur Consumables	1.000	EA	1,300.00	BIO_CONS001	0.00
000020	Micro Manipulators	1.000	EA	2,550.00	MIC_ANTIPL01	0.00
000030	Biopur Consumables	1.000	EA	1,258.00	BIO_CONS001	0.00
000040	Thermocyclers	1.000	EA	3,027.00	THEM_CYC03	0.00
000050	Photometers	1.000	EA	1,779.00	PHOT_MET002	0.00
000060	CO2 Incubators	1.000	EA	3,659.00	CO2_INCU002	0.00
000070	Shakers	1.000	EA	2,246.00	SHAK_003	0.00
000080	Easypet	1.000	EA	201.00	EAS_PET003	0.00
000090	Photometers	1.000	EA	1,850.00	PHOT_MET002	0.00
000100	CO2 Incubators	1.000	EA	3,778.00	CO2_INCU002	0.00
000110	Shakers	1.000	EA	2,300.00	SHAK_003	0.00
000120	Easypet	1.000	EA	200.00	EAS_PET003	0.00

Figure 13.11 Detail List for Code in Listing 13.9

13.7.2 READ LINE

Similar to the HIDE statement, the READ LINE or READ CURRENT LINE statement can be used to capture the value when the user double-clicks a record. Using the READ LINE statement, you can read any line by supplying the row number. You can use the GET CURSOR statement to dynamically identify the row on which the user double-clicked and use that row number with the READ LINE statement to get the field value, as shown in Listing 13.10. To read the line in succession, use the READ CURRENT LINE statement.

```
DATA row TYPE sy-index.
GET CURSOR LINE row.
READ LINE row FIELD VALUE wa_vbrk-vbeIn.
READ CURRENT LINE FIELD VALUE wa_vbrk-vbeIn.
```

Listing 13.10 Using READ LINE and READ CURRENT LINE

In [Listing 13.10](#), we are defining a row variable of type `sy-index` to identify the cursor position. The `GET CURSOR` statement fills the variable `row` with the line number in the list of the current cursor position. The `READ LINE` statement reads the field value of the supplied row number, and the `READ CURRENT LINE` statement automatically reads the line where the cursor is currently positioned.

13.7.3 GET CURSOR

To retrieve information about the cursor position, we can use the `GET CURSOR` statement to determine the current row and current field, as shown in [Listing 13.11](#).

```
*Using GET CURSOR to fetch row number
DATA row TYPE sy-index.
GET CURSOR LINE row.
*Using GET CURSOR to fetch column (field) name
DATA field TYPE string.
GET CURSOR FIELD field.
```

Listing 13.11 Using GET CURSOR

13.7.4 DESCRIBE LIST

The `DESCRIBE LIST` statement provides information about the attributes of the list levels. Using this statement, we can retrieve information such as the number of lines or number of pages of a list, get the page number of a particular line number, or get the properties of the particular page. Refer to the keyword documentation to see all the possible additions that can be used with this statement.

[Listing 13.12](#) uses the `DESCRIBE LIST` statement to identify the last page and then read the number of lines on the last page. In this example, the `last_page` field will contain the total number of pages generated. Using this value, we read the number of lines on the last page. The `lines` field will contain the number of lines in the last page.

```
DATA: last_page TYPE I,
      lines      TYPE I.
AT LINE-SELECTION.
DESCRIBE LIST NUMBER OF PAGES last_page.
DESCRIBE LIST PAGE last_page LINES lines.
```

Listing 13.12 Using DESCRIBE LIST Statement

13.8 Practice

Learning is all about putting theory into practice. The following practice examples should help you explore classical lists (list screens):

1. An `END-OF-PAGE` event is triggered when the page is filled. For example, if you have set the line count as 10 (2), it means the total number of lines per page is ten, out of which two lines are reserved for the footer. Therefore, the system will trigger the `END-OF-PAGE` event after printing eight lines on the page. However, if the page isn't filled, the `END-OF-PAGE` event isn't triggered. For example, if you've set the line count as 10 (2) but only six lines are printed in the output, then `END-OF-PAGE` isn't triggered.

The code in [Listing 13.13](#) doesn't trigger the `END-OF-PAGE` event. For this practice example, modify the code (without printing extra lines) in such a way as to trigger the `END-OF-PAGE` event even if the page isn't filled.

```
REPORT ZDEMO_LIST LINE-COUNT 12(2).
START-OF-SELECTION.
DO 6 TIMES.
WRITE : / 'Demo on end-of-page'.
ENDDO.
END-OF-SELECTION.
WRITE 'END-OF-PAGE event is executed'.
```

Listing 13.13 Code for Exercise 1

Hints

- ▶ Find out how many lines are printed and calculate the difference by checking a system field that gives the line count. Use the `SKIP` statement to print blank lines.
- ▶ Explore the `RESERVE` keyword.

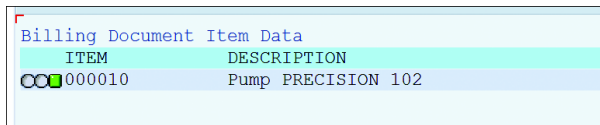
2. Create a basic list and multiple detail lists. For example, when you double-click a line in the basic list, it should open a detail list (use the `WRITE` statement in the `AT LINE-SELECTION` event to print some dummy data). When you double-click a line in the detail list, it should open a new detail list. Double-clicking a line in this list should open further detail lists. For example, say you've opened five detail lists. When you click the `BACK` button on the fifth detail list, it will take you to the fourth detail list. Clicking the `BACK` button on the fourth detail list will take you to the third detail list and so on, until you reach the basic list.

For this exercise, find out how you can reach the basic list directly when you click on any detail list in the system.

Hint

Explore the `sy-lsind` system field.

- Discover how to print icons on the list screen. We can print different icons to display the status of a record when displaying records in the output. [Figure 13.12](#) shows a green light icon in the first column of the record to signify that the record is processed successfully. For this exercise, find out how to print icons as shown in [Figure 13.12](#).



Billing Document Item Data	
ITEM	DESCRIPTION
000010	Pump PRECISION 102

Figure 13.12 Icon Displayed in Output

HINT

Explore all the additions for the `WRITE` statement.

13.9 Summary

In this chapter, we discussed the various program types in ABAP, which processing blocks they support, and how program types determine program execution flow. We also discussed different ways to execute programs, both internally in the program code and through SAP GUI and covered how understanding memory management is vital to determining how data is exchanged between programs. We looked at how programs run in an internal session and what happens behind the scenes when different programs are called internally.

In addition, we discussed various list events and explored when the events are called and their purposes. You also learned how different events are used to create interactive reports.

In the next chapter, we'll discuss another type of screen in ABAP: the selection screen.

Selection screens allow users to select what information they see in a specific application. This chapter discusses how to design events that occur in a selection screen.

14 Selection Screens

Selection screens are special screens created by using specific ABAP statements. Unlike dialog screens, which are created in Screen Painter, selection screens are auto-generated and don't require manual design. Selection screens are used when a user can provide the selection criteria to restrict the data processed by a program.

In this chapter, we'll look at the features of selection screens. We'll start by discussing how to define selection screens in [Section 14.1](#). When the selection screen is processed, the ABAP runtime environment triggers different selection screen events that allow us to prepare and process input fields on the selection screen. We'll explore the selection screen events in [Section 14.2](#) and how these events can be used for validating user inputs in [Section 14.3](#). A selection screen can contain many input fields to restrict the data being processed by the program.

Often, users run a report frequently with same set of inputs. For example, a user may input same selection criteria to check daily stock movements for a particular plant each time he runs the report. To avoid having to repeatedly input the same selection criteria each time the report is run, we can create a selection screen variant to save the selection criteria. We'll discuss how to create and use selection screen variants in [Section 14.4](#).

Reports can be executed online by the user through an SAP GUI screen or they can be scheduled to execute in the background. Background execution of a program is useful if the program is processing huge amounts of data and takes a long time to complete execution. Sometimes, a user may also want to run a report at a fixed time of day at regular intervals. We'll discuss how to execute programs in the background in [Section 14.5](#).

Some selection screens may have too many fields, and many fields may not be relevant for the given context. For example, a user without the authorization to settle a consignment may not require fields that allow the user to input settlement criteria for consignments. We can dynamically display or hide selection screens to improve user experience. We'll discuss dynamically manipulating screen elements in [Section 14.6](#).

Finally, we'll conclude this chapter in [Section 14.7](#), with a discussion of how programs can be called internally and how to use selection screen fields as an interface to pass data between programs.

14.1 Defining Selection Screens

Selection screens typically are used with reports that run executable programs to take user input. However, selection screens can also be defined with module pools and function groups.

With executable programs, a standard selection screen is defined with the `PARAMETERS` or `SELECT-OPTIONS` statement. The standard selection screen is assigned screen number 1000 and is automatically called when the program is executed. You can also define as many user-defined selection screens as you wish in the program.

User-defined selection screens allow you to dynamically present the user with a selection screen for further data restriction based on the user actions. For example, in a drilldown report, when the user double-clicks a line in the basic list, you can call a user-defined selection screen to allow the user to input restrictions for the data to be displayed on the secondary list.

Each user-defined screen is assigned a four-digit screen number—any number other than screen number 1000, which is reserved for the standard selection screen. The user-defined selection screens are defined between the `SELECTION SCREEN BEGIN OF SCREEN` and `SELECTION SCREEN END OF SCREEN` statements and are called using the `CALL SELECTION-SCREEN` statement.

[Listing 14.1](#) shows the syntax to define a standard selection screen and user-defined selection screen.

```
*Defines a standard selection screen.
PARAMETERS p_carrid TYPE s_carrid.
```

```
*Defines a user-defined selection screen with screen number 2000
SELECTION-SCREEN BEGIN OF SCREEN 2000.
PARAMETERS p_connid TYPE s_connid.
SELECTION-SCREEN END OF SCREEN 2000.
```

Listing 14.1 Defining Selection Screen

In [Listing 14.1](#), the `PARAMETERS p_carrid TYPE s_carrid` statement defines a standard selection that is assigned the default screen number 1000. Any `PARAMETERS` or `SELECT-OPTIONS` declarations that do not explicitly belong to a `SELECTION-SCREEN BEGIN OF SCREEN` block will be part of the standard selection screen.

To define a user-defined selection screen, use the `SELECTION-SCREEN BEGIN OF SCREEN` statement and assign a custom screen number. All the selection screen fields defined between the `SELECTION-SCREEN BEGIN OF SCREEN` and `SELECTION-SCREEN END OF SCREEN` statements will belong to this user-defined selection screen.

For executable programs, the standard selection screen is called automatically when the program is executed. User-defined selection screens should be called explicitly using the `CALL SELECTION-SCREEN` statement. For example, to call a user-defined selection screen with screen number 2000, you use the statement `CALL SELECTION-SCREEN 2000`.

In function groups and module pools, you can only define user-defined selection screens. In function modules, you call the user-defined selection screen with the `CALL SELECTION-SCREEN` statement. User-defined selection screens can be used as initial screens in module pool programs. The selection screen can also be included as a subscreen on a dialog screen.

The syntax to define and call a selection screen is the same irrespective of the program type. To define a selection screen as a subscreen, use the `AS SUBSCREEN` addition with the `SELECTION-SCREEN BEGIN OF SCREEN` statement. [Listing 14.2](#) shows an example to define a user-defined selection screen as a subscreen.

```
PROGRAM ZSUBSCREEN_EXAMPLE.
DATA v_matnr TYPE matnr.
SELECTION-SCREEN BEGIN OF SCREEN 500 AS SUBSCREEN.
SELECT-OPTIONS s_matnr FOR v_matnr.
SELECTION-SCREEN END OF SCREEN 500.
```

Listing 14.2 Defining Subscreen

In [Listing 14.2](#), we defined a subscreen with screen number 500 that contains a `SELECT-OPTIONS` field, `S_MATNR`. This subscreen can be included in the main screen

just like any regular subscreen created via Screen Painter, as discussed in [Chapter 12](#). The advantage of using a user-defined selection screen as a subscreen and then including it on a main screen is that with the user-defined selection screen, you can have range fields (select options) for input. You can't define range fields directly with Screen Painter.

Selection screens provide a standardized UI for users to enter selection criteria. They also allow users to input single values and complex selection criteria. Selection screens also provide an interface to transfer data between selection screen fields of executable programs called with the `SUBMIT` statement.

Now that you understand how to define selection screens using the program types, in the next section we'll discuss the various selection screen events that can be used.

14.2 Selection Screen Events

Selection screen events are called when a selection screen is processed. The standard selection screen event is `AT SELECTION-SCREEN`.

The different variations of the `AT SELECTION-SCREEN` event are as follows:

► `AT SELECTION-SCREEN OUTPUT`

This event is triggered by the `PBO` event of the selection screen. The code in this event is executed before the selection screen is displayed to the user. This event allows you to set default values for the selection screen fields or dynamically change field properties (e.g., enable or disable fields). This event is triggered before the selection screen is called.

You can use the `AT SELECTION-SCREEN OUTPUT` event to assign a default value. In this example, we set the current date as the default value for the `p_date` field. The `p_date = sy-datum` statement will be executed before the selection screen is displayed to the user. For example:

```
REPORT ZEXAMPLE_PROGRAM.
PARAMETERS p_date TYPE sy-datum.
AT SELECTION-SCREEN OUTPUT.
p_date = sy-datum.
```

► `AT SELECTION-SCREEN ON <field>`

This event is triggered in the `PAI` event of the selection screen, when the data

of the parameters field or the line of `SELECT-OPTIONS` is transferred to the ABAP program. This event is used to validate the field input. If an error or warning message is shown, the field will be automatically input-enabled. For example:

```
PARAMETERS p_matnr TYPE matnr.  
AT SELECTION-SCREEN ON p_matnr.
```

► **AT SELECTION-SCREEN ON END OF <sel>**

This event is triggered when the selection table of the `SELECT-OPTIONS` field is fully passed to the ABAP program after the user action on the multiple selection window. This event allows you to validate all the entries of the selection table. For example:

```
SELECT-OPTIONS s_matnr FOR v_matnr.  
AT SELECTION-SCREEN ON END OF s_matnr.
```

► **AT SELECTION-SCREEN ON BLOCK**

You can organize the selection screen fields into multiple blocks. This event is triggered when the data of all the fields of the block is transferred to the program. If an error or warning message is raised in this event, all the fields in the block will be input-enabled. For example:

```
SELECTION-SCREEN BEGIN OF BLOCK b1.  
PARAMETERS : p_matnr TYPE matnr,  
              P_spras TYPE spras.  
SELECTION-SCREEN END OF BLOCK.  
AT SELECTION-SCREEN ON BLOCK b1.
```

► **AT SELECTION-SCREEN ON RADIO BUTTON GROUP**

This event is triggered when all the fields of a radio button group are passed to the ABAP program. If an error or warning message is raised in this event, all the fields in the radio button group will be input-enabled. The individual radio button fields are not passed to the ABAP program independently, so they don't trigger `AT SELECTION-SCREEN ON` field event. For example:

```
PARAMETERS : p_day RADIOBUTTON GROUP rb1,  
              p_month RADIOBUTTON GROUP rb1,  
              p_year RADIOBUTTON GROUP rb1.  
  
AT SELECTION-SCREEN ON RADIOBUTTON GROUP rb1.
```

► **AT SELECTION-SCREEN**

This is a standard selection screen event and is triggered after all the selection screen events are executed. This event can be used to validate user inputs. If an error or warning message is raised in this event, all the fields of the selection

screen will be input-enabled and the selection screen becomes ready for input again.

► **AT SELECTION-SCREEN ON HELP-REQUEST FOR <field>**

This is the POH event of the selection screen. This event is triggered when the user selects **F1** help for the field. For example:

```
PARAMETERS p_matnr TYPE matnr.
AT SELECTION-SCREEN ON HELP-REQUEST FOR p_matnr.
```

► **AT SELECTION-SCREEN ON VALUE-REQUEST FOR <field>**

This is the POV event of the selection screen. This event is triggered when the user selects **F4** help for the field. For example:

```
PARAMETERS p_matnr TYPE matnr.
AT SELECTION-SCREEN ON VALUE-REQUEST FOR p_matnr.
```

► **AT SELECTION-SCREEN ON EXIT-COMMAND**

This event is triggered when the user selects the BACK, EXIT, or CANCEL button on the selection screen. Any cleanup code can be handled in this event.

Now that you understand different variations of the `AT SELECTION-SCREEN` event, in the next section we'll discuss how to validate the selection screen inputs.

14.3 Input Validations

Selection screen inputs should be validated before using the selection criteria to process the data from database. This allows us to identify any invalid inputs and show any resulting messages to the user so that the input can be corrected. Often, a user may not realize he's entered the wrong input on the selection screen and may assume that the program isn't working if he sees the wrong output. For such reasons, it's important to validate all selection screen inputs and alert the user if required.

Selection screen fields should always be validated using one of the selection screen events. Otherwise, when an error message is shown the selection screen becomes unusable, preventing the user from correcting his input. The user would then have to refresh the transaction to make a correct input.

If the selection screen has many fields, it's recommended to validate each individual field using the `AT SELECTION-SCREEN ON FIELD` event. This allows you to

automatically highlight the field with an error by disabling all the other fields on the screen and enabling only the field with the error.

In [Listing 14.3](#), we defined two selection screen fields, `p_matnr` and `p_spras`. When we use the `AT SELECTION-SCREEN ON` field event to independently validate the fields, the system will automatically highlight the field in error by input-enabling the error field and making the remaining fields on the selection screen input-disabled.

```
PARAMETERS : p_matnr TYPE matnr,
              p_spras TYPE spras.
AT SELECTION-SCREEN ON p_matnr.
*Validate p_matnr input field
AT SELECTION-SCREEN ON p_spras.
*Validate p_spras input field.
```

Listing 14.3 Validating Input Fields Independently

If a group of fields should be validated together, it's recommended to group the fields into a block (using `SELECTION-SCREEN BEGIN OF BLOCK`) and use the `AT SELECTION-SCREEN ON BLOCK` event. This allows you to automatically highlight the fields of the block by disabling all the other fields on the screen and enabling only the fields of the block.

In [Listing 14.4](#), we grouped the `p_matnr` and `p_spras` fields into block `b1`. When we use the `AT SELECTION-SCREEN ON BLOCK b1` event to validate the input fields, both `p_matnr` and `p_spras` will be input-enabled together when an error message is raised in the event.

```
SELECTION-SCREEN BEGIN OF BLOCK b1.
PARAMETERS : p_matnr TYPE matnr,
              p_spras TYPE spras.
SELECTION-SCREEN END OF BLOCK.
SELECTION-SCREEN BEGIN OF BLOCK b2.
PARAMETERS : p_werks TYPE werks_d,
SELECTION-SCREEN END OF BLOCK.

AT SELECTION-SCREEN ON BLOCK b1.
*Validate p_matnr and p_spras input fields together
```

Listing 14.4 Validating Input Fields as Related Group

Similarly, when validating radio buttons, use the `AT SELECTION-SCREEN ON RADIO BUTTON GROUP` event to ensure all the radio buttons are enabled when an error message is shown. For example, in [Listing 14.5](#), if an invalid radio button option

is selected based on the context, all the radio buttons will be input-enabled for the user to make a new selection when the error message is raised within the AT SELECTION-SCREEN ON RADIOBUTTON GROUP event.

```
PARAMETERS : p_day RADIOBUTTON GROUP rb1,
              p_month RADIOBUTTON GROUP rb1,
              p_year RADIOBUTTON GROUP rb1.
```

```
AT SELECTION-SCREEN ON RADIOBUTTON GROUP rb1.
*Validate radio button
```

Listing 14.5 Validating Radio Button Group

Next, we'll take a comprehensive look at selection screen variants, from their creation to user-specific variables.

14.4 Selection Screen Variants

Selection screens allow users to input selection criteria. When executing a report, users will input the selection criteria for the report. Based on the selection criteria, the report will process the records. For example, if the user wants to check the data for a given date range, then the report will process the data for the given date range only.

Sometimes, selection screens consist of many input fields and allow the user to provide complex selection criterion, but most often users run a report with the same selection criteria repeatedly. For example, a user in a particular plant may want to run a report by restricting the data to his particular plant each time. If the selection criteria are complex, it may be time-consuming to input the same data on the selection screen each time the report is used. If the report is run with same set of selections often, then the selection criteria can be saved as a *variant*. Once the selection data is saved as a variant, the user can use the variant to fill the selection screen and avoid manually inputting the same data over and over again.

We can create any number of variants for programs that have a selection screen defined. Therefore, each user can create his own variant to run the report. Variants are local to programs and are not global. Variants also can be used to hide the selection screen fields. This allows users to customize the selection screen by hiding the non-relevant optional fields, which is particularly useful when a report contains a large selection screen with many fields.

As we discussed, programs can contain multiple selection screens, so variants can be created for different selection screens of the program. Apart from using the variant to fill the selection screen fields on the screen, variants can also be used to run reports in background mode.

In this section, we'll discuss how to create a selection screen variant and different attributes of the variant in [Section 14.4.1](#) and [Section 14.4.2](#), then we'll cover how to fill the input fields with contents from database table in [Section 14.4.3](#).

Sometimes, the data that should be populated in the selection screen fields may be dynamic. For example, the user may want to use the variant in such a way that the date and time fields are always filled with current date and current time during the execution of the program. We'll discuss how to dynamically calculate the date in [Section 14.4.4](#) and the time in [Section 14.4.5](#). Selection screen fields can also be defaulted to user-specific values maintained in a user's personal settings. We'll look at these user-specific variables in [Section 14.4.6](#).

14.4.1 Creating a Variant

A variant can be created from the ABAP Editor initial screen or directly from the selection screen. The following steps walk through how to create or edit a selection screen variant:

1. To create or change a variant, from the ABAP Editor initial screen enter the program name, select the VARIANTS radio button under SUBOBJECTS, and click CHANGE, as shown in [Figure 14.1](#).

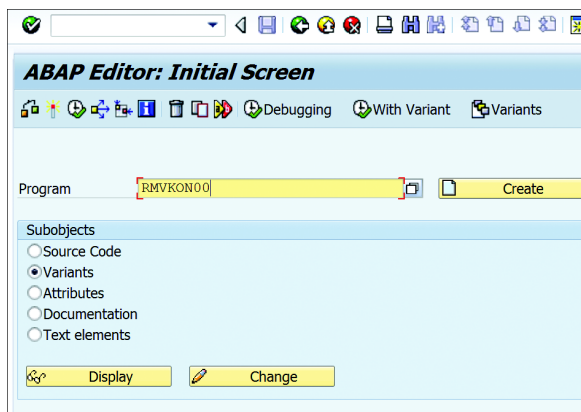


Figure 14.1 Create/Change Variant from ABAP Editor: Initial Screen

2. On the ABAP: VARIANTS initial screen, enter a name for your variant and select the VALUES radio button under SUBOBJECTS. Click the CREATE button when you've finished (see [Figure 14.2](#)).

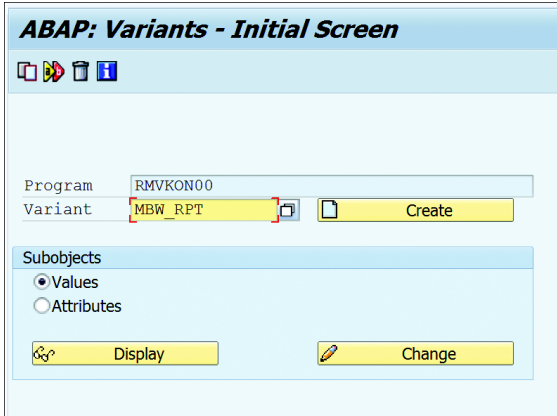


Figure 14.2 Variants Initial Screen

3. The system will call the selection screen for which the variant is being created, as shown in [Figure 14.3](#).

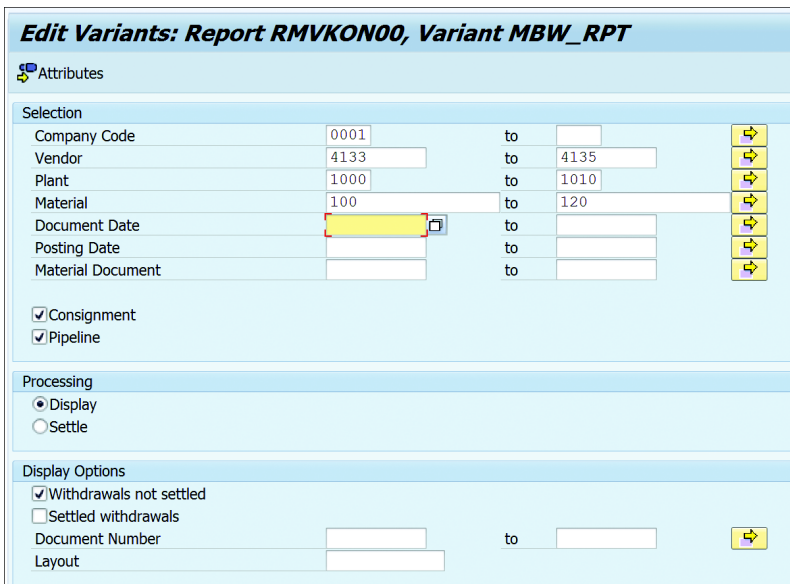


Figure 14.3 Selection Screen of Transaction MRKO

In this example, the system is showing the standard selection screen of program RMVKON00 (Transaction MRKO). On this selection screen, fill the selection criteria and click the ATTRIBUTES button in the application toolbar. The ATTRIBUTES button allows you to maintain different attributes for the selection screen fields.

4. The VARIANT ATTRIBUTES screen, as shown in [Figure 14.4](#), has options that can be maintained for the selection screen fields. Some of the important options include the follow:

- ▶ DESCRIPTION: Maintain a meaningful description for the variant.
- ▶ ONLY FOR BACKGROUND PROCESSING: Selecting this checkbox restricts the variant to be used only for background processing.
- ▶ PROTECT VARIANT: Selecting this checkbox will allow the variant to be changed only by the person who created it.
- ▶ ONLY DISPLAY IN CATALOG: When this option is checked, the variant will not be displayed in **[F4]** help; it will be displayed only in the catalog.
- ▶ SYSTEM VARIANT: This field can't be set manually; it's automatically set by the system. If you want the variant to be transported automatically with the program, then you must create a system variant by starting the variant name with CUS&.

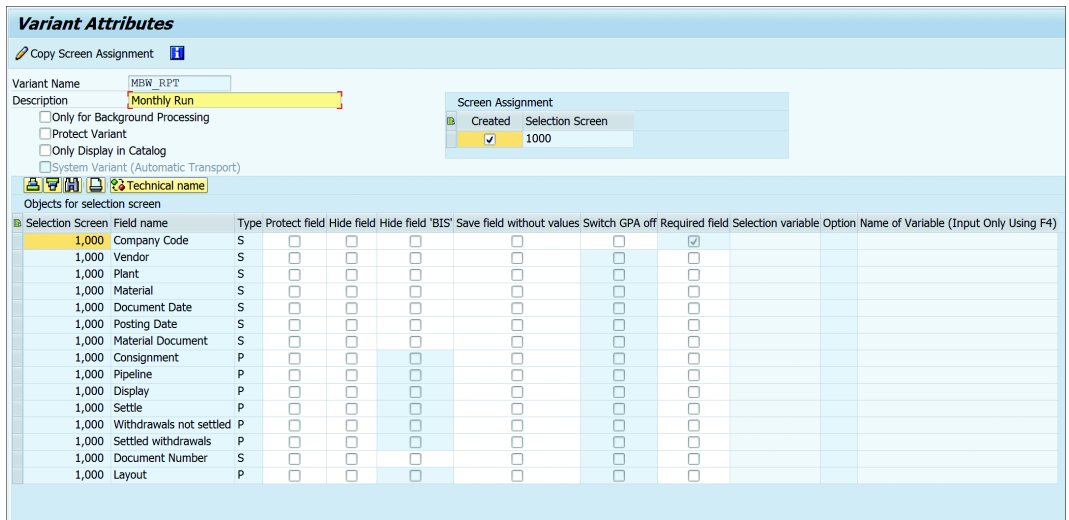


Figure 14.4 Attributes Screen

- ▶ **SCREEN ASSIGNMENT:** This box lists all the defined selection screens for which this variant can be assigned.
- ▶ **INFORMATION icon:** For each field of the selection screen, certain attributes can be set to manipulate the field. The **INFORMATION** button in the application toolbar documents all of the fields and their purposes—some of which we'll discuss in [Section 14.4.2](#).

5. Click **SAVE** in the standard toolbar to save the variant.

Once the variant is created, it can be accessed by choosing the **GET VARIANT** button in the application toolbar of the selection screen, as shown in [Figure 14.5](#).

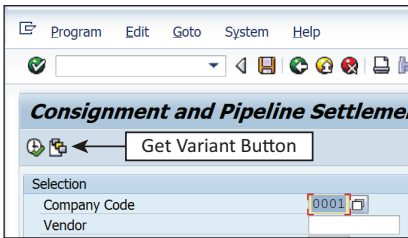


Figure 14.5 Get Variant

The variant can also be created by filling in the information in the program's selection screen and clicking the **SAVE** button in the standard toolbar. For example, [Figure 14.6](#) shows the standard selection screen called when we execute Transaction MRKO. On this screen, you can fill the selection criteria and click the **SAVE** button in the standard toolbar to save the selection criteria as a variant.

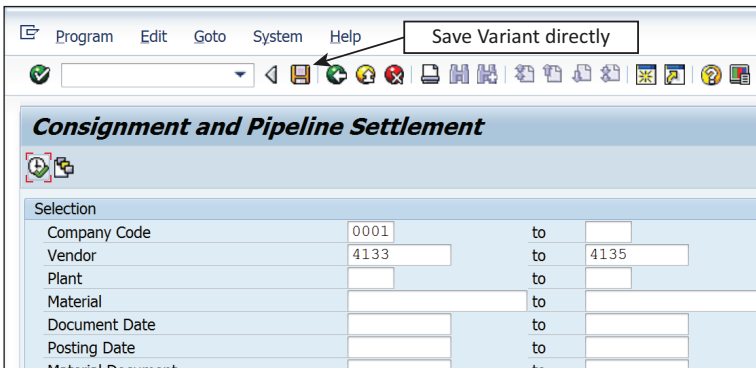


Figure 14.6 Save Variant Directly

Now that you know how to create and edit variants, let's explore the different attributes associated with selection screen fields.

14.4.2 Variant Attributes

The attributes of selection screen fields can be changed in a variant. For example, you can hide a field or restrict overwriting a field value. The INFORMATION icon on the VARIANT ATTRIBUTES screen (refer back to [Figure 14.4](#)) provides documentation to explain the purpose of each option.

Some of the important options include the following:

- ▶ **FIELD TYPE**
Indicates whether the field is of type `PARAMETERS` or `SELECT-OPTIONS`.
- ▶ **PROTECTED**
When this option is selected, the user can't overwrite the value in the field on the selection screen.
- ▶ **HIDE FIELD**
Selecting this checkbox will hide the field from the selection screen. If a value is maintained for the field in the variant, it will be used, but the field won't be displayed on the selection screen. This setting is useful if the selection screen has too many fields.
- ▶ **HIDE FIELD BIS**
This checkbox is enabled only for `SELECT-OPTIONS` fields. Selecting this checkbox will hide the range selection (only the low field is displayed).
- ▶ **SAVE FIELD WITHOUT VALUES**
If this is checked, the system will ignore the value saved for the field in the variant.
- ▶ **SWITCH GPA**
This field will disable the link to SAP memory so that the value isn't imported from SAP memory (`GET PARAMETER`). SPA/GPA is not active in background processing.
- ▶ **REQUIRED FIELD**
Marks the field as mandatory.

► **SELECTION VARIABLE**

To set dynamic values for selection screen fields, use the selection variable field. It supports four types of selection variables: table variables from table TVARVC, dynamic date calculations, dynamic time calculations, and user specific variables. For dynamic date and time calculations, the field should be of type D or T respectively.

In the following sections, we'll discuss the SELECTION VARIABLE types.

14.4.3 Table Variables from Table TVARVC

There may be times when the user wants to input a set of values on the selection screen, but these values may be updated frequently. In such a scenario, instead of modifying the variant each time the value needs to be changed, the field can be linked to the variant in table TVARVC.

To use the TABLE VARIABLES FROM TABLE TVARVC option, we need to first create a variant in table TVARVC. Proceed with the following steps:

1. Open Transaction STVARV.
2. Go to change mode by clicking the DISPLAY/CHANGE button, then click the CREATE button in the application toolbar (see [Figure 14.7](#)).

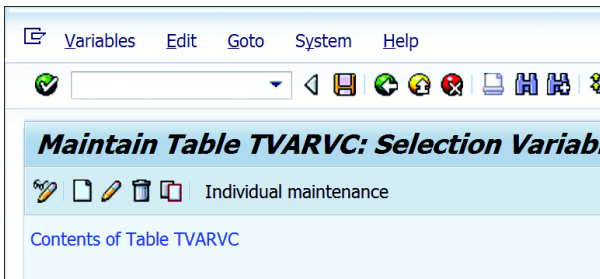


Figure 14.7 Transaction STVARV Initial Screen

3. Enter a name for the variant in the PARAMETERS or SELECT OPTIONS tab (depending on the screen field) and maintain the values (see [Figure 14.8](#)).

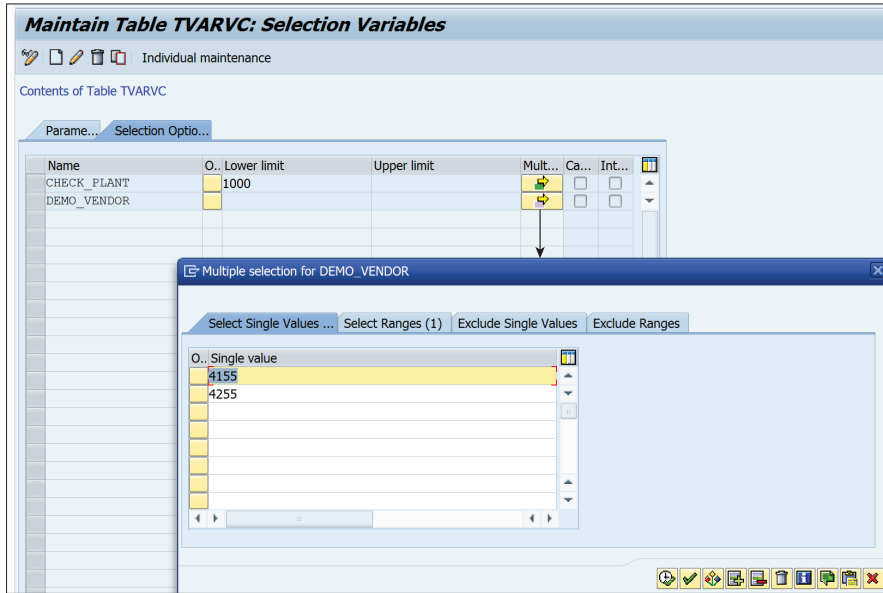


Figure 14.8 Creating Variant in Transaction STVARV

To use the variant for the selection screen field, proceed with the following steps:

1. Go the VARIANT ATTRIBUTES screen of the variant for the selection screen, select the **[F4]** help in the SELECTION VARIABLE field, and select the TABLE VARIABLE FROM TABLE TVARVC option, as shown in [Figure 14.9](#).

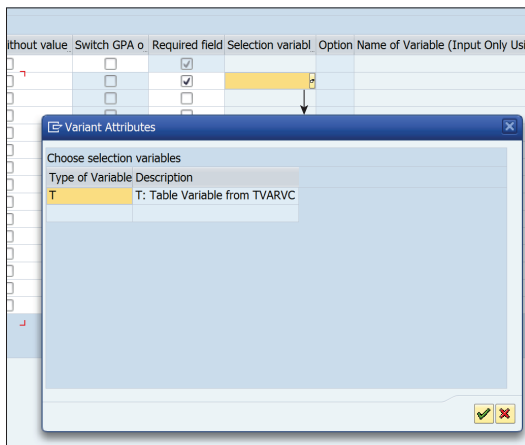


Figure 14.9 Selection Variable

2. Use the **[F4]** help in the NAME OF VARIABLE column to select the variant created in Transaction STVARV (see [Figure 14.10](#)).

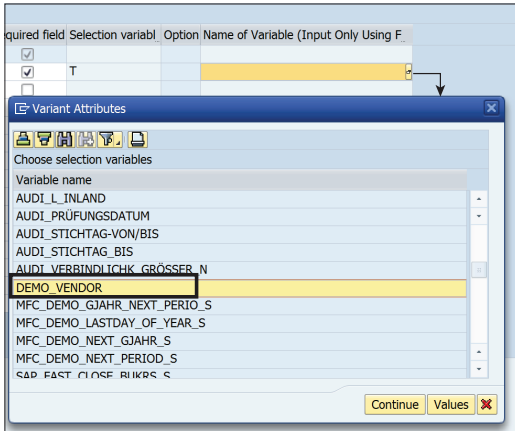


Figure 14.10 Selecting Variant

14.4.4 Dynamic Date Calculation

If the selection screen field is of TYPE D, then the field can be set to use a dynamic date calculation. For example, if you want to run a report with the current date daily, this option can automatically fill the date field with the current date.

We can use either the system date or the local date. Use the **[F4]** help in SELECTION VARIABLE for the DATE field to select either DYNAMIC DATE CALCULATION (LOCAL DATE) or DYNAMIC DATE CALCULATION (SYSTEM DATE), as shown in [Figure 14.11](#).

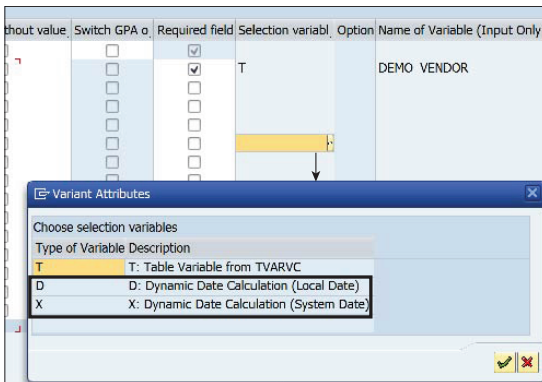


Figure 14.11 Dynamic Date Calculation

Once this option is selected, use the **F4** help in the NAME OF VARIABLE field to select from the available options, as shown in [Figure 14.12](#). As you can see, you can select from many available options, including CURRENT DATE or a specific number of days from the current date, among others.

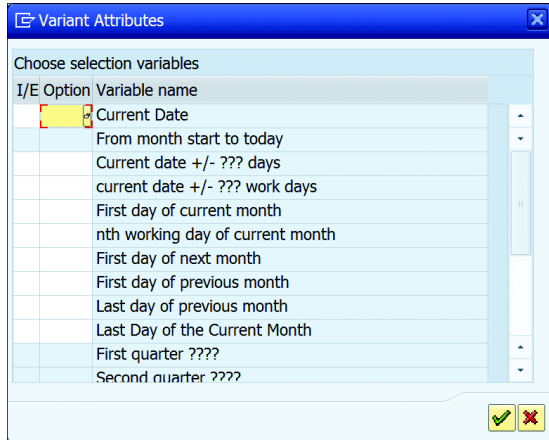


Figure 14.12 Dynamic Date Selection

14.4.5 Dynamic Time Calculation

Similar to dynamic date calculation, a field of type T can be set with dynamic time calculation. To set a dynamic time for the field, use the **F4** help in the SELECTION VARIABLE field and select the DYNAMIC TIME CALCULATION (LOCAL TIME) or DYNAMIC TIME CALCULATION (SYSTEM TIME) option, as shown in [Figure 14.13](#).

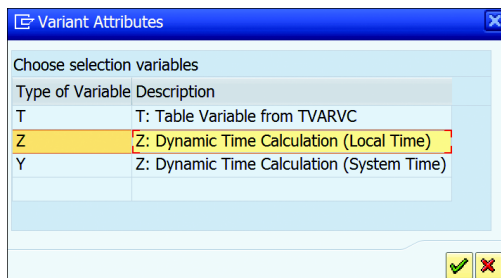


Figure 14.13 Dynamic Time Calculation

Once an option is selected, use the **F4** help in the NAME OF VARIABLE field to select from the available options, as shown in [Figure 14.14](#).

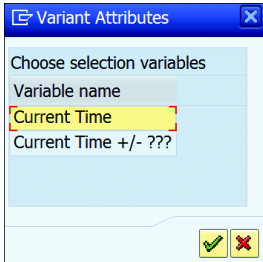


Figure 14.14 Dynamic Time Selection

14.4.6 User-Specific Variables

The USER-SPECIFIC VARIABLES option is available if the field is linked to a memory ID in the program using the MEMORY ID xxx addition (where xxx is the three-character ID) with the PARAMETERS or SELECT-OPTIONS statement. *User-specific values* that are created from the selection screen or from the user maintenance are set at runtime for the field. For example, if the selection screen field is defined in the program as PARAMETERS p_kunnr TYPE kunnr MEMORY ID dat and the parameter ID "DAT" is maintained in the user's personal settings (Transaction SU3) as shown in [Figure 14.15](#), then the value will be populated automatically for the selection screen field.

The PARAMETERS tab in the user's personal settings screen provides the option to maintain the SET/GET PARAMETER ID and PARAMETER VALUE fields. SET/GET PARAMETER ID will be linked to SAP memory. Any screen field pointing to this parameter ID in SAP memory will automatically show the parameter value as the default value within the user session.

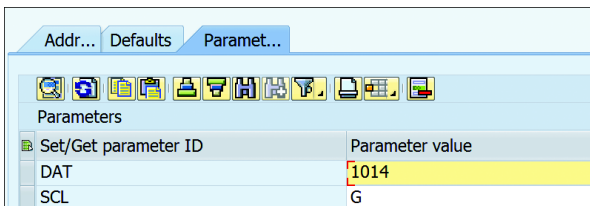


Figure 14.15 User Settings

Now that you understand selection screen variants, we'll discuss executing programs in the background.

14.5 Executing Programs in the Background

Executable programs are the only programs that can be run in the background. Programs can be executed in the background manually or automatically. For example, if a report processes huge amounts data and takes hours to finish, a user may want to run it in the background and check the report output in the spool.

If a report needs to be run at a fixed time at fixed intervals, then it can be scheduled to run as a batch job. For example, if you want to run a report every Sunday at 4:00 pm, it can be scheduled to run automatically in the background as a batch job.

To run a report in the background, proceed with the following steps:

1. Open Transaction SA38, input the program name, and click the BACKGROUND button in the application toolbar (see [Figure 14.16](#)).

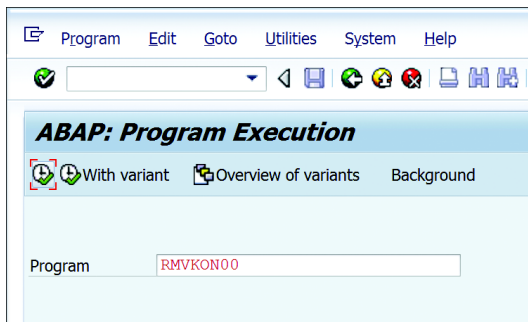


Figure 14.16 Transaction SA38 Initial Screen

2. On the EXECUTE REPORT IN BACKGROUND screen, provide the selection screen VARIANT NAME that should be used to run the report. Here, you can choose to run it immediately (via the EXECUTE IMMED. button) or schedule it to run at a later time (via the SCHEDULE button; see [Figure 14.17](#)).

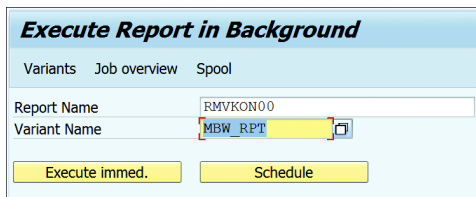


Figure 14.17 Scheduling Report Execution

- For this example, select the SCHEDULE button, which opens the SCHEDULE REPORT screen shown in [Figure 14.18](#). On this screen, you have the option to schedule the report once or periodically. Enter the job name, start date, and time. Click SCHEDULE ONCE to schedule the program to run once.

Figure 14.18 Scheduling Report for Background Execution

- If you wish to schedule the report to run periodically, click the SCHEDULE PERIODICALLY button which will open the SCHEDULE REPORT dialog box to maintain the schedule, as shown in [Figure 14.19](#).

Figure 14.19 Maintain Schedule Window

Next, let's discuss how to dynamically display or hide specific selection screen fields.

14.6 Displaying and Hiding Screen Elements Dynamically

You've seen that selection screen fields can be hidden manually by using a variant, but if you want to dynamically display or hide selection screen fields, you can handle that task programmatically in the code. For example, say you want to show certain fields to the user or hide fields based on user authorization; you can address this programmatically.

To manipulate the selection screen fields programmatically, use the AT SELECTION-SCREEN OUTPUT event. Because this event is called in the PBO event of the selection screen, we can change the field attributes before the selection screen is displayed to the user.

All the screen fields and their attributes are transferred through a special table called SCREEN. We can loop this table to access individual screen field attributes and manipulate them accordingly.

Figure 14.20 shows the components of the screen structure.

Component	Typing Method	Component Type	Data Type	Length	Deci...	Short Description
NAME	Types		CHAR	132	0	
GROUP1	Types		CHAR	3	0	
GROUP2	Types		CHAR	3	0	
GROUP3	Types		CHAR	3	0	
GROUP4	Types		CHAR	3	0	
REQUIRED	Types		CHAR	1	0	
INPUT	Types		CHAR	1	0	
OUTPUT	Types		CHAR	1	0	
INTENSIFIED	Types		CHAR	1	0	
INVISIBLE	Types		CHAR	1	0	
LENGTH	Types		RAW	1	0	
ACTIVE	Types		CHAR	1	0	
DISPLAY_3D	Types		CHAR	1	0	
VALUE_HELP	Types		CHAR	1	0	
REQUEST	Types		CHAR	1	0	
VALUES_IN_COMBO	Types		CHAR	1	0	
COLOR	Types		INT4	10	0	

Figure 14.20 Components of Screen Structure

Table 14.1 describes the components of the screen structure and provides their possible values (as shown in [Figure 14.20](#)).

Field Name	Possible Values	Meaning
NAME	Name of screen element	Name of the screen field
GROUP1 to GROUP4	id1 to id4	Modification group to apply common settings to related fields that belong to the same group
REQUIRED	0, 1, 2	Mandatory or recommended field
INPUT	0, 1	Input-ready field
OUTPUT	0, 1	Read-only/display field
INTENSIFIED	0, 1	Highlighted field
INVISIBLE	0, 1	Invisible element
LENGTH	Visible length of field	Field length
ACTIVE	0, 1	Active field
DISPLAY_3D	0, 1	Display frame
VALUE_HELP	0, 1, 2	Input help button
REQUEST	0, 1	Input exists
VALUES_IN_COMBO	0, 1	Dropdown list box

Table 14.1 Screen Structure Components

As discussed in [Chapter 12](#), we can use the `LOOP AT SCREEN` statement to access all the screen fields one at a time and modify their attributes. [Listing 14.6](#) shows an example of how to modify the screen fields using the screen structure. In this example, we're looping the screen structure to disable input for the `p_matnr` and `p_spras` fields.

```
PARAMETERS : p_kunnr TYPE kunnr,
             p_matnr TYPE matnr,
             p_spras TYPE spras.

DATA wa_screen TYPE screen.
AT SELECTION-SCREEN OUTPUT.
  LOOP AT SCREEN INTO wa_screen.
    IF wa_screen-name EQ 'P_MATNR'.
      wa_screen-input = abap_false.
    ENDIF.
    IF wa_screen-name EQ 'P_SPRAS'.
```

```

        wa_screen-input = abap_false.
    ENDF.
    MODIFY SCREEN FROM wa_screen.
    ENDL.

```

Listing 14.6 Using SCREEN Structure

Next, let's quickly discuss calling programs using selection screen.

14.7 Calling Programs via Selection Screens

For executable programs called using the `SUBMIT` statement, the selection screen fields can be used as a parameter interface to transfer data from the calling program to the called program. The called program should have a selection screen defined.

Use the `WITH` addition with the `SUBMIT` statement to map the fields between the calling program and the called program as shown:

```
SUBMIT zdemo_prg WITH p_matnr = v_matnr AND RETURN.
```

Here, the `p_matnr` selection screen field of the `zdemo_prg` program will be passed the value of `v_matnr` from the calling program.

If you wish to display the selection screen of the called program to the user, you can use the `VIA` addition as shown:

```
SUBMIT zdemo_prg WITH p_matnr = v_matnr VIA SELECTION-
SCREEN AND RETURN.
```

14.8 Summary

In this chapter, we discussed various selection screen concepts. We explored all the available selection screen events and when each of the selection screen events is called. We also explained how to create variants for a selection screen. Variants save time and effort by preventing a user from needing to enter the same input data each time he runs a report.

In addition, we looked at how to dynamically set values for selection screen fields using variants. We also showed how a variant can be used to run a program in background and discussed how selection screen fields can be manipulated

dynamically and how to use a selection screen as an interface to transfer data. At the end of this chapter, you should be comfortable working with selection screens and dynamically manipulating screen fields.

In the next chapter, we'll discuss how to develop reports using the SAP List Viewer (ALV).

SAP List Viewer (ALV) is another way to create reports in ABAP. This chapter explains how to develop reports using both procedural and object-oriented ALV libraries.

15 ALV Reports

Reports are a common development in ABAP. A typical report extracts data from the database and processes it per a user's requirements. The processed data is then presented to the user as a *report output*. Such report output typically shows data in rows and columns and also provides some basic functionality, such as sorting the output, filtering the data, summarizing the values, and so on.

As discussed in [Chapter 13](#), developing a classical report requires a developer to spend too much time making the report output presentable and provides only basic functions, such as sorting or filtering data.

To this end, SAP provides the SAP List Viewer (ALV) library, which automatically manages report output and provides additional functionalities without developers having to sweat. All the developer needs to do is supply the output data to the library. The ALV library is available for both procedural and object-oriented programming techniques. The object-oriented ALV library is available in two variants: the `CL_GUI_ALV_GRID` ABAP class and the ALV object model, the latter of which is an object-oriented encapsulation of the existing ALV tool.

In this chapter, we'll discuss how to work with the ALV library using function modules, the `CL_GUI_ALV_GRID` ABAP class, and the ALV object model. In [Section 15.1](#), we begin by discussing the standard ALV reports in the reuse library. We'll look at the four primary report displays: list, grid, block, and hierarchical.

From there, in [Section 15.2](#), we provide an overview of the interactive features that can be used in ALV reports. [Section 15.3](#) teaches how to use the Control Framework to create a custom ALV grid control. Finally, in [Section 15.4](#), we offer details on using the different displays of the ALV object model, including the table, hierarchical, and tree object displays.

15.1 Standard ALV Reports Using the Reuse Library

SAP provides various function modules in the *reuse library* that can be used to display report output. These function modules take care of formatting the output and automatically provide various additional functionalities, such as sorting, filtering, and so on. All the function module names in the reuse library start with REUSE_ALV*. The function modules in the reuse library require you to pass certain components to prepare the output; the basic components that you need to pass to the ALV function module include the following:

► **Output table**

Prepare an internal table with the output data that should be displayed as an ALV output and pass it to the ALV function module. The ALV function module reads this output table to display it as an ALV output.

► **Field catalog**

Prepare another internal table to define specifications for how the output data supplied to the ALV function module should be displayed. These specifications about the output data are called the *field catalog*. The field catalog consists of different specifications for the output fields, such as column name, length, data format, and so on. An ABAP Data Dictionary structure can also be passed to ALV instead of an internal table to describe the structure of the output data.

With the reuse library, you can create simple reports, block reports, and hierarchical reports.

The following are the different types of report output that can be displayed using the function modules from the reuse library:

► **List display**

The REUSE_ALV_LIST_DISPLAY function module is used to display a simple report as list output. The list display looks similar to the output generated using a WRITE statement. You'll learn how to use a list display in [Section 15.1.1](#).

► **Grid display**

The REUSE_ALV_GRID_DISPLAY function module is used to display a simple report as a two-dimensional grid output. The grid output is displayed using the control technology for the screen display and is part of the ALV grid control. *Controls* are software components that are installed on a local PC. You'll learn more about the Control Framework in [Section 15.3](#).

► **Block display**

List and grid displays allow you to show data from one internal table in the output, but a block display allows you to show data from multiple internal tables in the output. With a block display, you can show multiple report outputs on a single screen. The function modules used for block displays are `REUSE_ALV_BLOCK_LIST_INIT`, `REUSE_ALV_BLOCK_LIST_APPEND`, and `REUSE_ALV_BLOCK_LIST_DISPLAY`. You'll learn about block displays in [Section 15.1.2](#).

► **Hierarchical display**

This display is useful when you want to show header and line items in the output, such as a sales order header and its item details. The `REUSE_ALV_HIERSEQ_LIST_DISPLAY` function module is used for this purpose, which allows you to pass the header and item tables. The header and item tables should have a common key. You'll learn how to create a hierarchical display in [Section 15.1.3](#).

In the following subsections, we'll discuss each of these report outputs in detail.

15.1.1 List and Grid Display: Simple Reports

A simple report shows data from a single internal table as a report output; data from various database tables can be consolidated into a single internal table and passed to the ALV function module as report output. The reuse library provides two function modules—`REUSE_ALV_LIST_DISPLAY` and `REUSE_ALV_GRID_DISPLAY`—that can be used to show a simple report either as a list or as a grid, respectively.

The parameter interfaces of these two function modules are similar. The function module help documentation in Function Builder (Transaction SE37) for each function module contains detailed information about each parameter.

[Listing 15.1](#) shows sample code using a list display; it generates the output shown in [Figure 15.1](#).

```
REPORT ZDEMO_SIMPLE_REPORT.
DATA it_sflight TYPE STANDARD TABLE OF sflight.
PARAMETERS p_carrid TYPE s_carr_id.
SELECT * FROM sflight INTO TABLE it_sflight WHERE carrid EQ p_carrid.
CALL FUNCTION 'REUSE_ALV_LIST_DISPLAY'
  EXPORTING
    I_STRUCTURE_NAME      = 'SFLIGHT'
  TABLES
    t_outtab              = it_sflight
  EXCEPTIONS
    PROGRAM_ERROR        = 1
```

```

OTHERS
      = 2
      .
      IF sy-subrc <> 0.
      * Implement suitable error handling here
      ENDIF.
    
```

Listing 15.1 Simple ALV Report Using List Display

Simple Report														
ID	No.	Date	Airfare	Curr.	Plane Type	Capacity	Occupied	Booking total	Capacity	Occupied	Capacity	Occupied	Capacity	Occupied
AA	17	01.05.2013	422.94	USD	747-400	385	370	192,378.66	31	31	21	19	31	21
AA	17	29.05.2013	422.94	USD	747-400	385	372	192,860.92	31	30	21	20	31	21
AA	17	26.06.2013	422.94	USD	747-400	385	374	195,174.31	31	29	21	21	31	21
AA	17	24.07.2013	422.94	USD	747-400	385	364	189,143.24	31	29	21	20	31	21
AA	17	21.08.2013	422.94	USD	747-400	385	373	196,218.95	31	31	21	21	31	21
AA	17	18.09.2013	422.94	USD	747-400	385	373	192,230.65	31	30	21	19	31	21
AA	17	16.10.2013	422.94	USD	747-400	385	372	194,950.26	31	31	21	21	31	21
AA	17	13.11.2013	422.94	USD	747-400	385	367	188,711.76	31	28	21	20	31	21
AA	17	11.12.2013	422.94	USD	747-400	385	371	192,649.40	31	31	21	20	31	21
AA	17	08.01.2014	422.94	USD	747-400	385	48	25,604.82	31	4	21	3	31	21
AA	17	05.02.2014	422.94	USD	747-400	385	113	58,255.81	31	6	21	6	31	21
AA	17	05.03.2014	422.94	USD	747-400	385	25	11,969.21	31	1	21	1	31	21
AA	17	02.04.2014	422.94	USD	747-400	385	5	1,987.81	31	0	21	0	31	21
AA	17	30.04.2014	422.94	USD	747-400	385	54	26,442.24	31	3	21	2	31	21
AA	17	28.05.2014	422.94	USD	747-400	385	12	6,822.02	31	1	21	1	31	21
AA	64	03.05.2013	422.94	USD	A310-300	280	266	132,350.68	22	20	10	10	22	10
AA	64	31.05.2013	422.94	USD	A310-300	280	266	132,680.64	22	22	10	10	22	10
AA	64	28.06.2013	422.94	USD	A310-300	280	266	131,466.81	22	20	10	10	22	10
AA	64	26.07.2013	422.94	USD	A310-300	280	270	135,061.72	22	22	10	10	22	10
AA	64	23.08.2013	422.94	USD	A310-300	280	271	136,039.73	22	22	10	10	22	10
AA	64	20.09.2013	422.94	USD	A310-300	280	262	128,142.52	22	21	10	9	22	10
AA	64	18.10.2013	422.94	USD	A310-300	280	271	132,997.83	22	21	10	9	22	10
AA	64	15.11.2013	422.94	USD	A310-300	280	270	133,179.80	22	22	10	9	22	10
AA	64	13.12.2013	422.94	USD	A310-300	280	271	133,678.73	22	22	10	9	22	10
AA	64	10.01.2014	422.94	USD	A310-300	280	82	40,445.80	22	6	10	3	22	10
AA	64	07.02.2014	422.94	USD	A310-300	280	99	47,961.43	22	7	10	3	22	10
AA	64	07.03.2014	422.94	USD	A310-300	280	104	52,423.46	22	9	10	4	22	10
AA	64	04.04.2014	422.94	USD	A310-300	280	21	11,081.03	22	2	10	1	22	10
AA	64	02.05.2014	422.94	USD	A310-300	280	2	782.44	22	0	10	0	22	10
AA	64	30.05.2014	422.94	USD	A310-300	280	27	13,263.40	22	2	10	1	22	10

Figure 15.1 List Output Generated by Code in Listing 15.1

In Listing 15.1, we're selecting the data from table SFLIGHT for a given airline code and displaying it as report output using the list display ALV. We pass the structure name from ABAP Data Dictionary to the I_STRUCTURE_NAME formal parameter of the REUSE_ALV_LIST_DISPLAY function module, which uses this structure name to create the output layout.

The data that should be displayed in the output is selected into the internal table IT_SFLIGHT and passed to the T_OUTTAB formal parameter. The ALV function module takes care of printing all the data in internal table IT_SFLIGHT without us having to spend any additional effort on formatting the output.

The function module also loads a default SAP GUI status, which provides many buttons in the application toolbar for additional functionality, such as sorting the output, printing the output, exporting the report output, and so on, as shown in Figure 15.1.

Table 15.1 lists some of the significant parameters of the ALV function module, not all of which are used in the example. Please refer to the function module documentation in Function Builder for each ALV function module for a more exhaustive explanation.

Parameter	Description
I_CALLBACK_PROGRAM	Name of the calling program. The function module uses this information to call routines in the calling program for interactive reports.
I_CALLBACK_PF_STATUS_SET	A subroutine name of the program is passed to this parameter, in which the custom SAP GUI status can be set.
I_CALLBACK_USER_COMMAND	A subroutine name of the program is passed to this parameter, which is called when the user performs any actions on the displayed list, such as selecting a SAP GUI button or double-clicking a row.
I_CALLBACK_TOP_OF_PAGE	This parameter is specific to a grid display. A subroutine name of the program is passed to this parameter, to which simple header information can be passed for TOP-OF-PAGE.
I_GRID_TITLE	This parameter is specific to a grid display. A title for the grid can be passed to this parameter.
I_STRUCTURE_NAME	An ABAP Data Dictionary table or structure type name can be passed to this parameter, which contains the output fields.
IT_FIELDCAT	An internal table of type SLIS_T_FIELDCAT_ALV is passed to this parameter, which contains the field catalog information for the output fields.
IT_EXCLUDING	An internal table consisting of the function codes that should be excluded from the output can be passed to this parameter. All the function codes maintained in the internal table will be hidden when the list is generated.
IT_SPECIAL_GROUPS	An internal table with information about grouping fields for column selection is passed to this parameter.
IT_SORT	The sort criteria for the initial display can be passed to this parameter.

Table 15.1 ALV Function Module Parameters

Parameter	Description
IT_FILTER	The filter criteria for the initial display can be passed to this parameter.
IS_VARIANT	The variant that should be used for the initial display can be passed to this parameter.
IT_EVENTS	A table consisting of the relevant events and the form routines called for each event can be passed to this parameter. The ALV function module calls the form routine of the program for the respective event.
IS_PRINT	Print-related information can be passed to this parameter.

Table 15.1 ALV Function Module Parameters (Cont.)

The grid display ALV is similar to the list display ALV for all practical purposes, but the grid display is part of the Control Framework. The Control Framework uses JavaBeans or ActiveX controls in the presentation layer and provides certain additional features, such as grid title, TOP-OF-PAGE, removing vertical and horizontal lines, making the ALV editable, and so on.

JavaBeans and ActiveX are competing technologies that facilitate communication among components within a framework container (such as SAP GUI or a web browser). The grid display can also be used within the Control Framework, allowing multiple grid displays to be shown on a single screen, which isn't possible with a list display.

[Listing 15.2](#) shows the grid display ALV being called, which is similar to calling the list display ALV. Here, we're also passing the optional grid title, which is displayed in the report output, as shown in [Figure 15.2](#).

```
CALL FUNCTION 'REUSE_ALV_GRID_DISPLAY'
  EXPORTING
    I_STRUCTURE_NAME      = 'SFLIGHT'
    I_GRID_TITLE          = text-001
  TABLES
    t_outtab              = it_sflight
  EXCEPTIONS
    PROGRAM_ERROR         = 1
    OTHERS                 = 2.
```

Listing 15.2 Calling Grid Display ALV

Simple Report

ID	No. Flight Date	Airfare Curr.	Plane Type	Capacity	Occupied econ.	Booking total	Capacity	Occupied bus.	Capacity	Occupied 1st
AA	17 01.05.2013	422.94 USD	747-400	385	370	192,378.66	31	31	21	19
AA	17 29.05.2013	422.94 USD	747-400	385	372	192,860.92	31	30	21	20
AA	17 26.06.2013	422.94 USD	747-400	385	374	195,174.31	31	29	21	21
AA	17 24.07.2013	422.94 USD	747-400	385	364	189,143.24	31	29	21	20
AA	17 21.08.2013	422.94 USD	747-400	385	373	196,218.95	31	31	21	21
AA	17 18.09.2013	422.94 USD	747-400	385	373	192,230.65	31	30	21	19
AA	17 16.10.2013	422.94 USD	747-400	385	372	194,950.26	31	31	21	21
AA	17 13.11.2013	422.94 USD	747-400	385	367	188,711.76	31	28	21	20
AA	17 11.12.2013	422.94 USD	747-400	385	371	192,649.40	31	31	21	20
AA	17 08.01.2014	422.94 USD	747-400	385	48	25,604.82	31	4	21	3
AA	17 05.02.2014	422.94 USD	747-400	385	113	58,255.81	31	8	21	6
AA	17 05.03.2014	422.94 USD	747-400	385	25	11,969.21	31	1	21	1
AA	17 02.04.2014	422.94 USD	747-400	385	5	1,987.81	31	0	21	0
AA	17 30.04.2014	422.94 USD	747-400	385	54	26,442.24	31	3	21	2
AA	17 28.05.2014	422.94 USD	747-400	385	12	6,822.02	31	1	21	1
AA	64 03.05.2013	422.94 USD	A310-300	280	266	132,350.68	22	20	10	10

Figure 15.2 Grid Display

The ALV function module has many parameters (see [Table 15.1](#)) that can be passed to control the report output. At a bare minimum, the ALV function module requires you to supply the output layout information, such as the column positions, column names, and so on, and the actual data that should be displayed in the output.

If you wish to display all the fields of an ABAP Data Dictionary table or structure, then the table/structure name can be passed to the `I_STRUCTURE_NAME` formal parameter of the ALV function module.

In [Listing 15.1](#) and [Listing 15.2](#), we passed `SFLIGHT` as the structure name to the `I_STRUCTURE_NAME` formal parameter, because we wanted to print all the fields of table `SFLIGHT` in the output. However, you may often be printing only certain fields of a table or fields from different tables in the output. In such scenarios, you can create a field catalog and pass it to the function module using the `IT_FIELDCAT` formal parameter.

You can also create a structure in ABAP Data Dictionary consisting of the output fields and pass it to the `I_STRUCTURE_NAME` formal parameter. Using a structure name as opposed to defining a field catalog in the program helps you to maintain the output fields centrally if multiple reports are using a similar layout.

Using the field catalog has the advantage that additional attributes can be supplied for each field. The field catalog should be an internal table in which information about each output field is maintained. This internal table should be of

type `SLIS_T_FIELDCAT_ALV`, which belongs to the type group `SLIS`. The function module documentation provides information about each of the components of the `SLIS_T_FIELDCAT_ALV` structure and the valid values that can be passed.

Figure 15.3 shows some of the components of the field catalog structure. At a minimum, you should fill the `COL_POS`, `TABNAME`, `FIELDCAT`, and `SELTEXT` (long, medium, or short) components.

Dictionary: Display Type Group

Type group: `SLIS` Active
 Short description: Globale Typen für generische Listbausteine

Attribu... SourceC...

```

74 types: begin of slis_fieldcat_main0,
75     row_pos      like sy-curow, " output in row
76     col_pos      like sy-cucol, " position of the column
77     fieldname    type slis_fieldname,
78     tabname      type slis_tabname,
79     currency(5)  type c,
80     cfieldname   type slis_fieldname, " field with currency unit
81     ctabname     type slis_tabname, " and table
82     ifieldname   type slis_fieldname, " initial column
83     quantity(3) type c,
84     qfieldname   type slis_fieldname, " field with quantity unit
85     qtabname     type slis_tabname, " and table
86     round        type i, " round in write statement
87     exponent(3)  type c, " exponent for floats
88     key(1)       type c, " column with key-color
89     icon(1)      type c, " as icon
90     symbol(1)    type c, " as symbol
91     checkbox(1)  type c, " as checkbox
92     just(1)      type c, " (R)ight (L)eft (C)ent.
93     lzero(1)    type c, " leading zero
94     no_sign(1)   type c, " write no-sign
95     no_zero(1)   type c, " write no-zero
96     no_convext(1) type c,
97     edit_mask    type slis_edit_mask, "
98     emphasize(4) type c, " emphasize
99     fix_column(1) type c, " Spalte fixieren
100    do_sum(1)     type c, " sum up
101    no_out(1)     type c, " (O)blig.(X)no out
102    tech(1)       type c, " technical field
103    outputlen     like dd03p-outputlen,
104    offset         type dd03p-outputlen, " offset
105    seltext_l     like dd03p-scrtext_l, " long key word
106    seltext_m     like dd03p-scrtext_m, " middle key word
107    seltext_s     like dd03p-scrtext_s, " short key word
  
```

Figure 15.3 Field Catalog Components

Listing 15.3 shows the code for a simple report using a field catalog.


```

REPORT ZDEMO_SIMPLE_REPORT.
TYPES: BEGIN OF ty_sflight,
        CARRID TYPE S_CARR_ID,
        CONNID TYPE S_CONN_ID,
        FLDATE TYPE S_DATE,
        PRICE  TYPE S_PRICE,
        CURRENCY TYPE S_CURRCODE,
        END OF ty_sflight.

DATA : it_sflight TYPE STANDARD TABLE OF ty_sflight,
        it_fieldcat TYPE SLIS_T_FIELDCAT_ALV.

PARAMETERS p_carrid TYPE s_carr_id.

START-OF-SELECTION.
PERFORM get_data USING p_carrid CHANGING it_sflight.
END-OF-SELECTION.
PERFORM field_catalog CHANGING it_fieldcat.
PERFORM show_output.

*&-----*
*&      Form  GET_DATA
*&-----*
FORM get_data USING p_p_carrid CHANGING pt_sflight LIKE it_sflight.
SELECT carrid connid fldate price currency FROM sflight INTO TABLE
pt_sflight WHERE carrid EQ p_p_carrid.
ENDFORM.
*&-----*
*&      Form  FIELD_CATALOG
*&-----*
FORM field_catalog CHANGING pt_fieldcat LIKE it_fieldcat.
DATA lw_fieldcat LIKE LINE OF pt_fieldcat.
lw_fieldcat-col_pos = 1.
lw_fieldcat-fieldname = 'CARRID'.
lw_fieldcat-tabname = 'IT_SFLIGHT'.
lw_fieldcat-seltext_m = 'Airline Code'.
APPEND lw_fieldcat TO pt_fieldcat.
CLEAR lw_fieldcat.

lw_fieldcat-col_pos = 2.
lw_fieldcat-fieldname = 'CONNID'.
lw_fieldcat-tabname = 'IT_SFLIGHT'.
lw_fieldcat-seltext_l = 'Flight Connection Number'.
lw_fieldcat-outputlen = 25.
APPEND lw_fieldcat TO pt_fieldcat.
CLEAR lw_fieldcat.

lw_fieldcat-col_pos = 3.
lw_fieldcat-fieldname = 'FLDATE'.

```

```

lw_fieldcat-tabname = 'IT_SFLIGHT'.
lw_fieldcat-seltext_m = 'Flight Date'.
APPEND lw_fieldcat TO pt_fieldcat.
CLEAR lw_fieldcat.

lw_fieldcat-col_pos = 4.
lw_fieldcat-fieldname = 'PRICE'.
lw_fieldcat-tabname = 'IT_SFLIGHT'.
lw_fieldcat-seltext_m = 'Airfare'.
APPEND lw_fieldcat TO pt_fieldcat.
CLEAR lw_fieldcat.

lw_fieldcat-col_pos = 5.
lw_fieldcat-fieldname = 'CURRENCY'.
lw_fieldcat-tabname = 'IT_SFLIGHT'.
lw_fieldcat-seltext_m = 'Airline Currency'.
APPEND lw_fieldcat TO pt_fieldcat.
CLEAR lw_fieldcat.
ENDFORM.
*&-----*
*&      Form  SHOW_OUTPUT
*&-----*
FORM show_output.
  CALL FUNCTION 'REUSE_ALV_GRID_DISPLAY'
    EXPORTING
      I_GRID_TITLE           = 'ALV Grid Display'
      IT_FIELDCAT            = it_fieldcat
    TABLES
      t_outtab               = it_sflight
    EXCEPTIONS
      PROGRAM_ERROR          = 1
      OTHERS                  = 2
  .
  IF sy-subrc <> 0.
* Implement suitable error handling here
  ENDIF.
ENDFORM.

```

Listing 15.3 Using Field Catalog

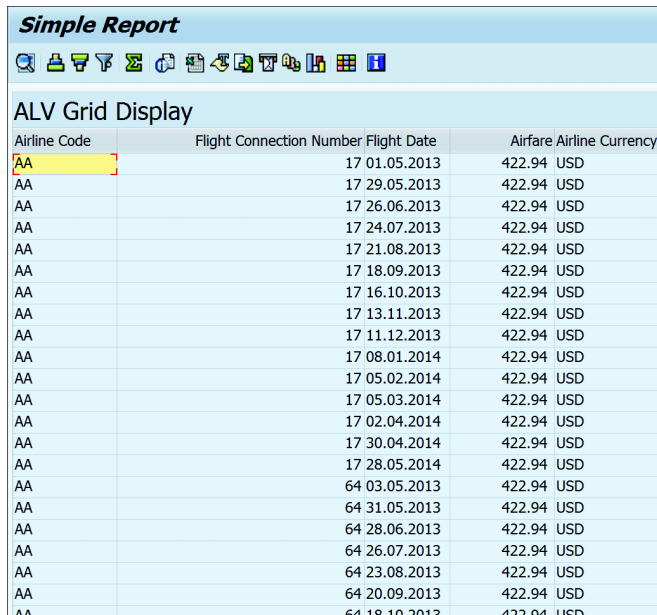
In [Listing 15.3](#), we are printing certain fields from table SFLIGHT (see [Figure 15.4](#)), as defined in the TY_SFLIGHT structure. We defined a field catalog internal table IT_FIELDCAT of type SLIS_T_FIELDCAT_ALV.SLIS_T_FIELDCAT_ALV as a table type, which belongs to the type group SLIS and has a line type of SLIS_FIELDCAT_ALV. The SLIS_FIELDCAT_ALV structure has various field catalog components.

In our program, we're passing various information about the output fields by filling the components of structure SLIS_FIELDCAT_ALV. For example, we're passing

the name of the internal table that holds the output data to the `TABNAME` component, the field name to the `FIELDNAME` component, the column position to the `COL_POS` component, and the column name to the relevant `SELTEXT` component.

In [Listing 15.3](#), the field catalog is filled inside the `FIELD_CATALOG` subroutine. The function module documentation should help you explore other components of the field catalog that can be used for formatting the output.

We can also use the `REUSE_ALV_FIELDATALOG_MERGE` function module to fill the field catalog internal table.



Airline Code	Flight Connection Number	Flight Date	Airfare	Airline	Currency
AA		17 01.05.2013	422.94	USD	
AA		17 29.05.2013	422.94	USD	
AA		17 26.06.2013	422.94	USD	
AA		17 24.07.2013	422.94	USD	
AA		17 21.08.2013	422.94	USD	
AA		17 18.09.2013	422.94	USD	
AA		17 16.10.2013	422.94	USD	
AA		17 13.11.2013	422.94	USD	
AA		17 11.12.2013	422.94	USD	
AA		17 08.01.2014	422.94	USD	
AA		17 05.02.2014	422.94	USD	
AA		17 05.03.2014	422.94	USD	
AA		17 02.04.2014	422.94	USD	
AA		17 30.04.2014	422.94	USD	
AA		17 28.05.2014	422.94	USD	
AA		64 03.05.2013	422.94	USD	
AA		64 31.05.2013	422.94	USD	
AA		64 28.06.2013	422.94	USD	
AA		64 26.07.2013	422.94	USD	
AA		64 23.08.2013	422.94	USD	
AA		64 20.09.2013	422.94	USD	
AA		64 18.10.2013	422.94	USD	

Figure 15.4 Printing Selected Table SFLIGHT Fields

The layout can be handled by passing a structure of type `SLIS_LAYOUT_ALV` to the `IS_LAYOUT` formal parameter. Using the `IS_LAYOUT` parameter, we can control the output layout, such as making the fields editable, using a stripped pattern for cells, removing horizontal or vertical lines, and so on. The function module documentation explains the significance of each component of the `SLIS_LAYOUT_ALV` structure.

[Listing 15.4](#) shows how the `IS_LAYOUT` parameter can be used to make the cells editable, as shown in [Figure 15.5](#).

Note

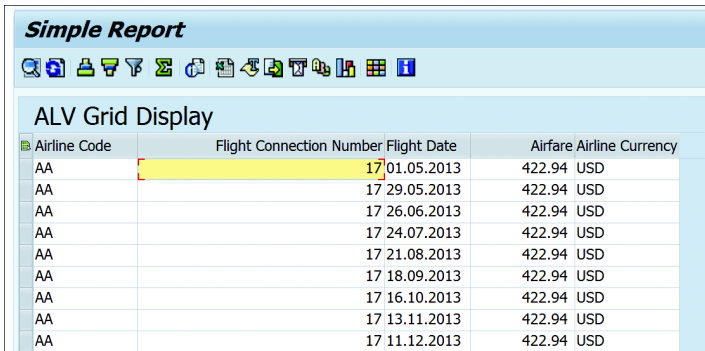
The cells can be made editable only in a grid display; this feature is not supported in a list display.

```

FORM show_output.
DATA layout TYPE SLIS_LAYOUT_ALV.
layout-edit = 'X'.
CALL FUNCTION 'REUSE_ALV_GRID_DISPLAY'
EXPORTING
  I_GRID_TITLE           = 'ALV Grid Display'
  IS_LAYOUT              = layout
  IT_FIELDCAT            = it_fieldcat
  TABLES
    t_outtab             = it_sflight
EXCEPTIONS
  PROGRAM_ERROR          = 1
  OTHERS                 = 2
.

```

Listing 15.4 Passing an ALV Layout



Airline Code	Flight Connection Number	Flight Date	Airfare Airline Currency
AA	17	01.05.2013	422.94 USD
AA		17 29.05.2013	422.94 USD
AA		17 26.06.2013	422.94 USD
AA		17 24.07.2013	422.94 USD
AA		17 21.08.2013	422.94 USD
AA		17 18.09.2013	422.94 USD
AA		17 16.10.2013	422.94 USD
AA		17 13.11.2013	422.94 USD
AA		17 11.12.2013	422.94 USD

Figure 15.5 Making Cells Editable

15.1.2 Block Display

Using a block display, we can print data from multiple internal tables in the output, with each internal table's data printed in a block. For a block display, we use three function modules:

► **REUSE_ALV_BLOCK_LIST_INIT**

This function module is used to initialize the block list and set the SAP GUI status.

► REUSE_ALV_BLOCK_LIST_APPEND

This function module is used to append the simple list in block mode; it's called each time the data from an internal table need to be appended to the output.

► REUSE_ALV_BLOCK_LIST_APPEND

This function module is used to display the block list. All the parameters of this function module are optional.

Listing 15.5 shows the sample code to work with a block display.

```
REPORT ZDEMO_BLOCK_REPORT.
DATA : it_sflight TYPE STANDARD TABLE OF sflight,
      it_spfli TYPE STANDARD TABLE OF spfli,
      it_fcat_sflight TYPE SLIS_T_FIELDCAT_ALV,
      it_fcat_spfli TYPE slis_t_fieldcat_alv,
      it_events TYPE SLIS_T_EVENT.

PARAMETERS p_carrid TYPE s_carr_id.
START-OF-SELECTION.
PERFORM get_data.
END-OF-SELECTION.
PERFORM field_catalog.
PERFORM show_output.

*&-----*
*&      Form  GET_DATA
*&-----*
FORM get_data .
SELECT * FROM sflight INTO TABLE it_sflight WHERE carrid EQ p_carrid.
SELECT * FROM spfli INTO TABLE it_spfli WHERE carrid EQ p_carrid.
ENDFORM.
*&-----*
*&      Form  FIELD_CATALOG
*&-----*
FORM field_catalog .
CALL FUNCTION 'REUSE_ALV_FIELDATALOG_MERGE'
  EXPORTING
    I_STRUCTURE_NAME      = 'SPFLI'
  CHANGING
    ct_fieldcat           = it_fcat_spfli
  EXCEPTIONS
    INCONSISTENT_INTERFACE = 1
    PROGRAM_ERROR         = 2
    OTHERS                 = 3
  .
IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.
```

```

CALL FUNCTION 'REUSE_ALV_FIELDATALOG_MERGE'
EXPORTING
  I_STRUCTURE_NAME      = 'SFLIGHT'
CHANGING
  ct_fieldcat           = it_fcat_sflight
EXCEPTIONS
  INCONSISTENT_INTERFACE = 1
  PROGRAM_ERROR          = 2
  OTHERS                 = 3
.
IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.
ENDFORM.
*&-----*
*&      Form  SHOW_OUTPUT
*&-----*
FORM show_output .
  DATA layout TYPE SLIS_LAYOUT_ALV.
  layout-zebra = 'X'.
  CALL FUNCTION 'REUSE_ALV_BLOCK_LIST_INIT'
  EXPORTING
    i_callback_program      = 'ZDEMO_BLOCK_REPORT'
.
CALL FUNCTION 'REUSE_ALV_BLOCK_LIST_APPEND'
EXPORTING
  is_layout                = layout
  it_fieldcat              = it_fcat_spfli
  i_tabname                = 'IT_SPFLI'
  it_events                = it_events
TABLES
  t_outtab                 = it_spfli
EXCEPTIONS
  PROGRAM_ERROR            = 1
  MAXIMUM_OF_APPENDS_REACHED = 2
  OTHERS                   = 3
.
IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.

CALL FUNCTION 'REUSE_ALV_BLOCK_LIST_APPEND'
EXPORTING
  is_layout                = layout
  it_fieldcat              = it_fcat_sflight
  i_tabname                = 'IT_SFLIGHT'
  it_events                = it_events
TABLES
  t_outtab                 = it_sflight
EXCEPTIONS

```

```

PROGRAM_ERROR                = 1
MAXIMUM_OF_APPENDS_REACHED  = 2
OTHERS                       = 3
.
IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.

CALL FUNCTION 'REUSE_ALV_BLOCK_LIST_DISPLAY'
EXCEPTIONS
PROGRAM_ERROR                = 1
OTHERS                       = 2
.
IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.
ENDFORM.

```

Listing 15.5 Block Display

In Listing 15.5, we're selecting data from tables SFLIGHT and SPFLI and displaying that data as blocks in the output, as shown in Figure 15.6.

ID	No.	Ctr	Depart. city	Depart	Ctr	Arrival city	Apt	FlightTime	Depart	Arrival	Distance	In	Charter	Day(s)
AA	17	US	NEW YORK	JFK	US	SAN FRANCISCO	SFO	6:01	11:00:00	14:01:00	2,572.000	MI		0
AA	64	US	SAN FRANCISCO	SFO	US	NEW YORK	JFK	5:21	09:00:00	17:21:00	2,572.000	MI		0

ID	No.	Date	Airfare	Curr.	Plane Type	Capacity	Occupied	Booking total	Capacity	Occupied	Capacity	Occupied
AA	17	01.05.2013	422.94	USD	747-400	385	370	192,378.66	31	31	21	19
AA	17	29.05.2013	422.94	USD	747-400	385	372	192,860.92	31	30	21	20
AA	17	26.06.2013	422.94	USD	747-400	385	374	195,174.31	31	29	21	21
AA	17	24.07.2013	422.94	USD	747-400	385	364	189,143.24	31	29	21	20
AA	17	21.08.2013	422.94	USD	747-400	385	373	196,218.95	31	31	21	21
AA	17	18.09.2013	422.94	USD	747-400	385	373	192,230.65	31	30	21	19
AA	17	16.10.2013	422.94	USD	747-400	385	372	194,950.26	31	31	21	21
AA	17	13.11.2013	422.94	USD	747-400	385	367	188,711.76	31	28	21	20
AA	17	11.12.2013	422.94	USD	747-400	385	371	192,649.40	31	31	21	20
AA	17	08.01.2014	422.94	USD	747-400	385	48	25,604.82	31	4	21	3
AA	17	05.02.2014	422.94	USD	747-400	385	113	58,255.81	31	8	21	6
AA	17	05.03.2014	422.94	USD	747-400	385	25	11,969.21	31	1	21	1
AA	17	02.04.2014	422.94	USD	747-400	385	5	1,987.81	31	0	21	0
AA	17	30.04.2014	422.94	USD	747-400	385	54	26,442.24	31	3	21	2
AA	17	28.05.2014	422.94	USD	747-400	385	12	6,822.02	31	1	21	1
AA	64	03.05.2013	422.94	USD	A310-300	280	266	132,350.68	22	20	10	10
AA	64	31.05.2013	422.94	USD	A310-300	280	266	132,680.64	22	22	10	10
AA	64	28.06.2013	422.94	USD	A310-300	280	266	131,466.81	22	20	10	10
AA	64	26.07.2013	422.94	USD	A310-300	280	270	135,061.72	22	22	10	10
AA	64	23.08.2013	422.94	USD	A310-300	280	271	136,038.73	22	22	10	10
AA	64	20.09.2013	422.94	USD	A310-300	280	262	128,142.52	22	21	10	9
AA	64	18.10.2013	422.94	USD	A310-300	280	271	132,997.93	22	21	10	9
AA	64	15.11.2013	422.94	USD	A310-300	280	270	133,179.80	22	22	10	9
AA	64	13.12.2013	422.94	USD	A310-300	280	271	133,678.73	22	22	10	9
AA	64	10.01.2014	422.94	USD	A310-300	280	82	40,445.80	22	6	10	3
AA	64	07.02.2014	422.94	USD	A310-300	280	99	47,961.43	22	7	10	3
AA	64	07.03.2014	422.94	USD	A310-300	280	104	52,423.46	22	9	10	4
AA	64	04.04.2014	422.94	USD	A310-300	280	21	11,081.03	22	2	10	1
AA	64	02.05.2014	422.94	USD	A310-300	280	2	782.44	22	0	10	0
AA	64	30.05.2014	422.94	USD	A310-300	280	27	13,263.40	22	2	10	1

Figure 15.6 Block Display Showing Data from Tables SPFLI and SFLIGHT

15.1.3 Hierarchical Sequential Display

A hierarchical sequential display is used when displaying related data, such as the header and item data of a document. We use the `REUSE_ALV_HIERSEQ_LIST_DISPLAY` function module to generate the hierarchical sequential display. This function module requires you to pass the header and item tables along with the key information of the tables.

The key information is used to identify the related records in the header and item tables. A field catalog is prepared with all the fields of the header and item tables and is passed to the function module.

Listing 15.6 shows the sequential report for a billing document, with certain data about the header and item details of the billing document printed in the output (see Figure 15.7).

```
REPORT ZDEMO_HIERARCHICAL_REPORT.
TYPES : BEGIN OF ty_vbrk,
         vbeln TYPE vbeln_vf,           "Document Number
         kunrg TYPE kunrg,             "Payer
         fkdat TYPE fkdat,            "Billing date
         netwr TYPE netwr,            "Net Value
       END OF ty_vbrk.
TYPES : BEGIN OF ty_vbrp,
         vbeln TYPE vbeln_vf,         " Document Number
         posnr TYPE posnr_vf,         "Billing item
         arktx TYPE arktx,           "Description of Item
         fkimg TYPE fkimg,           "Billed quantity
         vrkme TYPE vrkme,           "Sales Unit
         netwr TYPE netwr_fp,        "Net Value
         matnr TYPE matnr,           "Material Number
       END OF ty_vbrp.
DATA : it_vbrk TYPE STANDARD TABLE OF ty_vbrk,
       it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
       it_fieldcat TYPE SLIS_T_FIELDCAT_ALV,
       wa_fieldcat TYPE SLIS_FIELDCAT_ALV,
       v_vbeln TYPE vbeln_vf.
SELECT-OPTIONS s_vbeln FOR v_vbeln.
START-OF-SELECTION.
PERFORM get_data.
END-OF-SELECTION.
PERFORM field_catalog.
PERFORM show_output.
*&-----*
*&      Form  GET_DATA
*&-----*
```

```
FORM get_data .
SELECT vbeln kunrg fkdat netwr FROM vbrk INTO TABLE it_
```



```

vbrk WHERE vbeln IN s_vbeln.
SELECT vbeln posnr arktx fking vrkme netwr matnr FROM vbrp INTO TABLE
it_vbrp WHERE vbeln IN s_vbeln.
ENDFORM.
*&-----*
*&          Form  FIELD_CATALOG
*&-----*
FORM field_catalog .
  wa_fieldcat-col_pos = 1.
  wa_fieldcat-fieldname = 'VBELN'.
  wa_fieldcat-tabname = 'IT_VBRK'.
  wa_fieldcat-seltext_m = 'Document No.'.
  APPEND wa_fieldcat TO it_fieldcat.
  CLEAR wa_fieldcat.

  wa_fieldcat-col_pos = 2.
  wa_fieldcat-fieldname = 'KUNRG'.
  wa_fieldcat-tabname = 'IT_VBRK'.
  wa_fieldcat-seltext_m = 'Customer'.
  APPEND wa_fieldcat TO it_fieldcat.
  CLEAR wa_fieldcat.

  wa_fieldcat-col_pos = 3.
  wa_fieldcat-fieldname = 'FKDAT'.
  wa_fieldcat-tabname = 'IT_VBRK'.
  wa_fieldcat-seltext_m = 'Billing Date'.
  APPEND wa_fieldcat TO it_fieldcat.
  CLEAR wa_fieldcat.

  wa_fieldcat-col_pos = 4.
  wa_fieldcat-fieldname = 'NETWR'.
  wa_fieldcat-tabname = 'IT_VBRK'.
  wa_fieldcat-seltext_m = 'Net Value'.
  APPEND wa_fieldcat TO it_fieldcat.
  CLEAR wa_fieldcat.

  wa_fieldcat-col_pos = 5.
  wa_fieldcat-fieldname = 'POSNR'.
  wa_fieldcat-tabname = 'IT_VBRP'.
  wa_fieldcat-seltext_m = 'Item No'.
  APPEND wa_fieldcat TO it_fieldcat.
  CLEAR wa_fieldcat.

  wa_fieldcat-col_pos = 6.
  wa_fieldcat-fieldname = 'MATNR'.
  wa_fieldcat-tabname = 'IT_VBRP'.
  wa_fieldcat-seltext_m = 'Document No.'.
  APPEND wa_fieldcat TO it_fieldcat.
  CLEAR wa_fieldcat.

```

```

wa_fieldcat-col_pos = 7.
wa_fieldcat-fieldname = 'ARKTX'.
wa_fieldcat-tabname = 'IT_VBRP'.
wa_fieldcat-seltext_m = 'Description'.
APPEND wa_fieldcat TO it_fieldcat.
CLEAR wa_fieldcat.

wa_fieldcat-col_pos = 8.
wa_fieldcat-fieldname = 'FKIMG'.
wa_fieldcat-tabname = 'IT_VBRP'.
wa_fieldcat-seltext_m = 'Quantity'.
APPEND wa_fieldcat TO it_fieldcat.
CLEAR wa_fieldcat.

wa_fieldcat-col_pos = 9.
wa_fieldcat-fieldname = 'VRKME'.
wa_fieldcat-tabname = 'IT_VBRP'.
wa_fieldcat-seltext_m = 'UoM'.
APPEND wa_fieldcat TO it_fieldcat.
CLEAR wa_fieldcat.

wa_fieldcat-col_pos = 10.
wa_fieldcat-fieldname = 'NETWR'.
wa_fieldcat-tabname = 'IT_VBRP'.
wa_fieldcat-seltext_m = 'Net Value'.
APPEND wa_fieldcat TO it_fieldcat.
CLEAR wa_fieldcat.
ENDFORM.
*&-----*
*&      Form  SHOW_OUTPUT
*&-----*
FORM show_output .
  DATA layout TYPE SLIS_LAYOUT_ALV.
  DATA key_info TYPE SLIS_KEYINFO_ALV.
  key_info-header01 = 'VBELN'.
  key_info-item01 = 'VBELN'.
  layout-zebra = 'X'.
  CALL FUNCTION 'REUSE_ALV_HIERSEQ_LIST_DISPLAY'
    EXPORTING
      IS_LAYOUT                = layout
      IT_FIELDCAT              = it_fieldcat
      i_tabname_header         = 'IT_VBRK'
      i_tabname_item           = 'IT_VBRP'
      is_keyinfo               = key_info
    TABLES
      t_outtab_header          = it_vbrk
      t_outtab_item            = it_vbrp
    EXCEPTIONS
      PROGRAM_ERROR            = 1
      OTHERS                    = 2

```

```

IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.
ENDFORM.

```

Listing 15.6 Hierarchical Sequential List Display

In Listing 15.6, data from the billing document header table VBRK is fetched into the internal table IT_VBRK, and the item data from the database table VBRP is fetched into the internal table IT_VBRP.

A field catalog is built manually with all the fields of the header and item tables. The REUSE_ALV_HIERSEQ_LIST_DISPLAY function module is called in the program to which the field catalog is passed, along with the internal tables that contain the header and item table. A KEY_INFO structure of type SLIS_KEYINFO_ALV is used to pass the common key fields of the header and item tables.

Hierarchical List Display						
Document No.	Customer	Billing Date	Net Value			
Item No	Document No.	Description	Quantity	UoM	Net Value	
90030560	99	23.11.2001	1,000.00			
1	Erste Hilfe	1 N.N.	1.000	ST	1,000.00	
90030561	1032	23.11.2001	536042.30			
10	P-402	Pumpe Stand	76.000	ST	242090.40	
20	P-102	Pumpe PRECI	27.000	ST	82,001.70	
30	P-103	Pumpe PRECI	35.000	ST	116319.00	
40	P-104	Pumpe PRECI	28.000	ST	95,631.20	
90030594	R111	26.11.2001	6,612.30			
1	R100000	Joghurt Sta	1,724.000	ST	1,111.74	
2	R100001	Rotbandmarg	1,312.000	ST	1,459.14	
3	R100002	Salami Parm	360.000	LB	1,145.17	
4	R100004	Pizza 'Soph	134.000	ST	1,268.38	
5	R100005	Ravioli 'La	45.000	ST	42.28	
7	R100007	Orangen Beu	44.000	ST	246.17	
8	R100008	Paprikascho	112.000	LB	317.66	
9	R100009	Schraubendr	476.000	ST	1,021.76	
90030595	R112	26.11.2001	7,919.84			
1	R100000	Joghurt Sta	2,156.000	ST	1,390.32	
2	R100001	Rotbandmarg	1,312.000	ST	1,459.14	
3	R100002	Salami Parm	452.000	LB	1,437.83	
4	R100004	Pizza 'Soph	168.000	ST	1,590.21	
5	R100005	Ravioli 'La	56.000	ST	52.62	
7	R100007	Orangen Beu	56.000	ST	313.31	
8	R100008	Paprikascho	140.000	LB	397.07	
9	R100009	Schraubendr	596.000	ST	1,279.34	

Figure 15.7 Hierarchical Sequential Report

In this section, we discussed the different standard ALV report types available from the reuse library. In the next section, we'll transition into a discussion of interactive reports.

15.2 Interactive Reports

Similar to classical reports that can be made interactive in order to respond to user actions on the list output, ALV reports also support various interactive features. Unlike classical reports, in which list events are maintained in the program directly to make the report interactive, ALV requires you to pass the subroutine name of the calling program that should be called for the event.

The output screen of the ALV report belongs to the `SLVC_FULLSCREEN` function group; when the user performs any action on the screen, it triggers the PAI module of the function group, in which the user action is evaluated and the respective subroutine of the calling program is called using dynamic programming techniques.

If we dig into the ALV function group code, we'll see the dynamic subroutine call, as shown in [Figure 15.8](#).

```

Include          LSLVC_FULLSCREENF01          Active
1167 *-----*
1168 *
1169 *%      Form  USER
1170 *%-----*
1171 *
1172 * form user_command using r_ucomm type sy-ucomm
1173                          r_refresh
1174                          r_exit
1175                          rs_stable type lvc_s_stbl.
1176 data: ls_selfield type slis_selfield.
1177 *
1178 check not gt_grid-i_callback_user_command is initial.
1179 check not i_callback_program is initial.
1180
1181 perform selfield_get using ls_selfield.
1182 if r_ucomm is initial.
1183   r_ucomm = '&IC1'.
1184 endif.
1185
1186 perform (gt_grid-i_callback_user_command)
1187        in program (i_callback_program)
1188        using r_ucomm
1189              ls_selfield.
1190
1191 rs_stable-row = ls_selfield-row_stable.
1192 rs_stable-col = ls_selfield-col_stable.
1193
1194 r_refresh = ls_selfield-refresh.
1195 r_exit   = ls_selfield-exit.
1196
1197 endform.          "user command
  
```

Figure 15.8 Dynamic Subroutine Call of Calling Program from ALV

The code shown in [Figure 15.8](#) uses the information supplied to the `I_CALLBACK_PROGRAM` and `I_CALLBACK_USER_COMMAND` parameters of the ALV function modules to execute the code in the supplied subroutine.

Now, let's look at some of the interactive elements that can be added to ALV reports.

15.2.1 Loading a Custom SAP GUI Status

ALV provides standard SAP GUI buttons, which can be replaced or supplemented by a custom SAP GUI status in the program. The SAP GUI status must first be defined for the program using Menu Painter before it can be set. You can also set the SAP GUI status defined for another program. When you pass your custom SAP GUI status to ALV, it will completely replace the default ALV SAP GUI status.

If you want to use the standard features provided by the default ALV SAP GUI status and supplement it with your custom buttons, then we recommend copying the standard ALV SAP GUI status and modifying it. The `SAPLSLVC_FULLSCREEN` ALV function group contains the `STANDARD_FULLSCREEN` SAP GUI status, which can be copied to your program and modified.

Follow these steps to copy the SAP GUI status:

1. On the MENU PAINTER initial screen (Transaction SE41), enter the name of the function group, `SAPLSLVC_FULLSCREEN`, and the SAP GUI status, `STANDARD_FULLSCREEN`, that you intend to copy, as shown in [Figure 15.9](#). Click the **COPY STATUS** button in the application toolbar.

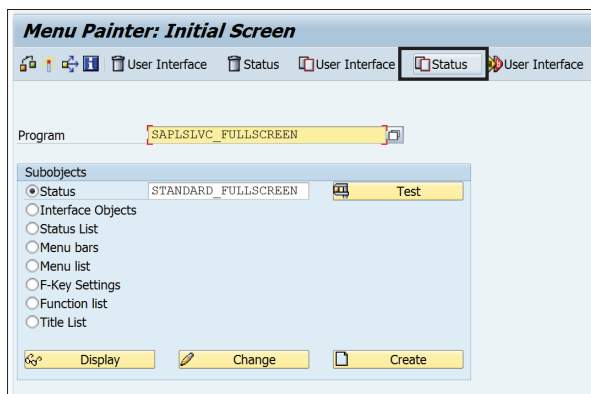


Figure 15.9 Menu Painter: Initial Screen

- In the COPY STATUS dialog box that appears, enter your program name and the SAP GUI status, as shown in [Figure 15.10](#). Click COPY on the current and subsequent windows to copy the status.

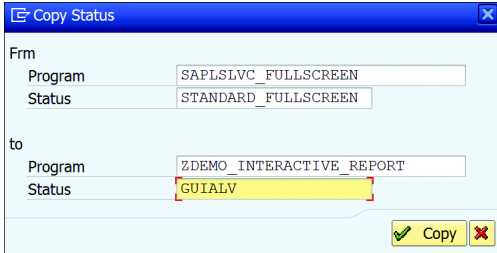


Figure 15.10 Copy Status Window

- Once the status is copied, you can edit it to add or remove any function codes. In [Figure 15.11](#), we added a PDF icon to the application toolbar with PDF as the function code. Note that all the standard ALV function codes start with &; this helps ALV to distinguish between standard function codes (which are part of ALV) and user-defined function codes. When you select a standard function code, ALV automatically handles the function code, but you need to handle custom function codes in your program.

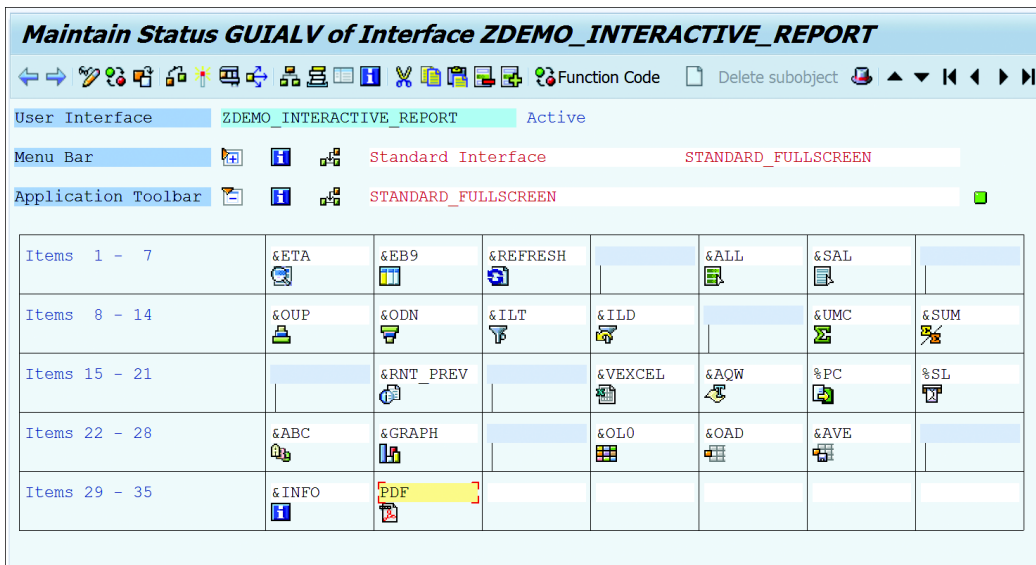


Figure 15.11 Adding Function Codes to Copied Status

To set an SAP GUI status, you need to pass the program name to the `I_CALLBACK_PROGRAM` parameter and a subroutine name to the `I_CALLBACK_PF_STATUS_SET` parameter of the ALV function module. The subroutine name that is passed to the `I_CALLBACK_PF_STATUS_SET` formal parameter should be maintained in the program with one formal parameter of type `SLIS_T_EXTAB`. This formal parameter will contain the function codes to be hidden from the UI.

Listing 15.7 shows sample code to set an SAP GUI status.

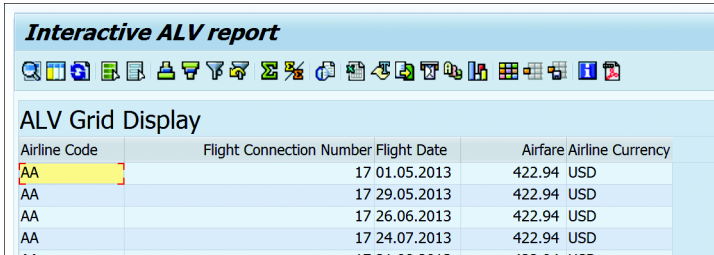
```
*&-----*
*&      Form  SHOW_OUTPUT
*&-----*
FORM show_output.
CALL FUNCTION 'REUSE_ALV_GRID_DISPLAY'
  EXPORTING
    I_CALLBACK_PROGRAM          = 'ZDEMO_INTERACTIVE_REPORT'
    I_CALLBACK_PF_STATUS_SET   = 'SET_PF_STATUS'
    I_GRID_TITLE                = 'ALV Grid Display'
    IS_LAYOUT                   = layout
    IT_FIELDCAT                 = it_fieldcat
  TABLES
    t_outtab                    = it_sflight
  EXCEPTIONS
    PROGRAM_ERROR               = 1
    OTHERS                      = 2
  .
ENDFORM.
*&-----*
*&      Form  SET_PF_STATUS
*&-----*
FORM set_pf_status USING rt_extab TYPE slis_t_extab.
SET PF-STATUS 'GUIALV'.
ENDFORM.
```

Listing 15.7 Setting Custom SAP GUI Status

In Listing 15.7, the calling program name and the subroutine name of the program are passed to the `I_CALLBACK_PROGRAM` and `I_CALLBACK_PF_STATUS_SET` parameters, respectively. The subroutine name `'SET_PF_STATUS'` is passed to the ALV function module and is maintained in the calling program with the `rt_extab` formal parameter, which is of type `SLIS_T_EXTAB`.

In this example, the `RT_EXTAB` formal parameter of the `SET_PF_STATUS` subroutine can be used to hide the function codes in the output, using the `EXCLUDING` addition with the `SET PF-STATUS` statement.

For example, we defined the PDF icon in the currently displayed SAP GUI status, as shown in [Figure 15.12](#).



Interactive ALV report

ALV Grid Display

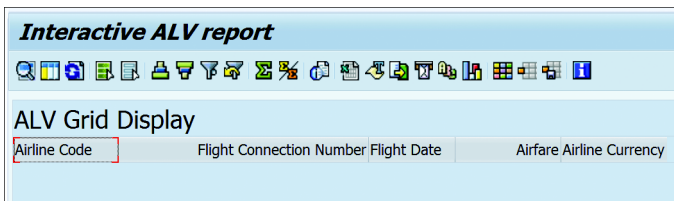
Airline Code	Flight Connection Number	Flight Date	Airfare	Airline	Currency
AA		17 01.05.2013	422.94	USD	
AA		17 29.05.2013	422.94	USD	
AA		17 26.06.2013	422.94	USD	
AA		17 24.07.2013	422.94	USD	

Figure 15.12 Showing Custom PDF Icon

However, say you want to hide this icon from the display if the output has no records; in that case, you can use the code shown in [Listing 15.8](#). This code will ensure that the PDF icon is displayed only if the output table `IT_SFLIGHT` is not initial, as shown in [Figure 15.13](#). We're clearing the internal table `RT_EXTAB` to ensure no other function codes are excluded.

```
*&-----*
*&      Form  SET_PF_STATUS
*&-----*
FORM set_pf_status USING rt_extab TYPE slis_t_extab.
DATA rw_extab LIKE LINE OF rt_extab.
IF it_sflight IS INITIAL.
  CLEAR rt_extab.
  rw_extab-fcode = 'PDF'.
  APPEND rw_extab to rt_extab.
  SET PF-STATUS 'GUIALV' EXCLUDING rt_extab.
ELSE.
  SET PF-STATUS 'GUIALV'.
ENDIF.
ENDFORM.
```

Listing 15.8 Excluding Function Codes



Interactive ALV report

ALV Grid Display

Airline Code	Flight Connection Number	Flight Date	Airfare	Airline	Currency
--------------	--------------------------	-------------	---------	---------	----------

Figure 15.13 PDF Icon Hidden from User

15.2.2 Reacting to User Actions

When we provide additional function codes using a custom SAP GUI status, we need to handle the related functionality in the program. For example, if we add a PDF button to allow a user to download the output as a PDF, then we need to write code in the program to download the output as a PDF. This requires identifying the user action on the screen and writing the respective logic.

To achieve this, we need to pass the subroutine name to the `I_CALLBACK_USER_COMMAND` parameter. This subroutine should be maintained in the program with two formal parameters, one of type `SY-UCOMM` and another of type `SLIS_SELFIELD`, as shown:

```
FORM user_command USING r_ucomm LIKE sy-ucomm
                    rs_selfield TYPE slis_selfield.
ENDFORM.
```

When this subroutine is called from the ALV function module (refer back to [Figure 15.8](#)), it will pass these parameters to the calling program. Here, `R_UCOMM` will contain the function code, and `RS_SELFIELD` will contain information about the output table, the current row, the user selected, and so on.

[Listing 15.9](#) shows sample code to identify if the user has selected the PDF icon we created previously. Multiple user actions can be evaluated within the subroutine using the `CASE` statement. The standard function code '&IC1' is passed to the `R_UCOMM` formal parameter if the user double-clicks the row. The `rs_selfield-fieldname` and `RS_SELFIELD-VALUE` fields will contain the column name in which the user double-clicked the displayed output and the value in the cell, respectively.

```
FORM user_command USING r_ucomm LIKE sy-ucomm
                    rs_selfield TYPE slis_selfield.

CASE r_ucomm.
  WHEN 'PDF'.
    *write appropriate logic.
  WHEN '&IC1'.
    *write appropriate logic.
ENDCASE.
ENDFORM.
```

Listing 15.9 Identifying User Action

15.2.3 Printing TOP-OF-PAGE

To print the TOP-OF-PAGE information, we can pass the subroutine name to the `I_CALLBACK_TOP_OF_PAGE` parameter. Within this subroutine, the `REUSE_ALV_COMMENTARY_WRITE` function module is called to print TOP-OF-PAGE, as shown in [Listing 15.10](#).

Refer to the function module documentation for the `REUSE_ALV_COMMENTARY_WRITE` function module to learn more about the parameter interface.

```
FORM top_of_page.
  DATA: lt_cw TYPE slis_t_listheader,
         lw_cw TYPE slis_listheader.
  lw_cw-tyt = 'H'.
  lw_cw-info = 'Flight Information'.
  APPEND lw_cw TO lt_cw.

  lw_cw-tyt = 'S'.
  lw_cw-key = P_CARRID.
  lw_cw-info = 'Selection criteria'.
  APPEND lw_cw TO lt_cw.
  CALL FUNCTION 'REUSE_ALV_COMMENTARY_WRITE'
    EXPORTING
      it_list_commentary      = lt_cw.
ENDFORM.
```

Listing 15.10 Printing TOP-OF-PAGE

ALV function modules support various other events, such as `USER_COMMAND`, `TOP_OF_COVERPAGE`, `END_OF_COVERPAGE`, and so on, depending on the type of list generated. These events allow you to provide additional information to ALV when the report is running. The `REUSE_ALV_EVENTS_GET` function module can be used to get the supported events for the list, and the subroutine names can be passed for different events using the `IT_EVENTS` parameter.

15.3 ALV Reports Using the Control Framework

As discussed in [Chapter 12](#), the Control Framework shows the grid display in a custom control. The `CL_GUI_ALV_GRID` control class is used to display the ALV output in a custom container. The `CL_GUI_ALV_GRID` class is the classic (old) variant used to work with the Control Framework. We recommend using the modern variant class, `CL_SALV_TABLE`, for new developments. We'll provide more details about the `CL_SALV_TABLE` class in [Section 15.4](#).

Similar to the picture control example from [Chapter 12](#), working with the ALV grid control is pretty straightforward. In this section, you'll learn how to use the ALV grid control via the `CL_GUI_ALV_GRID` control class.

To begin, we first need to define a screen for the program and create a custom container on the screen. In the PBO module of the screen, we instantiate the container object that is referenced to the `CL_GUI_CUSTOM_CONTAINER` class. The custom container name is passed to the constructor during instantiation.

Once the container object is instantiated, we instantiate the ALV grid object that's referenced to the `CL_GUI_ALV_GRID` class. The parent container object is passed to the constructor of this class.

The `SET_TABLE_FOR_FIRST_DISPLAY` method of the `CL_GUI_ALV_GRID` class is called to pass the output data that should be displayed in the grid control. The parameter interface of this method is similar to the ALV function modules we discussed in the previous section.

[Listing 15.11](#) shows the sample code to load the custom ALV grid control.

```
REPORT zdemo_alv_grid.
TABLES sflight.
DATA : it_sflight TYPE STANDARD TABLE OF sflight.
DATA: g_container TYPE REF TO cl_gui_custom_container,
      g_grid      TYPE REF TO cl_gui_alv_grid.

START-OF-SELECTION.
  CALL SCREEN 100.
*&-----*
*&      Module PBO  OUTPUT
*&-----*
MODULE pbo OUTPUT.
  IF g_grid IS NOT BOUND.
    CREATE OBJECT g_container
      EXPORTING
        container_name = 'CUSTOM'.

    CREATE OBJECT g_grid
      EXPORTING
        i_parent = g_container.
  ENDIF.

ENDMODULE.
*&-----*
*&      Module PAI  INPUT
*&-----*
```

```

MODULE pai INPUT.
  SELECT * FROM sflight INTO TABLE it_sflight WHERE carrid EQ sflight-
carrid.

  CALL METHOD g_grid->set_table_for_first_display
    EXPORTING
      i_structure_name = 'SFLIGHT'
    CHANGING
      it_outtab        = it_sflight
  EXCEPTIONS
    invalid_parameter_combination = 1
    program_error                = 2
    too_many_lines               = 3
    others                       = 4
  .

  IF sy-subrc <> 0.
* Implement suitable error handling here
  ENDIF.
ENDMODULE.

```

Listing 15.11 Working with ALV Grid Custom Control

In [Listing 15.11](#), we defined two reference objects, G_CONTAINER and G_GRID, referring to CL_GUI_CUSTOM_CONTAINER and CL_GUI_ALV_GRID, respectively. A screen with screen number 100 is defined for the program, with one input field, SFLIGHT-CARRID, and a custom container named 'CUSTOM'.

The G_CONTAINER reference object is instantiated by passing the custom container name. The reference object G_GRID is instantiated by passing the G_CONTAINER object as the parent container. We're ensuring the grid object is instantiated only once by using the IF condition IF g_grid IS NOT BOUND. The IF statement ensures that the grid object isn't instantiated each time the PBO module is called.

In the PAI module, the data from table SFLIGHT is selected into internal table IT_SFLIGHT and passed to the SET_TABLE_FOR_FIRST_DISPLAY method of the G_GRID reference object. The user can input the airline code and press to see the details in the grid, as shown in [Figure 15.14](#).

The CL_GUI_ALV_GRID class provides various methods that help you to work with the ALV grid. The class also triggers various events that can be handled by maintaining suitable event handler methods in the program. To learn more about each method of the class, visit https://help.sap.com/saphelp_erp60_sp/helpdata/en/22/a3f5ecd2fe11d2b467006094192fe3/content.htm.

Airline AA

ID	No.	Flight Date	Airfare	Curr.	Plane Type
AA	17	01.05.2013	422.94	USD	747-400
AA	17	29.05.2013	422.94	USD	747-400
AA	17	26.06.2013	422.94	USD	747-400
AA	17	24.07.2013	422.94	USD	747-400
AA	17	21.08.2013	422.94	USD	747-400
AA	17	18.09.2013	422.94	USD	747-400
AA	17	16.10.2013	422.94	USD	747-400
AA	17	13.11.2013	422.94	USD	747-400
AA	17	11.12.2013	422.94	USD	747-400
AA	17	08.01.2014	422.94	USD	747-400
AA	17	05.02.2014	422.94	USD	747-400

Figure 15.14 ALV Grid Control

15.4 ALV Object Model

With the 2004 SAP NetWeaver release, SAP introduced the ALV object model, which simplifies ALV generation via the `CL_SALV` family of classes. As you've seen in previous sections, we have to pass a lot of parameters to ALV and manually build the field catalog.

ALV function modules use procedural programming, which may not always suit you if you try to implement your programs in a completely object-oriented way. The `CL_GUI_ALV_GRID` custom controller class does provide object-oriented implementation, but you still need to create your own screens, pass various parameters, and build the field catalog manually.

The ALV object model provides a purely object-oriented implementation of ALV and simplifies creating ALV reports. The ALV object model supports three types of display: tabular display, hierarchical display, and the tree model.

The tabular display shows the output in a two-dimensional grid format; the hierarchical display shows the output in a hierarchical sequence, such as showing header and item data; and the tree model shows the output as a tree structure with parent and child nodes.

[Table 15.2](#) lists the classes of the ALV object model for each type of display.

Display	Class
Table display	CL_SALV_TABLE
Hierarchical display	CL_SALV_HIERSEQ_TABLE
Tree model	CL_SALV_TREE

Table 15.2 ALV Object Model Classes

The classes listed in [Table 15.2](#) implement a static factory method that is called to get the instance of the ALV object. The class contains many methods that can be used to manipulate ALV.

In the following subsections, we'll look at these different ALV object model display types in detail.

15.4.1 Table Display

The most common form of report output is the two-dimensional table display. The sample code in [Listing 15.12](#) shows how to create such a table display.

```
DATA : it_sflight TYPE STANDARD TABLE OF sflight,
      lo_alv TYPE REF TO cl_salv_table,
      lcx_msg TYPE REF TO cx_salv_msg.
PARAMETERS p_carrid TYPE s_carrid.

SELECT * FROM sflight INTO TABLE it_sflight WHERE carrid EQ p_carrid.

TRY.
  CALL METHOD cl_salv_table=>factory
    IMPORTING
      r_salv_table = lo_alv
    CHANGING
      t_table      = it_sflight.
  .

  CATCH cx_salv_msg INTO lcx_msg.
ENDTRY.
lo_alv->display( ).
```

Listing 15.12 ALV Table Display

The code in [Listing 15.12](#) defines a `lo_alv` reference object of type `cl_salv_table`. The factory method of the `cl_salv_table` class is called to get the ALV object reference into the `lo_alv` reference object, and the output table with the

data is passed to this factory method. Various methods of this class can be called to work with the ALV object, and we call the method `display` to display the ALV output.

As you can see, we didn't spend any time building the field catalog or passing various parameters. Calling ALV is pretty intuitive and straightforward. We can extend this basic ALV code by using various methods of the `cl_salv_table` class.

For example, the code in [Listing 15.13](#) uses the `get_functions` method to get the reference to the function object, which is used to set the default SAP GUI status.

```
TRY.
    CALL METHOD cl_salv_table=>factory
        IMPORTING
            r_salv_table = lo_alv
        CHANGING
            t_table      = it_sflight.
    .
    CATCH cx_salv_msg INTO lcx_msg.
ENDTRY.
DATA: lo_functions TYPE REF TO cl_salv_functions_list.
lo_functions = lo_alv->get_functions( ).
lo_functions->set_default( abap_true ).
lo_alv->display( ).
```

Listing 15.13 Setting Default SAP GUI Status

Similarly, the `SET_SCREEN_STATUS` method can be used to set the custom SAP GUI status as shown in [Listing 15.14](#).

```
lo_alv->set_screen_status(
REPORT      = 'ZDEMO_INTERACTIVE_REPORT'
PFSTATUS    = 'GUIALV').
```

Listing 15.14 Setting Custom GUI Status

Refer to the class documentation to explore all the available methods of the `CL_SALV_TABLE` class.

15.4.2 Hierarchical Display

The hierarchical display model can be used to replace the `REUSE_ALV_HIERSEQ_LIST_DISPLAY` function module that we discussed in [Section 15.1.3](#). Similar to the table display, using a hierarchical display is pretty straightforward. For a

hierarchical display, use the `CL_SALV_HIERSEQ_TABLE` class. This class implements a factory method, which gives the reference to the ALV object.

We pass the header and item tables and also the table binding information to this factory method. The sample code in [Listing 15.15](#) shows how to generate a hierarchical report using the ALV object model. To keep it simple, we're using the same example that we used in [Listing 15.6](#).

```
REPORT ZDEMO_HIERARCHICAL_ALV_REPORT.
TYPES : BEGIN OF ty_vbrk,
         vbeln TYPE vbeln_vf,           "Document Number
         kunrg TYPE kunrg,             "Payer
         fkdat TYPE fkdat,             "Billing date
         netwr TYPE netwr,             "Net Value
       END OF ty_vbrk.

TYPES : BEGIN OF ty_vbrp,
         vbeln TYPE vbeln_vf,           " Document Number
         posnr TYPE posnr_vf,           "Billing ITEM
         arktx TYPE arktx,             "Description of Item
         fking TYPE fking,             "Billed quantity
         vrkme TYPE vrkme,             "Sales Unit
         netwr TYPE netwr_fp,          "Net Value
         matnr TYPE matnr,             "Material Number
       END OF ty_vbrp.

DATA : it_vbrk TYPE STANDARD TABLE OF ty_vbrk,
       it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
       lo_alv TYPE REF TO CL_SALV_HIERSEQ_TABLE,
       v_vbeln TYPE vbeln_vf.
SELECT-OPTIONS s_vbeln FOR v_vbeln.

START-OF-SELECTION.
PERFORM get_data.
END-OF-SELECTION.
PERFORM show_output.

*&-----*
*&      Form  GET_DATA
*&-----*
FORM get_data .
SELECT vbeln kunrg fkdat netwr FROM vbrk INTO TABLE it_
vbrk WHERE vbeln IN s_vbeln.
SELECT vbeln posnr arktx fking vrkme netwr matnr FROM vbrp INTO TABLE i
t_vbrp WHERE vbeln IN s_vbeln.
ENDFORM.
*&-----*
```



```

*&      Form  SHOW_OUTPUT
*&-----*
FORM show_output .
  DATA: lt_key TYPE salv_t_hierseq_binding,
         lw_key LIKE LINE OF lt_key,
         lcx_data_err  TYPE REF TO cx_salv_data_error,
         lcx_not_found TYPE REF TO cx_salv_not_found.

  lw_key-master = 'VBELN'.
  lw_key-slave  = 'VBELN'.
  APPEND lw_key TO lt_key.

  TRY.
    CALL METHOD cl_salv_hierseq_table=>factory
      EXPORTING
        t_binding_level1_level2 = lt_key
      IMPORTING
        r_hierseq                = lo_alv
      CHANGING
        t_table_level1           = it_vbrk
        t_table_level2           = it_vbrp.
    CATCH cx_salv_data_error INTO lcx_data_err.
    CATCH cx_salv_not_found INTO lcx_not_found.
  ENDTRY.

  lo_alv->display( ).

ENDFORM.

```

Listing 15.15 Hierarchical Display

In [Listing 15.15](#), we're defining a LO_ALV reference object of type CL_SALV_HIERSEQ_TABLE. In the SHOW_OUTPUT subroutine, we're calling the factory method of the CL_SALV_HIERSEQ_TABLE class to get the instance of the ALV reference object. To this factory method, we're also passing the internal tables that contain the header and item data.

The factory method also requires you to pass the binding information (i.e., the common key between the header and item tables). The DISPLAY method is called to show the output. [Figure 15.15](#) shows the output generated by the sample code in [Listing 15.15](#).

Note that the default output shown in [Figure 15.15](#) doesn't show a toolbar for performing actions on the generated list. We can add the toolbar by using the GET_FUNCTIONS method to get the reference to the functions list reference object via the code in [Listing 15.16](#).

Hierarchical List Display							
Bill.Doc.	Payer	Bill. Date	Net Value				
Bill.Doc.	Item	Description		Billed Quantity	Sal	Net value	Material
90030560	99	23.11.2001	1,000.00				
90030560	1	N.N.		1.000	ST	1,000.00	Erste Hilfe
90030561	1032	23.11.2001	536,042.30				
90030561	10	Pumpe Standard IDESNORM 100-402		76.000	ST	242,090.40	P-402
90030561	20	Pumpe PRECISION 102		27.000	ST	82,001.70	P-102
90030561	30	Pumpe PRECISION 103		35.000	ST	116,319.00	P-103
90030561	40	Pumpe PRECISION 104		28.000	ST	95,631.20	P-104
90030594	R111	26.11.2001	6,612.30				
90030594	1	Joghurt Stadtliebe		1,724.000	ST	1,111.74	R100000
90030594	2	Rotbandmargarine		1,312.000	ST	1,459.14	R100001
90030594	3	Salami Parma		360.000	LB	1,145.17	R100002
90030594	4	Pizza 'Sophia L.' 3er Pack		134.000	ST	1,268.38	R100004
90030594	5	Ravioli 'La Mamma'		45.000	ST	42.28	R100005
90030594	7	Orangen Beutel a 4 Pfund		44.000	ST	246.17	R100007
90030594	8	Paprikaschote		112.000	LB	317.66	R100008
90030594	9	Schraubendreher (Kreuzschlitz)		476.000	ST	1,021.76	R100009
90030595	R112	26.11.2001	7,919.84				
90030595	1	Joghurt Stadtliebe		2,156.000	ST	1,390.32	R100000
90030595	2	Rotbandmargarine		1,312.000	ST	1,459.14	R100001
90030595	3	Salami Parma		452.000	LB	1,437.83	R100002
90030595	4	Pizza 'Sophia L.' 3er Pack		168.000	ST	1,590.21	R100004
90030595	5	Ravioli 'La Mamma'		56.000	ST	52.62	R100005
90030595	7	Orangen Beutel a 4 Pfund		56.000	ST	313.31	R100007
90030595	8	Paprikaschote		140.000	LB	397.07	R100008
90030595	9	Schraubendreher (Kreuzschlitz)		596.000	ST	1,279.34	R100009

Figure 15.15 Output Generated by Code in Listing 15.16

The default status should be set before calling the `DISPLAY` method. The default status is shown in [Figure 15.16](#).

TRY.

```
CALL METHOD cl_salv_hierseq_table=>factory
EXPORTING
  t_binding_level1_level2 = lt_key
IMPORTING
  r_hierseq                = lo_alv
CHANGING
  t_table_level1          = it_vbrk
  t_table_level2          = it_vbrp.
CATCH cx_salv_data_error INTO lcx_data_err.
CATCH cx_salv_not_found INTO lcx_not_found.
ENDTRY.
DATA: lo_functions TYPE REF TO cl_salv_functions_list.
lo_functions = lo_alv->get_functions( ).
lo_functions->set_all( abap_true ).
lo_alv->display( ).
```

Listing 15.16 Adding Default Status

Bill.Doc.	Payer	Bill. Date	Net
90030560	99	23.11.2001	1,
90030560	1 1 N.N.		

Figure 15.16 Default Status

15.4.3 Tree Object Model

With the tree object model, we can display data in a tree structure. The tree model is node-based, and each record represents a node. We create an empty tree control first and add records as nodes to the tree structure. The tree object model uses the `CL_SALV_TREE` class, and the node objects are managed by the `CL_SALV_NODES` class.

The factory method of the `CL_SALV_TREE` class is called to get the instance of the tree object. We need to pass an empty internal table to the factory method for which the line type represents the node of the tree structure. Once we have the reference to the tree object, we add the records of the internal table by manipulating the nodes.

The sample code in [Listing 15.17](#) generates a tree display, as shown in [Figure 15.17](#).

```
REPORT ZDEMO_ALV_TREE.
DATA: it_sflight TYPE STANDARD TABLE OF sflight,
      it_outtab LIKE it_sflight,
      wa_sflight TYPE sflight.

DATA: lo_tree TYPE REF TO cl_salv_tree.
DATA: nodes TYPE REF TO cl_salv_nodes,
      node TYPE REF TO cl_salv_node,
      columns type ref to cl_salv_columns,
      lo_functions TYPE REF TO cl_salv_functions_tree,
      key TYPE salv_de_node_key.
SELECT-OPTIONS s_carrid FOR wa_sflight-carrid.
* Select data
SELECT * FROM sflight INTO TABLE it_sflight WHERE carrid IN s_carrid.
* Create instance with an empty table
CALL METHOD cl_salv_tree=>factory
IMPORTING
```

```

R_SALV_TREE = lo_tree
CHANGING
t_table      = it_outtab.

* Add the nodes to the tree
NODES = lo_tree->get_nodes( ).
LOOP AT it_sflight INTO wa_sflight.
  ON CHANGE OF wa_sflight-carrid.
    CLEAR key.
    TRY.
      node = nodes->add_node( related_node = KEY
        relationship = cl_gui_column_tree=>relat_first_child
        data_row = wa_sflight ).
      KEY = node->get_key( ).
    CATCH cx_salv_msg.
    ENDTRY.
  ENDON.
  TRY.
    node = nodes->add_node( related_node = KEY
      relationship = cl_gui_column_tree=>relat_last_child
      data_row = wa_sflight ).
    CATCH cx_salv_msg.
    ENDTRY.
  ENDLLOOP.
columns = lo_tree->get_columns( ).
columns->set_optimize( abap_true ).
*Set default status
lo_functions = lo_tree->get_functions( ).
lo_functions->set_all( abap_true ).
*Display table
lo_tree->display( ).

```

Listing 15.17 Tree Display

In [Listing 15.17](#), we're selecting data from table `SFLIGHT` and displaying it as a tree structure. We've defined an internal table `IT_OUTTAB`, which is of the same type as the internal table `IT_SFLIGHT`.

The empty table `IT_OUTTAB` is passed to the factory method, and the reference to the tree object is imported to the `LO_TREE` reference object. We're getting the nodes references via the `GET_NODES` method. The records are added to the tree structure by looping the internal table `IT_SFLIGHT` and calling the `ADD_NODE` method of the `nodes` object. To this method, the record of the internal table and the node relationship is transferred, which is displayed as a node in the output.

The screenshot shows the SAP ALV Tree Display interface. The table has 14 columns: Client, Carrier, Flight No., Date, Price, Currency, Pl.type, Capacity, Occupied, Total, Capacity, Occupied, Capacity, and Occupied. The data is organized into a tree structure with expandable nodes. The first major node is '800 JL', which contains 20 sub-rows. The second major node is '800 DL', containing one row. The third major node is '800 AZ', containing one row. The fourth major node is '800 AA', containing one row. The 'Total' column for the '800 JL' node is 371,479,964,955. The 'Total' column for the '800 DL' node is 271,193,109.97. The 'Total' column for the '800 AZ' node is 213,49,348.75. The 'Total' column for the '800 AA' node is 370,192,378.66.

Client	Carrier	Flight No.	Date	Price	Currency	Pl.type	Capacity	Occupied	Total	Capacity	Occupied	Capacity	Occupied
800	JL	40804.05.2013	106,136	JPY	747-400	385	371 47,964,955	31	31	21	19		
800	JL	40804.05.2013	106,136	JPY	747-400	385	371 47,964,955	31	31	21	19		
800	JL	40801.06.2013	106,136	JPY	747-400	385	363 47,278,272	31	29	21	20		
800	JL	40829.06.2013	106,136	JPY	747-400	385	374 48,518,988	31	29	21	21		
800	JL	40827.07.2013	106,136	JPY	747-400	385	370 48,564,621	31	30	21	21		
800	JL	40824.08.2013	106,136	JPY	747-400	385	373 49,019,939	31	31	21	21		
800	JL	40821.09.2013	106,136	JPY	747-400	385	372 48,349,171	31	30	21	20		
800	JL	40819.10.2013	106,136	JPY	747-400	385	368 48,707,902	31	31	21	21		
800	JL	40816.11.2013	106,136	JPY	747-400	385	373 47,866,256	31	29	21	19		
800	JL	40814.12.2013	106,136	JPY	747-400	385	372 48,677,118	31	31	21	20		
800	JL	40811.01.2014	106,136	JPY	747-400	385	188 24,363,504	31	15	21	10		
800	JL	40808.02.2014	106,136	JPY	747-400	385	33 4,416,322	31	3	21	2		
800	JL	40808.03.2014	106,136	JPY	747-400	385	43 5,871,438	31	4	21	3		
800	JL	40805.04.2014	106,136	JPY	747-400	385	35 4,245,435	31	2	21	1		
800	JL	40803.05.2014	106,136	JPY	747-400	385	36 4,440,729	31	2	21	2		
800	JL	40831.05.2014	106,136	JPY	747-400	385	30 3,948,261	31	2	21	2		
800	DL	10627.05.2013	611.01	USD	A310-300	280	271 193,109.97	22	22	10	9		
800	AZ	55501.05.2013	185.00	EUR	A319	220	213 49,348.75	22	22	10	10		
800	AA	1701.05.2013	422.94	USD	747-400	385	370 192,378.66	31	31	21	19		

Figure 15.17 Tree Display

15.5 Summary

In this chapter, we discussed the ways to display reports using the ALV model. We started our discussion by focusing on procedural techniques, then progressed to the object-oriented implementation of ALV. Object-oriented ALV, especially the ALV object model, makes it easy to issue ALV output without having to spend any time performing mundane tasks, such as formatting the output.

We discussed different ways to make ALV reports interactively react to user actions. The ALV model makes displaying the output such an easy task that you can spend the majority of your time getting the program logic correct instead of wasting time formatting the output.

Creating reports is a frequent task for ABAP users, and we hope this chapter makes you comfortable working with reports.

So far in this book, we've approached problems in a static way; that is, we knew what problems we were trying to solve. However, sometimes we may need to address problems dynamically and may not have all the information we need until runtime. In the next chapter, we'll focus on dynamic programming so that you can be prepared for such situations.

A dynamic program returns results based on specific user input that changes and can only be determined at runtime. This chapter explains how to enable dynamic programs in ABAP.

16 Dynamic Programming

Software applications are developed to solve a problem or to help users with their day-to-day activities. Most of the time, the problem is known beforehand and the design and development of the application focuses on solving the problem at hand. For example, assume your company follows a vendor-managed inventory model in which if goods are unsold for thirty days, the company becomes billable to the supplier. At this point, the company can either pay to keep the inventory or return it to the supplier.

To accommodate this business process, you need to develop a set of applications that allow a user to capture information when goods arrive in the warehouse and a report that can show the age of the inventory. The report helps the supervisor check the goods that are reaching the end of the supplier contract period.

To develop applications for this requirement, we know all the information that we need to process in the program at design time. For example, we know which tables to update the data in, and we know which tables to query to calculate the age of the inventory for the report. All this information is available statically (beforehand) at design time.

To show the age of the inventory, we could write the code shown in [Listing 16.1](#).

```
SELECT item_id item_name in_date INTO it_inv FROM ztabinv WHERE
in_date LE p_date.
WRITE : / 'The following items exceed 30 days'.
LOOP AT it_inv INTO wa_inv.
WRITE: / wa_inv-item_id, wa_inv-item_name, wa_inv-in_date.
ENDLOOP.
```

Listing 16.1 Sample Code for Inventory Age

To write the code as shown in [Listing 16.1](#), we need to know the table name and the fields of the table statically. In the program, we statically define the data objects and access them statically by addressing the name of the data object.

However, sometimes we may not have all the information until runtime. What if the table to which we need to write the select query depends on the result of the previous statement? It isn't possible to include the table name statically in the query; to handle such scenarios, we use *dynamic programming*.

There are various ways in which you can handle dynamic programming. Sometimes, a program may have mostly static elements, but some part of it may be dynamic, or the complete program itself could be dynamic. Sometimes, we may even need to create new programs on the fly dynamically or call procedures dynamically. In this chapter, we'll explore various ways to handle all these scenarios.

ABAP supports various concepts to make ABAP programs dynamic. We'll begin by discussing field symbols in [Section 16.1](#). *Field symbols* act as pointers and can be used to point to any data object of the program dynamically, allowing you to determine dynamically which data object should be accessed at runtime.

We'll discuss *data references* in [Section 16.2](#), which, similarly to field symbols, allow you to point dynamically to existing data objects in the program. However, data references also allow you to create data objects dynamically. Another important aspect of dynamic programming is the ability to dynamically determine the type information of an existing data object or to create a data type dynamically.

In [Section 16.3](#), we will discuss Runtime Type Services (RTTS) and explore identifying and creating data types dynamically. Sometimes, business requirements demand that you determine dynamically the database table from which the data is fetched or which program/transaction should be called during the program flow or something similar. ABAP allows you to dynamically determine the database table, procedure, program, transaction, and so on by providing the token dynamically in the code.

You'll learn about *dynamic token specification* in [Section 16.4](#) and dynamic procedure calls in [Section 16.5](#). Not only can you make the ABAP code within an ABAP program dynamic, but you also can generate the complete ABAP program itself

dynamically. We'll conclude this chapter with a discussion of dynamic program generation in [Section 16.6](#).

16.1 Field Symbols

A *field symbol* is similar to a pointer, which points to an existing data object. A field symbol is not a data object; it doesn't hold any memory on its own. Instead, when a data object is assigned to a field symbol, it points to the memory location of the assigned data object.

A field symbol serves as a label for its assigned data object, which you can access as you'd access the data object itself. For example, assume we have an internal table with three fields, `FIELD1`, `FIELD2`, and `FIELD3`, and you want to modify the third row of the internal table. To do so, write the code shown in [Listing 16.2](#).

```
DATA wa LIKE LINE OF itab.
wa-field1 = 'ABC'.
wa-field2 = 'XYZ'.
wa-field3 = 123.
MODIFY itab FROM wa INDEX 3.
```

Listing 16.2 Modifying Internal Table Record Using Work Area

In [Listing 16.2](#), we're using a work area `wa` to modify the internal table record. Here, `wa` and `itab` are two separate memory locations, and we're using the `MODIFY` statement to copy the contents of `wa` to `itab`.

[Listing 16.3](#) shows how the same end can be achieved using field symbols.

```
FIELD-SYMBOLS: <wa> LIKE LINE OF itab.
READ TABLE itab ASSIGNING <wa> INDEX 3.
IF <wa> IS ASSIGNED.
<wa>-field1 = 'ABC'.
<wa>-field2 = 'XYZ'.
<wa>-field3 = 123.
ENDIF.
```

Listing 16.3 Modifying Internal Table Record Using a Field Symbol

In [Listing 16.3](#), we defined a `<wa>` field symbol as a line of `itab`. The `READ` statement assigns the third row of the internal table to the field symbol `<wa>`. At this point, the field symbol `<wa>` points to the third row of the internal table. When

you change a field, through the field symbol, the field of the third row is changed immediately, because the field symbol points to the third row directly.

We don't have to copy the contents manually using the `MODIFY` statement as we did in [Listing 16.2](#), because the `<wa>` field symbol in [Listing 16.3](#) doesn't take a separate memory space but points to the existing memory, unlike the work area `wa` in [Listing 16.2](#).

Once the data object is assigned to the field symbol, we can work with the field symbol just like the assigned data object and perform operations that can usually be performed on the assigned data object. Almost any data object can be assigned to a field symbol.

As you can see in [Listing 16.3](#), there is a way to define field symbols, assign a data object to a field symbol, check if the field symbol is assigned before accessing the field symbol, and more. Before we discuss these aspects, however, let's look at how field symbols can help make programs dynamic.

16.1.1 Using Field Symbols to Make Programs Dynamic

For our example, we'll use a report to show some basic information about a material. In this report, we'll use the ALV object model to display the output. We'll provide functionality for the user to double-click in any cell to filter the records based on the value of the cell and show the result in a new container below the displayed output.

Most of the components of this report are static: We already know the data to be displayed in the output and the user input for the report. However, filtering the records based on user action is dynamic. Let's look at how we can achieve this in a static way and then look at how to make it dynamic.

[Listing 16.4](#) shows the code for the report using the ALV object model. The code is long, but very straightforward. Take some time to go through the code and try it out in your system.

```
REPORT zdemo_salv_static.
CLASS lcl_event_handler DEFINITION DEFERRED.
DATA: custom_container TYPE REF TO cl_gui_custom_container,
      splitter          TYPE REF TO cl_gui_splitter_container,
      container_1       TYPE REF TO cl_gui_container,
      so_alv            TYPE REF TO cl_salv_table,
      mo_alv            TYPE REF TO cl_salv_table.
```

```

        gr_event_handler TYPE REF TO lcl_event_handler,
        gr_event         TYPE REF TO cl_salv_events_table.
TYPES: BEGIN OF ty_mara,
        matnr TYPE mara-matnr,
        ersda TYPE ersda,
        ernam TYPE ernam,
        laeda TYPE laeda,
        mtart TYPE mtart,
        END OF ty_mara.
DATA: it_mara TYPE STANDARD TABLE OF ty_mara,
      gv_matnr TYPE matnr.
SELECT-OPTIONS: s_matnr FOR gv_matnr.

*&-----*
*&      Class LCL_EVENT_HANDLER
*&-----*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    METHODS : on_double_click FOR EVENT double_click OF cl_salv_events_
table IMPORTING row column.
ENDCLASS.          "LCL_EVENT_HANDLER

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
CLASS lcl_event_handler IMPLEMENTATION.
  METHOD on_double_click.
    DATA : lo_sel      TYPE REF TO cl_salv_selections,
            ls_cell     TYPE salv_s_cell,
            container_2 TYPE REF TO cl_gui_container,
            lt_mara     TYPE STANDARD TABLE OF ty_mara,
            lw_mara     TYPE ty_mara.
    *Get reference to the user selection.
    lo_sel = mo_alv->get_selections( ).
    *Get the value of the cell
    ls_cell = lo_sel->get_current_cell( ).
    *Logic to filter records
    IF ls_cell-columnname EQ 'MATNR'.
      LOOP AT it_mara INTO lw_mara WHERE matnr EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDOLOOP.
    ELSEIF ls_cell-columnname EQ 'ERSDA'.
      LOOP AT it_mara INTO lw_mara WHERE ersda EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDOLOOP.
    ELSEIF ls_cell-columnname EQ 'ERNAM'.
      LOOP AT it_mara INTO lw_mara WHERE ernam EQ ls_cell-value.
        APPEND lw_mara TO lt_mara.
      ENDOLOOP.
  ENDMETHOD.

```

```

ELSEIF ls_cell-columnname EQ 'LAEDA'.
  LOOP AT it_mara INTO lw_mara WHERE laeda EQ ls_cell-value.
    APPEND lw_mara TO lt_mara.
  ENDLOOP.
ELSEIF ls_cell-columnname EQ 'MTART'.
  LOOP AT it_mara INTO lw_mara WHERE mtart EQ ls_cell-value.
    APPEND lw_mara TO lt_mara.
  ENDLOOP.
ENDIF.
CALL METHOD splitter->get_container
  EXPORTING
    row      = 2
    column   = 1
  RECEIVING
    container = container_2.

TRY.
  IF so_alv IS BOUND.
    so_alv->set_data( CHANGING t_table = lt_mara ).
  ELSE.
    CALL METHOD cl_salv_table=>factory
      EXPORTING
        list_display = if_salv_c_bool_sap=>false
        r_container   = container_2
      IMPORTING
        r_salv_table = so_alv
      CHANGING
        t_table      = lt_mara.
    ENDIF.
  CATCH cx_salv_no_new_data_allowed.
  CATCH cx_salv_msg .
ENDTRY.

so_alv->display( ).
ENDMETHOD.
ENDCLASS.
"on_double_click
"lcl_event_handler

START-OF-SELECTION.
  PERFORM fill_table.
  CALL SCREEN 100.

*&-----*
*&      Module STATUS_100  OUTPUT
*&-----*
MODULE status_100 OUTPUT.
* Create Container Object
  CREATE OBJECT custom_container
  EXPORTING

```

```

        container_name = 'CONTAINER'.
*Create splitter Object
    CREATE OBJECT splitter
        EXPORTING
            parent = custom_container
            rows   = 2
            columns = 1.
*Split the container
    CALL METHOD splitter->get_container
        EXPORTING
            row       = 1
            column    = 1
        RECEIVING
            container = container_1.
*Get ALV Object reference
    TRY.
        CALL METHOD cl_salv_table=>factory
            EXPORTING
                list_display = if_salv_c_bool_sap=>false
                r_container  = container_1
            IMPORTING
                r_salv_table = mo_alv
            CHANGING
                t_table      = it_mara.
        CATCH cx_salv_msg .
    ENDTRY.
* Set the event handler here
    gr_event = mo_alv->get_event( ).
    CREATE OBJECT gr_event_handler.
    SET HANDLER gr_event_handler->on_double_click FOR gr_event.
*Display Output
    mo_alv->display( ).
ENDMODULE.                " STATUS_100  OUTPUT
*&-----*
*&      Form  FILL_TABLE
*&-----*
FORM fill_table .
    SELECT matnr ersda ernam laeda mtart FROM mara INTO TABLE it_
    mara WHERE matnr IN s_matnr.
ENDFORM.                  " FILL_TABLE

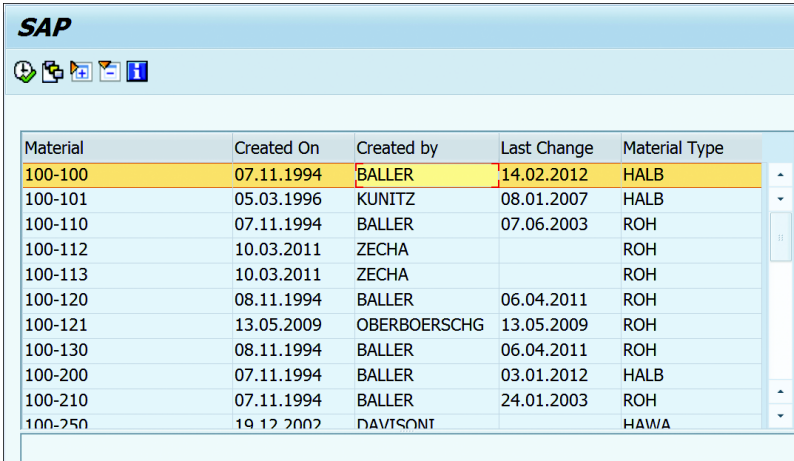
```

Listing 16.4 Report to Display Material and Filter Functionality

In [Listing 16.4](#), we're selecting certain field data from table `MARA` and displaying the output. We're using the ALV object model to generate the ALV output.

We've defined a screen 100 for our program, in which we've created a custom control. Because we want to display the filtered records on the same screen, we're using the `CL_GUI_SPLITTER_CONTAINER` class to split the container into two rows.

In the first row, we show the complete selected data, and in the second row, we display the filtered data. [Figure 16.1](#) shows the initial display of the report.



Material	Created On	Created by	Last Change	Material Type
100-100	07.11.1994	BALLER	14.02.2012	HALB
100-101	05.03.1996	KUNITZ	08.01.2007	HALB
100-110	07.11.1994	BALLER	07.06.2003	ROH
100-112	10.03.2011	ZECHA		ROH
100-113	10.03.2011	ZECHA		ROH
100-120	08.11.1994	BALLER	06.04.2011	ROH
100-121	13.05.2009	OBERBOERSCHG	13.05.2009	ROH
100-130	08.11.1994	BALLER	06.04.2011	ROH
100-200	07.11.1994	BALLER	03.01.2012	HALB
100-210	07.11.1994	BALLER	24.01.2003	ROH
100-250	19.12.2002	DAVISONI		HAWA

Figure 16.1 Initial ALV Display

The user can double-click any cell to show the filtered records in a new container below the main display. For example, if the user double-clicks the material type HALB in [Figure 16.1](#), the output would look like [Figure 16.2](#).

To display the filtered data in a new container as per the user's selection, we need to do a couple of things:

1. Identify the user action (double-click)
2. Identify the column and value of the cell to filter the records

The ALV object model raises the `ON_DOUBLE_CLICK` event when the user double-clicks the cell. In the program, we're setting the event handler and also have defined the `ON_DOUBLE_CLICK` event handler method in the `LCL_EVENT_HANDLER` local class. In this method, we're filtering the records based on the value and displaying the filtered records, as shown in [Figure 16.2](#).

When the user double-clicks, the `ON_DOUBLE_CLICK` method of the `LCL_EVENT_HANDLER` local class is called. The code in this method is where our interest lies. If you look at the code in this method, you'll notice that we used the `IF...ELSEIF...ENDIF` block to manually check the field name and accordingly filter the internal table data.

Material	Created On	Created by	Last Change	Material Type
100-100	07.11.1994	BALLER	14.02.2012	HALB
100-101	05.03.1996	KUNITZ	08.01.2007	HALB
100-110	07.11.1994	BALLER	07.06.2003	ROH
100-112	10.03.2011	ZECHA		ROH
100-113	10.03.2011	ZECHA		ROH
100-120	08.11.1994	BALLER	06.04.2011	ROH
100-121	13.05.2009	OBERBOERSCHG	13.05.2009	ROH
100-130	08.11.1994	BALLER	06.04.2011	ROH
100-200	07.11.1994	BALLER	03.01.2012	HALB
100-210	07.11.1994	BALLER	24.01.2003	ROH
100-250	19.12.2007	DAVISONI		HAWA

Material	Created On	Created by	Last Change	Material Type
100-100	07.11.1994	BALLER	14.02.2012	HALB
100-101	05.03.1996	KUNITZ	08.01.2007	HALB
100-200	07.11.1994	BALLER	03.01.2012	HALB
100-300	07.11.1994	BALLER	31.03.2015	HALB
100-301	23.01.1996	KUNITZ	08.01.2007	HALB
100-302	20.03.2002	KUNITZ	24.01.2003	HALB
100-400	07.11.1994	BALLER	09.07.2014	HALB
100-401	02.10.1996	KUNITZ	24.01.2003	HALB
100-500	07.11.1994	BALLER	19.12.2011	HALB

Figure 16.2 Filtered Output in New Container

Because there's no way for us to know statically which column the user will double-click, we had to construct the `IF...ELSEIF` block to include all the fields of the internal table. Luckily, we only had five fields in the internal table, so it wasn't that difficult.

However, what if the internal table has thirty fields or fifty fields? What if some new fields are added to the internal table as part of future enhancement? We have no other option but to update this logic to include the additional fields. Not only does listing all the fields statically in an `IF...ELSEIF` block make the code cumbersome, but hard-coding also is not a good programming practice and should be avoided as much as possible.

There must be a better way to handle this, right? Of course, this is where field symbols come to the rescue. Using field symbols, we can modify the code in the `ON_DOUBLE_CLICK` method of the `LCL_EVENT_HANDLER` local class, as shown in [Listing 16.5](#). Nothing much has changed in [Listing 16.5](#), except how we're filtering the records.

```
METHOD on_double_click.  
    FIELD-SYMBOLS: <field> TYPE any.
```

```

DATA : lo_sel      TYPE REF TO cl_salv_selections,
      ls_cell     TYPE salv_s_cell,
      container_2 TYPE REF TO cl_gui_container,
      lt_mara     TYPE STANDARD TABLE OF ty_mara,
      lw_mara     TYPE ty_mara.

lo_sel = mo_alv->get_selections( ).
ls_cell = lo_sel->get_current_cell( ).

LOOP AT it_mara INTO lw_mara.
  ASSIGN COMPONENT ls_cell-columnname OF STRUCTURE lw_mara TO
<field>.
  IF <field> IS ASSIGNED.
    IF <field> EQ ls_cell-value.
      APPEND lw_mara TO lt_mara.
    ENDIF.
  ENDIF.
ENDLOOP.
CALL METHOD splitter->get_container
  EXPORTING
    row      = 2
    column   = 1
  RECEIVING
    container = container_2.

TRY.
  IF so_alv IS BOUND.
    so_alv->set_data( CHANGING t_table = lt_mara ).
  ELSE.
    CALL METHOD cl_salv_table=>factory
      EXPORTING
        list_display = if_salv_c_bool_sap=>false
        r_container   = container_2
      IMPORTING
        r_salv_table = so_alv
      CHANGING
        t_table      = lt_mara.
  ENDIF.
CATCH cx_salv_msg .
ENDTRY.
so_alv->display( ).
ENDMETHOD.                                "on_double_click

```

Listing 16.5 Filter Functionality Using Field Symbols

In [Listing 16.5](#), we've defined a <FIELD> field symbol as a generic type (we'll discuss typing field symbols in a moment). Within the loop, we're dynamically assigning the component of the table to the field symbol. Based on which column the user double-clicks, the LS_CELL-COLUMNNAME field will have that field name.

The `ASSIGN COMPONENT OF` statement will assign that field to the `<FIELD>` field symbol.

From here, all we need to do is to check the cell value and filter the records. As you can see, now our filtering logic is dynamic. The code will filter the records irrespective of which column the user double-clicks. This code will be equally effective when new fields are added to the structure as part of future enhancement, because we're no longer hard-coding anything.

Now that you understand how field symbols can be used to make programs dynamic, let's go into further detail about how to define field symbols.

16.1.2 Defining Field Symbols

Field symbols are declared using the `FIELD-SYMBOLS` statement. The name of the field symbol should be enclosed between angled brackets (`<` and `>`). You can type the field symbol just like any other data object using the `TYPE` reference or `LIKE` reference.

You can also define a generic field symbol by typing it as `ANY`. When you define a fully typed field symbol, only a data object of the same type can be assigned to the field symbol. This is useful if you plan to access the components of a structure statically.

For example, if you want to work with the individual components of a structure as shown in [Listing 16.6](#), then the field symbol should be fully typed. Without typing the field symbol, there's no way for the compiler to know what fields it contains when the code is compiled.

```
DATA st_mara TYPE mara.
FIELD-SYMBOLS: <fs_mara> TYPE mara,
               <fs_matnr> TYPE matnr.
ASSIGN st_mara TO <fs_mara>.
<fs_mara>-matnr = '100'.
ASSIGN st_mara-matnr TO <fs_matnr>.
```

Listing 16.6 Working with Structure Components

If you plan to use the field symbol to point to any field or structure dynamically, you can define the field symbol as a generic type using the `TYPE ANY` addition, as shown in [Listing 16.7](#). However, with generic types, the components are known only at runtime, so you can't access them statically.

```
DATA st_mara TYPE mara.
FIELD-SYMBOLS: <fs_any> TYPE ANY.
ASSIGN st_mara TO <fs_any>.
* <fs_any>-matnr = '100'. "This will result in syntax error
ASSIGN st_mara-matnr TO <fs_any>.
```

Listing 16.7 Generic Type

You can also assign an internal table to the field symbol. To assign an internal table, the field symbol should be defined as a table. You can type the field symbol fully as shown:

```
FIELD-SYMBOLS <fs_mara> TYPE STANDARD TABLE OF mara.
```

You can define it as any kind of internal table of any type, just like a data object. You can also perform all the actions and use all the statements that are applicable to an internal table. For example, the syntax checker won't complain if you use a LOOP statement with a field symbol.

If you plan to assign any internal table dynamically (for which the line type is known only at runtime), you can define the field symbol as a generic table using the ANY TABLE syntax, as shown:

```
FIELD-SYMBOLS <fs_table> TYPE ANY TABLE.
```

16.1.3 Assigning a Data Object

The ASSIGN statement is used to assign the data object to a field symbol, as shown:

```
ASSIGN dobj TO <fs>.
```

The assignment operation can be performed both statically and dynamically.

If we know the data object name that we want to assign to the field symbol, we can assign the data object statically, as shown in [Listing 16.8](#).

```
DATA st_mara TYPE mara.
FIELD-SYMBOLS: <fs_mara> TYPE mara.
ASSIGN st_mara TO <fs_mara>.
```

Listing 16.8 Static Assignment of Field Symbol

Sometimes, we may know the data object that we want to assign dynamically only at runtime. For example, say that you're maintaining some change history in a table in which you're storing the field name with its old and new values. In your

program, you want to process only those fields for which a record is available in the change history table.

You can assign the data object dynamically, as shown:

```
ASSIGN (name) TO <fs>.
```

Here, the `name` variable contains the name of the data object. The variable should be enclosed within parenthesis.

[Listing 16.9](#) shows an example of dynamic assignment. Here, the data object name stored in the `FIELD_NAME` variable will be assigned to the field symbol. If the assignment is successful (i.e., the data object with that field name exists in the program), the `SY-SUBRC` system field will be set to 0.

When using dynamic assignments, it's important to check if the field symbol is assigned before accessing the field symbol. Trying to access an unassigned field symbol will result in a runtime error.

```
DATA field_name TYPE string.
FIELD-SYMBOLS <fs_field> TYPE ANY.
SELECT SINGLE ... FROM ... INTO field_name WHERE ...
ASSIGN (field_name) TO <fs_field>.
```

Listing 16.9 Dynamic Field Assignment

You can use dynamic assignment for the attributes of a class as shown:

```
ASSIGN oref->(attribute_name) TO <fs>.
```

In this assignment, the name of the attribute is derived dynamically. If you're accessing a static attribute, you can even derive the class name dynamically, as shown in [Listing 16.10](#).

```
ASSIGN (class_name)=>(attribute_name) TO <fs>.
ASSIGN (class_name)=>attribute TO <fs>.
```

Listing 16.10 Accessing Class Attributes Dynamically

The first line in [Listing 16.10](#) addresses both the class name and the attribute name dynamically, whereas the second line addresses the class name dynamically and the attribute name statically.

You can also assign the components of a structure dynamically. The syntax to assign a component of the structure dynamically is as follows:

```
ASSIGN comp OF STRUCTURE struc TO <fs>.
```

With this statement, the `comp` component of the `struc` structure is assigned to the `<fs>` field symbol. If the `comp` data object is set as `TYPE c` or `TYPE string`, then the content of the `comp` field is interpreted to be the component name. For example, in [Listing 16.11](#), the `matnr` field of the `st_mara` structure will be assigned to the field symbol.

```
DATA : st_mara TYPE mara,
      1_field TYPE string VALUE 'MATNR'.
FIELD-SYMBOLS: <fs_any> TYPE any.
ASSIGN 1_field OF STRUCTURE st_mara TO <fs_any>.
```

Listing 16.11 Assigning Structure Field

However, if the `comp` data object is of a different type and has a number, then the system will interpret it as a field position and assign that field to the field symbol.

For example, in [Listing 16.12](#), the third field, `mtart`, of the `st_mara` structure will be assigned to the field symbol because the `1_field` data object contains the value 3.

```
TYPES: BEGIN OF ty_mara,
      matnr TYPE matnr,
      ersda TYPE ersda,
      mtart TYPE mtart,
      END OF ty_mara.
DATA : st_mara TYPE ty_mara,
      1_field TYPE n VALUE 3.
FIELD-SYMBOLS: <fs_any> TYPE any.
ASSIGN 1_field OF STRUCTURE st_mara TO <fs_any>.
```

Listing 16.12 Dynamic Assignment Based on Field Position

You can use the field symbol as a work area when working with internal tables. The internal table record is assigned to the field symbol using the `ASSIGNING` addition with the `LOOP` or `READ` statement, as shown in [Listing 16.13](#).

```
READ TABLE itab ASSIGNING <fs>...
LOOP AT itab ASSIGNING <fs>..
ENDLOOP.
```

Listing 16.13 Using the `ASSIGNING` Addition

When an internal table record is assigned to a field symbol using the `LOOP` or `READ` statement, the field symbol will directly point to the assigned row of the table.

For example, in [Listing 16.14](#), we are assigning the first row of the internal table `IT_MARA` to the `<FS_MARA>` field symbol.

After this assignment, the field symbol will point to the first row of the internal table, and you can change any field in the row directly using the field symbol.

In this example, the `<FS_MARA>-MTART = 'ROH'` statement will change the value of the `MTART` field in the first row of the internal table. Unlike a work area, when using a field symbol you don't need to use the `MODIFY` statement to copy the contents of the work area back to the internal table.

```
TYPES: BEGIN OF ty_mara,
        matr TYPE matr,
        ersda TYPE ersda,
        mtart TYPE mtart,
      END OF ty_mara.
DATA : it_mara TYPE TABLE OF ty_mara.
FIELD-SYMBOLS: <fs_mara> LIKE LINE OF it_mara.
PARAMETERS p_matnr TYPE matr.
SELECT matr ersda mtart FROM mara INTO TABLE it_mara WHERE matr EQ p_
matnr.
READ TABLE it_mara ASSIGNING <fs_mara> INDEX 1.
IF <fs_mara> IS ASSIGNED.
  <fs_mara>-mtart = 'ROH'.
ENDIF.
```

Listing 16.14 Assigning Internal Table Record to Field Symbol

16.1.4 Checking if a Field Symbol is Assigned

When a data object is assigned to a field symbol, the `SY-SUBRC` system field is set to 0. If you want to check immediately after the assignment, you can check the `SY-SUBRC` field, as shown in [Listing 16.15](#).

```
ASSIGN dobj TO <fs>.
IF sy-subrc IS INITIAL.
  <fs> = 123.
ENDIF.
```

Listing 16.15 Checking the `SY-SUBRC` Field

However, in most situations, you won't need to access the field symbol immediately after the assignment but may need to access it later in the code. In such situations, checking the `SY-SUBRC` system field isn't useful, because `SY-SUBRC` is

updated by many statements and it always contains the result of the last statement that updated it.

To be sure that the field symbol is assigned before accessing it, use the `IS ASSIGNED` statement with the following syntax:

```
IF <fs> IS ASSIGNED.  
ENDIF.
```

We recommend that you always check if the field symbol is assigned before accessing it. Trying to access a field symbol that isn't assigned will result in an untreatable exception.

16.1.5 Unassigning a Field Symbol

If you want to remove the assignment of the field symbol to ensure that no data object is assigned to it, use the `UNASSIGN` statement as follows:

```
UNASSIGN <fs>.
```

After the field symbol is unassigned, it will be initialized, so it won't point to any data object. You shouldn't access the field symbol after it's unassigned unless another data object is assigned to the field symbol.

Using the `CLEAR` statement won't initialize the field symbol; it'll simply clear the contents of the data object assigned to the field symbol.

16.1.6 Casting

When a data object is assigned to a field symbol, the field symbol should be of a type compatible with the data object. However, you can perform a cast on any data type when the data object is assigned to the field symbol.

Casting allows you to treat a particular data type as another data type. This means that any area of memory can be viewed as having assumed a given type. For example, you can interpret the value of a character field as a date field when the data object is assigned to the field symbol by performing a cast.

`CASTING` is used to perform a cast, as shown:

```
ASSIGN dobj TO <fs> CASTING.
```

When you use `CASTING`, a data object of a different type can be assigned to the field symbol. Casting allows you to assign a data object to the field symbol that is not compatible with the field symbol.

[Listing 16.16](#) shows sample code in which a field of `TYPE C` is assigned to the field symbol, which is of `TYPE D`. Once the data object is assigned, the contents of the data object will be interpreted as `TYPE D`.

```
DATA text(8) TYPE c VALUE '20111201'.
FIELD-SYMBOLS <fs> TYPE sy-datum.
ASSIGN text TO <fs> CASTING.
WRITE <fs>.
```

Listing 16.16 Casting

Casting can be performed implicitly or explicitly. If the field symbol is fully typed or is typed generically using one of the built-in ABAP types (`c`, `n`, `p`, or `x`), then `CASTING` can be used with the `ASSIGN` statement, which performs an implicit casting to cast the assigned memory to the type of the field symbol.

If the field symbol is typed generically, then you can perform an explicit casting by specifying the data type in the `ASSIGN` statement as follows:

```
ASSIGN dobj TO <fs> CASTING [TYPE data_type|LIKE dobj].
```

After the `CASTING` addition, you can specify the data type or a data object. The casting will be performed as per the specified type. Your specified type should be compatible with the generic type of the field symbol, and you shouldn't perform an explicit typing for a fully typed field symbol. Casting allows you to interpret data differently per type conversions.

In this section, we discussed the use of field symbols in dynamic programming. In the next section, we'll look at using data references to point to a data object in memory.

16.2 Data References

Similar to object references, a *data reference* points to a data object in memory. Data references can be used as alternate access references to existing data objects, similarly to field symbols. However, with data references, it's possible to define

anonymous data objects. The data reference is stored in the data reference variable defined using the `TYPE REF TO` addition of the `DATA` statement.

Now, we'll explain how to define and work with data references. We'll also discuss how to define data objects dynamically using data references and assignments between data references.

16.2.1 Defining Reference Variables

The syntax to define a data reference variable is shown in [Listing 16.17](#).

```
DATA dref TYPE REF TO dtype
DATA dref LIKE REF TP dobj
DATA dref TYPE REF TO DATA
```

Listing 16.17 Defining Data Reference Variable

In [Listing 16.17](#), the first statement uses the data type reference, and the second statement uses the data object reference. The data reference variables are fully typed and can point to data objects of this type. The third statement is defined as a generic type, `DATA`, and can point to any data object.

Similar to object references, a data reference can be a static type or a dynamic type. A static type of data reference should be a fully typed or fully generic type, as shown in [Listing 16.17](#). A dynamic type should be either equal to the static type or more special than its static type (not generic).

When the data reference variable is defined, it's initial; in other words, the data reference variable doesn't point to any data object. A reference to a data object should be supplied to the data reference variable.

Once the data reference is supplied, it holds the memory location of the referenced data object and not its content. To access the contents of the referenced data object, the reference variable should be dereferenced using the `->*` dereference selector.

[Figure 16.3](#) shows the data reference variable in debug mode for better visualization. As shown, when the data reference variable `DREF` is accessed, it holds the reference to the `V_MATNR` data object. To access the contents of `V_MATNR`, we dereference the reference variable using the `->*` selector.

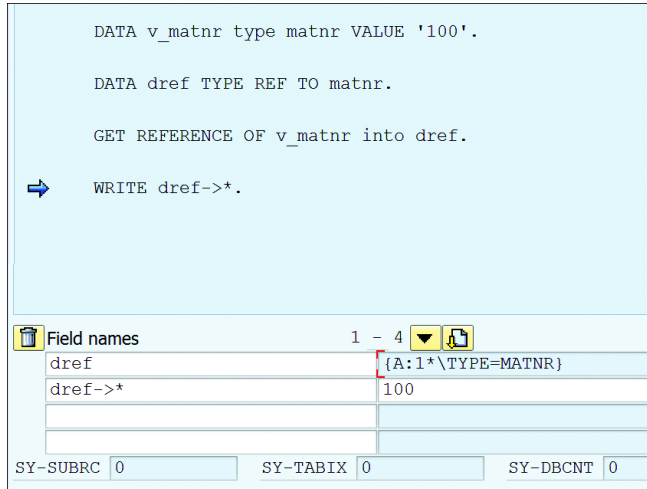


Figure 16.3 Data Reference Variable in Debug Mode

16.2.2 Getting Data References

The `GET REFERENCE OF` statement is used to obtain a reference to an existing data object. You can obtain a reference to an existing data object, an assigned field symbol, or a dereferenced data reference variable.

[Listing 16.18](#) shows how to obtain a data reference using the `GET REFERENCE OF` statement from a data object, field symbol, or dereferenced reference variable.

```

DATA v_matnr type matnr VALUE '100'.
DATA dref TYPE REF TO matnr.
DATA dref1 TYPE REF TO matnr.
FIELD-SYMBOLS <fs> TYPE matnr.
ASSIGN v_matnr TO <fs>.
*Getting reference from data object
GET REFERENCE OF v_matnr into dref.
*Getting reference from dereferenced reference variable
GET REFERENCE OF dref->* INTO dref1.
*Getting reference from field symbol
GET REFERENCE OF <fs> INTO dref1.
WRITE dref->*.

```

Listing 16.18 Getting Data Object Reference

Once the data object reference is obtained, the reference variable should be dereferenced to access the contents of the referenced data object. For example, in [Listing 16.18](#), we're dereferencing the reference variable using the `->*` dereference

selector to access the contents of the referenced data object with the `WRITE` statement.

Once the reference variable is dereferenced, the reference variable can be used to perform any actions or use any statements that are possible with the referenced data object. You can dereference the data reference variable directly only if it's fully typed. If the reference variable is generically typed, then it should be assigned to a field symbol to access the contents of the referenced data object.

[Listing 16.19](#) shows an example using generic reference variables.

```
DATA v_matnr type matnr VALUE '100'.
DATA dref TYPE REF TO data.
FIELD-SYMBOLS <fs> TYPE any.

GET REFERENCE OF v_matnr into dref.
ASSIGN dref->* TO <fs>.
* WRITE dref->*. " Results in syntax error
WRITE <fs>.
```

Listing 16.19 Dereferencing Generic Type

In [Listing 16.19](#), the `DREF` reference variable is declared as a generic type. The reference of data object `V_MATNR` is obtained using the `GET REFERENCE` statement. However, because the `DREF` reference variable is a generic type, we can't access the contents of `V_MATNR` by directly dereferencing the reference variable as we did in the `WRITE` statement in [Listing 16.18](#); in fact, dereferencing a generic type results in a syntax error.

To dereference the generic type, we assign it to a field symbol as shown in [Listing 16.19](#). If the reference variable has an empty reference, then the field symbol will not be assigned. You can check if the field symbol is assigned before accessing it to avoid an untreatable exception.

16.2.3 Anonymous Data Objects

Anonymous data objects are used when the data type is unknown statically. When we define data objects using the `DATA` statement, all the data objects are created by the runtime environment when the program starts. All these data objects can be accessed in the program statically using the name of the data object. These are called *named data objects*.

Anonymous data objects are created on demand in the program as required and can be assigned a suitable type per requirements. You use the `CREATE DATA` statement to instantiate an anonymous data object with the following syntax:

```
CREATE DATA dref TYPE|LIKE dtype.
```

The `CREATE DATA` statement creates an anonymous data object and assigns the reference of the data object to the reference variable. Anonymous data objects can't be accessed directly by their names. The data object needs to be dereferenced to access its contents.

Listing 16.20 shows an example using an anonymous data object.

```
FIELD-SYMBOLS <fs> TYPE DATA.
DATA dref TYPE REF TO DATA.
CREATE DATA dref TYPE i.
ASSIGN dref->* TO <fs>.
<fs> = 5.
```

Listing 16.20 Anonymous Data Object

The data type can be specified dynamically using a field in parenthesis—for example, `CREATE DATA dref TYPE (name)`. Here, `name` is a data object that contains the data type.

With the release of SAP NetWeaver 7.4, a new instantiation operation, `NEW`, has been introduced to instantiate both data references and object references in lieu of the `CREATE` statement. The `NEW` statement, as shown in Listing 16.21, takes its cue from other programming languages, such as Java.

```
FIELD-SYMBOLS <fs> TYPE DATA.
DATA dref TYPE REF TO data.
dref = NEW i( 5 ).
ASSIGN dref->* TO <fs>.
```

Listing 16.21 Using `NEW` Instantiation Operator

Similarly to accessing reference objects, you can access the components of a data reference using the `->` selector—for example, `DREF->COMP`. If no `TYPE` specification is provided with the `CREATE DATA` statement, the anonymous data object is created as a static type of the data reference variable.

Listing 16.22 shows example code to access components of the anonymous data object.

```

TYPES: BEGIN OF ty_mara,
        matnr TYPE matnr,
        mstart TYPE mstart,
      END OF ty_mara.
DATA dref TYPE REF TO ty_mara.
CREATE DATA dref.
dref->matnr = '100'.

```

Listing 16.22 Accessing Components of Anonymous Data Objects

You can use the `IS BOUND` statement to check if the data reference variable contains a valid reference and is dereferenceable—for example, `IF dref IS BOUND`.

16.2.4 Assignment between Reference Variables

As with object references, you can perform assignments between data references. When a data reference is assigned, the target reference variable will point to the same data object as the source reference variable.

A *reference variable* can be assigned only to another reference variable. In other words, you can't assign a reference variable to a data object or a reference object and vice versa.

Like object references, data references also support upcasting and downcasting. When assigning a reference variable, if the static type of the source reference variable is similar to or more special than the static type of the target reference variable, then an upcast is performed. You can use the `=` assignment operator or the `MOVE` statement to perform the assignment.

An upcast is performed when the target reference variable is a generic type and the source reference variable is fully typed, as shown in [Listing 16.23](#).

```

TYPES: BEGIN OF ty_mara,
        matnr TYPE matnr,
        mstart TYPE mstart,
      END OF ty_mara.
DATA : dref TYPE REF TO ty_mara, " Fully Typed
      dref1 TYPE REF TO DATA. " Generic type
CREATE DATA dref.
dref1 = dref. " Upcast

```

Listing 16.23 Upcast

If the static type of the source reference variable is more general than the static type of the target reference variable, it leads to a downcast. The assignment can

only be validated at runtime, because the type of the source reference variable will be known only at runtime.

For this assignment, the syntax checker can't validate statically and will throw a syntax error if the = assignment operator or the MOVE statement is used for the assignment.

For such assignments, we have to use the "?=" casting operator or the MOVE ?TO statement. An upcast is performed if the source reference variable is of a generic type and the target reference variable is fully typed.

Listing 16.24 shows an example of a downcast.

```
TYPES: BEGIN OF ty_mara,
        matnr TYPE matnr,
        mstart TYPE mstart,
      END OF ty_mara.
DATA : s_mara TYPE ty_mara,
      dref TYPE REF TO ty_mara, " Fully Typed
      dref1 TYPE REF TO DATA. " Generic type
CREATE DATA dref.
GET REFERENCE OF s_mara INTO dref1.
TRY,
dref ?= dref1. " Down Cast
CATCH cx_sy_move_cast_error.
ENDTRY.
```

Listing 16.24 Downcast

16.3 Runtime Type Services

ABAP *Runtime Type Services (RTTS)* is an object-oriented framework that consists of the following elements:

► **Runtime Type Information (RTTI)**

Using RTTI, we can determine the type information about data objects or instances of classes at runtime during program execution.

► **Runtime Type Creation (RTTC)**

RTTC allows us to define new data types during the program execution.

The RTTS framework implements a hierarchy of classes, as shown in [Figure 16.4](#), via which we can determine the type information of data objects and define new

data types at runtime. The instances of these type classes are known as *type objects*.

For each ABAP type, such as elementary types, reference types, complex types (structures and tables), and so on, there exists a type class.

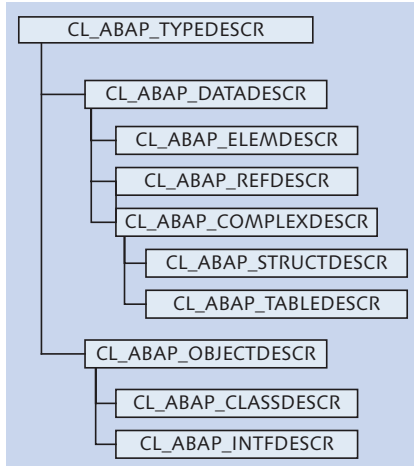


Figure 16.4 Type Class Hierarchy

For each ABAP type, there can exist exactly one type object in the program, and we can create new type objects for new types. The type objects contain attributes that describe the type. The class documentation provides information on each type class.

In the following subsections, we'll discuss RTTI and RTTC in greater detail.

16.3.1 Runtime Type Information

Often, you'll need to dynamically determine the type information of an object. For example, before using a generically typed parameter of a procedure or a generically typed anonymous data object in an arithmetic operation, you may want to check its type, length, and number of decimal places. In such cases, you can use type objects to identify the type information of a data object.

Type objects are created using the methods of the type class. The attributes of the type object contain the type information. [Listing 16.25](#) shows example code to identify the type information.

```

REPORT ZDEMO_EXAMPLES.
TYPES: ty_type TYPE p DECIMALS 2.
DATA: v_data    TYPE ty_type,
      r_typedescr TYPE REF TO cl_abap_typedescr.
START-OF-SELECTION.
r_typedescr = cl_abap_typedescr=>describe_by_data( v_data ).

WRITE: / 'Kind:',      r_typedescr->type_kind.
WRITE: / 'Length:',   r_typedescr->length.
WRITE: / 'Decimals:', r_typedescr->decimals.

```

Listing 16.25 Querying Type Information

In [Listing 16.25](#), we've defined a `R_TYPEDESCR` type object as an instance of the type class `CL_ABAP_TYPEDESCR`. The type object is instantiated when the `DESCRIBE_BY_DATA` static method of the type class is called. To this static method, we're passing the data object for which the type information is required.

Once the type object is instantiated with the type information, we can access its attributes to determine the type information of the data object. The type object also contains constants that can be used in an `IF` or `CASE` structure to compare the `TYPE_KIND` attribute.

16.3.2 Runtime Type Creation

As discussed in the previous section, type objects are created using RTTI methods of RTTS; these type objects can be used to create new types using the RTTC methods of RTTS. The RTTC methods allow you to dynamically create data types that can be used to define data objects dynamically.

For example, say you want to dynamically create a structure on the fly and use it in a `SELECT` statement. You'll only know the fields of the structure at runtime. In such a situation, you use the RTTC methods to define a data type and use to dynamically create the data object. The `GET` factory method of the `CL_ABAP_STRUCTDESCR`, `CL_ABAP_TABLEDESCR`, and `CL_ABAP_REFDESCR` classes allows you to create a type object.

The required information to create the type object is passed to the factory method of these classes. For example, the component details are passed to the factory method of the `CL_ABAP_STRUCTDESCR` class to create the type object for structure. Similarly, the structure type object is passed to the factory method of the `CL_ABAP_TABLEDESCR` class to create the table type object. [Listing 16.26](#) shows how

the RTTC methods can be used to dynamically define data types and create data objects using these types.

In [Listing 16.26](#), we've defined two selection screen fields to take the table name and the WHERE clause as input. The program logic is built to display data from any table given as the input.

To achieve this, in the GET_TABLE subroutine, we're dynamically identifying the type information of the given structure to get its components. We're building an internal table on the fly using these components and writing a dynamic SELECT statement to get the data from the requested table. Finally, we're displaying the output using ALV in the DISPLAY_TABLE subroutine.

```
PARAMETERS: p_table TYPE string,
            p_where TYPE string.

DATA:      r_typedescr   TYPE REF TO cl_abap_typedescr,
           r_structdescr TYPE REF TO cl_abap_structdescr,
           r_tabledescr  TYPE REF TO cl_abap_tabledescr,
           r_table       TYPE REF TO data,
           components    TYPE cl_abap_structdescr=>component_table,
           component     LIKE LINE OF components,
           ro_alv        TYPE REF TO cl_salv_table.

FIELD-SYMBOLS : <table> TYPE ANY TABLE.
START-OF-SELECTION.
  PERFORM get_table.
  PERFORM display_table.
*&-----*
*&      Form  GET_TABLE
*&-----*
*      text
*-----*
FORM get_table .
  r_typedescr = cl_abap_typedescr=>describe_by_name( p_table ).
  TRY.
    r_structdescr ?= r_typedescr.
    CATCH cx_sy_move_cast_error.
  ENDTRY.
  components = r_structdescr->get_components( ).
  TRY.
    r_structdescr = cl_abap_structdescr=>get( components ).
    r_tabledescr = cl_abap_tabledescr=>get( r_structdescr ).
    CATCH cx_sy_struct_creation.
    CATCH cx_sy_table_creation .
  ENDTRY.
  TRY.
```



```

        CREATE DATA r_table TYPE HANDLE r_tabledescr.
        ASSIGN r_table->* TO <table>.
    CATCH cx_sy_create_data_error.
ENDTRY.
TRY.
    SELECT *
    FROM (p_table)
    INTO CORRESPONDING FIELDS OF TABLE <table> WHERE (p_where).
    CATCH cx_sy_sql_error.
ENDTRY.
ENDFORM.
*&-----*
*&      Form  DISPLAY_TABLE
*&-----*
*      text
*-----*
FORM display_table .
TRY.
    CALL METHOD cl_salv_table=>factory
    IMPORTING
        r_salv_table = ro_alv
    CHANGING
        t_table      = <table>.
    CATCH cx_salv_msg.
ENDTRY.
ro_alv->display( ).
ENDFORM.

```

Listing 16.26 Dynamic Type Creation

In the `GET_TABLE` subroutine of [Listing 16.26](#), we're first creating the `R_TYPEDESCR` type object based on the table name entered on the selection screen. We're down-casting the `R_TYPEDESCR` type object to `R_STRUCTDESCR`. The `R_STRUCTDESCR` type object is an instance of the `CL_ABAP_STRUCTDESCR` class that contains the RTTC methods to create new type objects.

We're calling the `GET_COMPONENTS` method to get the components of the structure. Using this information, we call the `GET` factory method of the `CL_ABAP_STRUCTDESCR` class to create the structure type object. We pass this structure type object to the `GET` factory method of the `CL_ABAP_TABLEDESCR` class to create the table type object.

Once we have the table type object, we use the `TYPE HANDLE` addition with the `CREATE` statement to instantiate the data reference with the type object. This will create the `R_TABLE` data reference, the type definition of which is defined in the `R_TABLEDESCR` type object.

Although the `TYPE` or `LIKE` addition allows you to specify a data type or data object, respectively, the `TYPE HANDLE` addition with the `CREATE DATA` statement allows you to specify a type object.

After the internal table is defined dynamically, we use it in the `SELECT` statement to query the given database table and subsequently display the output. As you can see, we're handling the errors at each step; error handling is vital to dynamic programming.

If the user completes the fields as shown in [Figure 16.5](#), the code in [Listing 16.26](#) will generate the output as shown in [Figure 16.6](#). We can enhance the code further to select only specific fields of the table and create the structure with only specific fields.

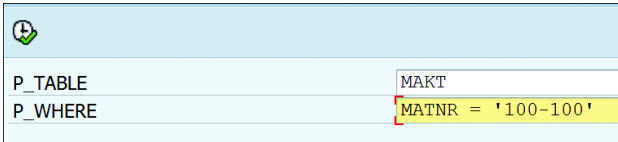


Figure 16.5 Selection Screen for Data Browser

Client	Material	Language	Description	Material description
800	100-100	ZH	外壳	外壳
800	100-100	KO	케이스	케이스
800	100-100	RO	SHOUKAT	SHOUKATFAROOQI
800	100-100	SL	Casings	CASINGS
800	100-100	HR	Casings	CASINGS
800	100-100	DE	Gehäuse	Gehäuse
800	100-100	EN	Casings	CASINGS
800	100-100	FR	Carters	Carters
800	100-100	EL	Casings	CASINGS
800	100-100	JA	ケーシング	ケーシング
800	100-100	PT	Carcaças	Carcaças
800	100-100	RU	Kopnyca	Kopnyca
800	100-100	ES	Cajas	Cajas
800	100-100	BG	Casings	CASINGS
800	100-100	SH	Casings	CASINGS

Figure 16.6 Data Browser Output

In this section, we looked at both RTTI and RTTC. In the next section, we'll look at using dynamic tokens.

16.4 Dynamic Token Specification

With many ABAP statements, you can specify individual operands as tokens dynamically by using a data object enclosed within parentheses. The content of the data object will then be used as the operand of the statement. For example, in `PERFORM (dobj)`, `dobj` (a token) usually is a character-type elementary data object.

The statement will be evaluated at runtime, and if there's a syntax error, it will result in a runtime error. For this reason, catching exceptions is very important while using dynamic tokens in a statement.

The dynamic assignment of operands can cause security risks, especially if the operands are supplied externally. For this reason, the `CL_ABAP_DYN_PRG` class should be used to check the operand thoroughly before using it in the statement. The class documentation for `CL_ABAP_DYN_PRG` provides more information about different scenarios in which the methods of the class can be used.

Some of the statements in which operands can be specified dynamically include the following:

► **SUBMIT (dobj)**

With this statement, you supply the program name dynamically. The following code example shows using the dynamic token with a `SUBMIT` statement:

```
DATA lv_progname TYPE progname.
SELECT SINGLE progname FROM zprogdire INTO lv_progname WHERE ...
SUBMIT (lv_progname).
```

► **WRITE (dobj) TO..**

With this statement, you can pass formatting options dynamically. For example, in the following code, we're passing the formatting of the `SY-DATUM` system date field to the `DATE` character field.

After the `WRITE` statement is executed, the contents of the `DATE` field will have the formatting of the `SY-DATUM` field:

```
DATA format_date TYPE c LENGTH 8 VALUE 'SY-DATUM'.
DATA date TYPE c LENGTH 10.
date = sy-datum.
WRITE (format_date) TO date.
```

► **CALL FUNCTION**

The `CALL FUNCTION` statement only supports dynamic calls. We can either pass the data object or use a literal for the function module name.

For example:

```
DATA form_name TYPE C LENGTH 30 VALUE 'ZDEMO_FM'.
CALL FUNCTION (form_name)
EXPORTING
.....
IMPORTING
.....
```

► **CALL TRANSACTION**

Similar to `CALL FUNCTION`, the `CALL TRANSACTION` statement also supports only dynamic call. The name of the transaction should be supplied dynamically using a data object or statically using a literal.

For example:

```
DATA tcode TYPE C LENGTH 30 VALUE 'MM03'.
CALL TRANSACTION (tcode).
```

► **ASSIGN (dobj)**

Dynamically assigns the field to a field symbol.

For example:

```
DATA field TYPE C LENGTH 30 VALUE 'V_MATNR'.
FIELD-SYMBOLS <fs> TYPE ANY.
ASSIGN (field) TO <fs>.
```

► **READ, MODIFY, DELETE, SORT**

You can specify internal table fields dynamically with these statements.

For example:

```
DATA field TYPE C LENGTH 30 VALUE 'MATNR'
SORT itab BY (field).
```

► **CLASSES**

You can access class attributes dynamically. For an instance attribute, you can specify the attribute name dynamically, as follows:

```
ASSIGN oref->(attribute_name) TO <fs>
```

For a static attribute, you can even specify the class name dynamically, as follows:

```
ASSIGN (class_name)=>(attribute_name) TO <fs>
```

Apart from operands, the `WHERE` clause for an Open SQL statement can also be constructed dynamically, as shown in [Listing 16.26](#). With the `WHERE` clause, the

complete statement can be constructed dynamically without any restrictions. The dynamic `WHERE` clause is supported for internal tables as well.

In this section we looked at dynamic tokens. In the next section, we'll look at calling procedures dynamically.

16.5 Dynamic Procedure Calls

Procedures can be called dynamically using *dynamic tokens*. For example, if you can determine the subroutine that needs to be called only at runtime, then you can pass the subroutine name using a dynamic token. When calling an external subroutine, the program name can also be passed dynamically as follows:

```
PERFORM (l_name) IN PROGRAM (l_program) IF FOUND.
```

When calling a subroutine dynamically, the system will generate a runtime error if the specified subroutine or program is not found. To avoid the runtime error, you should use the `IF FOUND` addition.

Methods can also be called dynamically by passing the class name and the method name with a dynamic token. In fact, the parameters of the method can also be supplied dynamically using the `PARAMETER-TABLE` addition as follows:

```
CALL METHOD class_name=>method_name PARAMETER-TABLE para.
```

In the preceding statement, we're passing the parameters to the method dynamically using the `PARAMETER-TABLE` addition. Here, an internal table of type `ABAP_PARBIND_TAB` is expected. The line type of `ABAP_PARBIND_TAB` consists of three fields: `NAME`, `KIND`, and `VALUE`. The `NAME` field should contain the formal parameter name.

The `KIND` field should be filled with the parameter type (whether exporting, importing, returning, etc., from a caller's point of view). The `VALUE` field should contain the reference of the actual parameter.

We can use the constants of the `cl_abap_objectdescr` class to fill the `KIND` field. As with `PARAMETER-TABLE`, non-class-based exceptions can be handled using the `EXCEPTION-TABLE` addition.

As with methods, the parameters for a function module can also be supplied dynamically using `PARAMETER-TABLE`. The only difference is that `PARAMETER-TABLE` when used for the function module is of type `ABAP_FUNC_PARBIND_TAB`.

Listing 16.27 shows example code using a parameter table to pass parameters dynamically.

```

DATA: fm_name TYPE STRING,
      filename TYPE STRING,
      filetype TYPE C LENGTH 80,
      text_tab LIKE STANDARD TABLE OF LINE,
      fleng     TYPE I,
      ptab TYPE abap_func_parmbind_tab,
      ptab_line LIKE LINE OF ptab,
      etab TYPE abap_func_excpbind_tab,
      etab_line LIKE LINE OF etab.

fm_name = 'GUI_DOWNLOAD'.
filename = 'C:\text.txt'.
filetype = 'ASC'.

ptab_line-NAME = 'FILENAME'.
ptab_line-KIND = abap_func_exporting.
GET REFERENCE OF filename INTO ptab_line-VALUE.
INSERT ptab_line INTO TABLE ptab.

ptab_line-NAME = 'FILETYPE'.
ptab_line-KIND = abap_func_exporting.
GET REFERENCE OF filetype INTO ptab_line-VALUE.
INSERT ptab_line INTO TABLE ptab.

ptab_line-NAME = 'DATA_TAB'.
ptab_line-KIND = abap_func_tables.
GET REFERENCE OF text_tab INTO ptab_line-VALUE.
INSERT ptab_line INTO TABLE ptab.

ptab_line-NAME = 'FILELENGTH'.
ptab_line-KIND = abap_func_importing.
GET REFERENCE OF fleng INTO ptab_line-VALUE.
INSERT ptab_line INTO TABLE ptab.

...

etab_line-NAME = 'OTHERS'.
etab_line-VALUE = 10.
INSERT etab_line INTO TABLE etab.

CALL FUNCTION fm_name
PARAMETER-TABLE

```

```
ptab
EXCEPTION-TABLE
etab.
```

Listing 16.27 Parameter Table

In [Listing 16.27](#), we defined the parameter table `PTAB` as type `ABAP_FUNC_PARM-BIND_TAB`. We're filing the parameter of the function module `GUI_DOWNLOAD` dynamically in the internal table `PTAB`.

The function module is called dynamically, and the parameter table is passed using the `PARAMETER-TABLE` addition. We're also passing the exceptions table `ETAB`, which is defined as type `ABAP_FUNC_EXCPBIND_TAB`.

This section looked at the process of calling procedures dynamically. In the next section, we look at a more advanced feature of ABAP: dynamic program generation.

16.6 Dynamic Program Generation

In addition to all the dynamic programming concepts discussed in this chapter, which can be used within an ABAP program, we can also generate ABAP programs dynamically. Generating a program dynamically should be used as a last resort if other dynamic programming techniques do not satisfy your requirements.

Dynamic program generation is for expert programmers and should be used with caution; it can cause serious problems if you don't know what you're doing. The downsides of programs generated dynamically are that these programs can't be tested like regular programs and can involve serious security risks.

There are two types of programs that can be generated dynamically:

► **Transient program**

A *transient program* exists temporarily in the internal session memory and can be accessed only from the current internal session. The `GENERATE SUBROUTINE POOL itab NAME prog` statement is used to generate a temporary subroutine pool program. The internal table `itab` should be a character type and contains the source code of the program. The `prog` variable contains the name of the generated program, via which the program can be accessed.

The generated subroutine pool program can contain local classes or subroutines, which can be accessed from outside of the program. If the source code of the program contains any syntax errors, the program isn't generated.

Listing 16.28 shows sample code to generate a temporary subroutine pool program. In this example, we're appending the internal table `IT_SOURCE` with the source code of the subroutine pool program and generating the program using the `GENERATE SUBROUTINE POOL` statement.

```
DATA: it_source TYPE TABLE OF string,
      program TYPE string ,
      msg TYPE string.

APPEND 'PROGRAM.' TO it_source.
APPEND 'FORM subr.' TO it_source.
APPEND 'WRITE / 'This is dynamic subroutine''.' TO it_source.
APPEND 'ENDFORM.' TO it_source.

GENERATE SUBROUTINE POOL it_source NAME program MESSAGE msg.
IF sy-subrc = 0.
PERFORM subr IN PROGRAM (PROGRAM).
ENDIF.
```

Listing 16.28 Dynamic Subroutine Pool

► Persistent program

Persistent programs exist permanently in the repository and can be accessed just like other programs that are created manually. The `INSERT REPORT prog FROM itab` statement generates a persistent program. Here, the internal table `itab` contains the source code of the program, and the `prog` variable contains the program name.

You should be careful when using this statement: If a program with the same name already exists, it will be overwritten without any warning. To ensure you don't overwrite existing programs, we recommend checking the database table `TRDIR` to see if a program with the same name already exists.

Listing 16.29 shows example code to generate a program dynamically. In this example, we're filling the internal table `IT_SOURCE` with the source code line by line and then using the `INSERT REPORT` keyword to generate the program.

```
REPORT ZDYN_PROG.

DATA: it_source TYPE TABLE OF rsource-line.
```



```

APPEND 'REPORT ZDYN_EXAMPLE.' TO it_source.

APPEND 'WRITE / ''This program is generated dynamically''.'
      TO it_source.

INSERT REPORT 'ZDYN_EXAMPLE' FROM it_source.

```

Listing 16.29 Dynamic Program Generation

When generating programs dynamically, it's a good idea to create a program statically and use that program as a template to generate new programs. You can use the `READ REPORT INTO itab` statement to read the source code of an existing program, which can then be used as a template to generate a new program.

16.7 Summary

In this chapter, we explored the various concepts that can be used to make ABAP programs dynamic. Dynamic programming may seem intimidating, especially for a beginner, but it opens up many possibilities, such as developing applications that can handle dynamic requirements and work across multiple business functions.

This chapter looked at how field symbols and data references can be used to dynamically point to existing memory of data objects, and you saw how anonymous data objects can be created using data references.

We also discussed the RTTS framework and how it helps to dynamically find the type information of program data objects and to create new data types in the program during runtime. We concluded the chapter with a discussion of calling procedures dynamically and how to generate programs dynamically. We hope this knowledge will help you to approach program development from a different perspective.

In the next chapter, we'll discuss the process of debugging, which plays an important role in identifying problems within a program, especially if dynamic elements are involved.

This chapter touches on the debugging options offered in ABAP. The ABAP Debugger, as the name suggests, will help you debug code. This chapter explains all the options available in both the new and classic versions of this tool. We'll also briefly look at the new AMDP Debugger.

17 Debugging

A debugger is an invaluable tool for any programming environment. Whereas the syntax checker helps identify any syntax or static errors, the debugger helps you to identify any logical errors. Sometimes, it's impossible to detect logical errors without debugging. Having good debugging skills helps you to troubleshoot many problems with application programs.

Debugging helps you to analyze program flow for a specific scenario and analyze program execution. Using a debugger, you can stop at a particular point in the program execution and analyze the system status, the contents of the data objects, and so on. The debugger is a program tool that allows you to execute a program systematically and analyze the system status at each step to troubleshoot problems with the program.

If you're a beginner to ABAP, a debugger can be your best friend when it comes to understanding ABAP programming; it allows you to examine your programs step by step, which in turn can help you understand how each statement behaves and the functionality it offers. The more comfortable you are with ABAP, the better you become at debugging programs to troubleshoot problems. If you're an intermediate to advanced ABAP user, the debugger is still an immensely helpful tool to ensure error-free code.

SAP offers two debugging tool versions: (1) the old debugger, called the Classic Debugger, which was available for releases earlier than SAP NetWeaver AS ABAP 6.40, and (2) the New Debugger, released with SAP NetWeaver AS ABAP 6.40. As you may expect, the New Debugger offers many more features than the Classic Debugger.

If you're a beginner, you may find the New Debugger overwhelming, so we'll start our discussion in this chapter with the Classic Debugger and then look at all the new features offered by the New Debugger. However, SAP is phasing out the Classic Debugger, and as of SAP NetWeaver 7.5, the Classic Debugger has been declared obsolete. It's no longer possible to set the Classic Debugger as the default debugger, and we recommend using the New Debugger, which is the default debugger. The major difference between the two debuggers is that the Classic Debugger loads in the same session as the program, whereas the New Debugger uses a two-process architecture so that it's loaded in a new session. The New Debugger also offers a customizable user interface with many new tools.

In [Section 17.1](#), we'll begin by looking at the Classic Debugger features, which are similar to the New Debugger features. Specifically, we'll discuss how to activate and use the Classic Debugger and discuss the different types of views and additional features available.

[Section 17.2](#) will then touch on the features offered in the New Debugger, including its UI and tools. From there, in [Section 17.3](#), we'll discuss a new debugger tool as of SAP NetWeaver 7.5: the ABAP Managed Database Procedures (AMDP) Debugger.

In [Section 17.4](#) and [Section 17.5](#), we end the chapter by looking at how the debugging tools can enable to troubleshoot code and identify important concepts.

17.1 Classic Debugger

To understand the various features of the Classic Debugger, let's work with an example that sets up a problem and see how to go about troubleshooting the problem using this tool. For the example, assume a report isn't showing the expected output, and you want to debug the program to figure out where things are going wrong.

In this section, we'll begin by looking at how to activate and use the Classic Debugger.

17.1.1 Activating and Using the Classic Debugger

The Classical Debugger can be activated by changing the user settings. From any ABAP Workbench tool, such as ABAP Editor (Transaction SE38), use the menu path UTILITIES • SETTINGS; under the ABAP EDITOR tab, select the DEBUGGING sub-tab, as shown in [Figure 17.1](#).

Here, you can select the CLASSIC DEBUGGER radio button under the ABAP DEBUGGER section. Once the setting is changed, the system will call the Classic Debugger by default. Note that this setting is no longer possible to set in SAP NetWeaver 7.5.

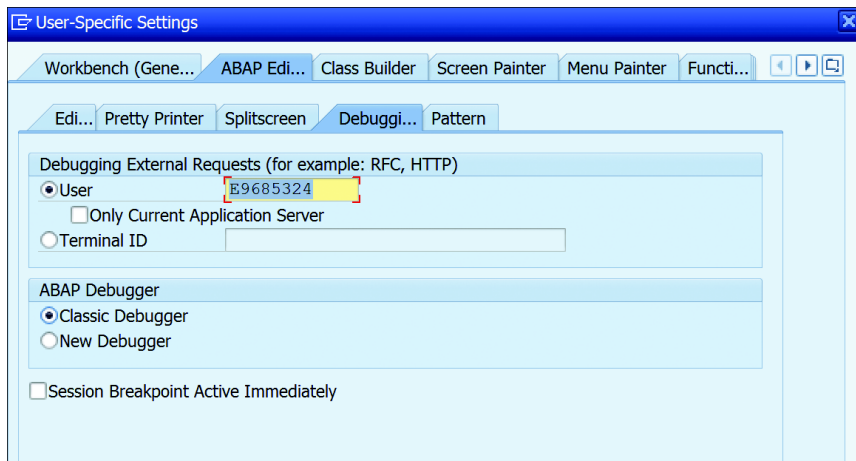


Figure 17.1 Setting Classic Debugger

Now that we've activated the Classic Debugger tool, let's look at the debugging process.

The first step to debugging is to identify from which point in the program flow you want to start debugging. Do you want to start debugging from the beginning of the program or from a specific point in the program? For example, if you're not seeing the expected output for your report program, you may want to start by troubleshooting the data selection logic. If your program has logic for selection screen validations, which typically occurs before data selection, you may want to skip debugging all of the selection screen logic and jump directly to the selection logic. So, in this case, our starting point will be the selection logic.

Once you identify the starting point, you need to set a breakpoint in the code. A breakpoint stops the program execution and calls the debugger. There are two types of breakpoints you can set:

► **Session breakpoint**

A *session breakpoint* is triggered when the program is executed from with the SAP GUI session.

► **External breakpoint**

The *external breakpoint* is triggered when the program is called externally from a different system. This is useful when you want to debug, say, RFC or Web Dynpro ABAP, when they're called from external applications.

To set a breakpoint, keep your cursor on the statement and click the SESSION BREAKPOINT button in the application toolbar, as shown in [Figure 17.2](#). Alternatively, you can double-click in the first column of ABAP Editor to set a breakpoint. Repeat the same process to delete the breakpoint. The program should be in an active status to set a breakpoint. If the program is not active, activate the program first before setting the breakpoint.

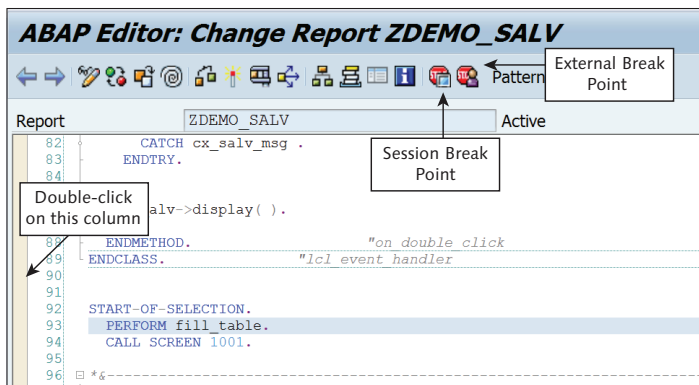


Figure 17.2 Setting Breakpoint

Once the breakpoint is set, execute the program; the debugger will automatically load when the program reaches the breakpoint location during program execution. When you're in debug mode, you can set new breakpoints or delete existing breakpoints by double-clicking at the beginning of the line. Click the SAVE button in the standard toolbar to save the new breakpoints. The Classic Debugger should now look similar to [Figure 17.3](#).

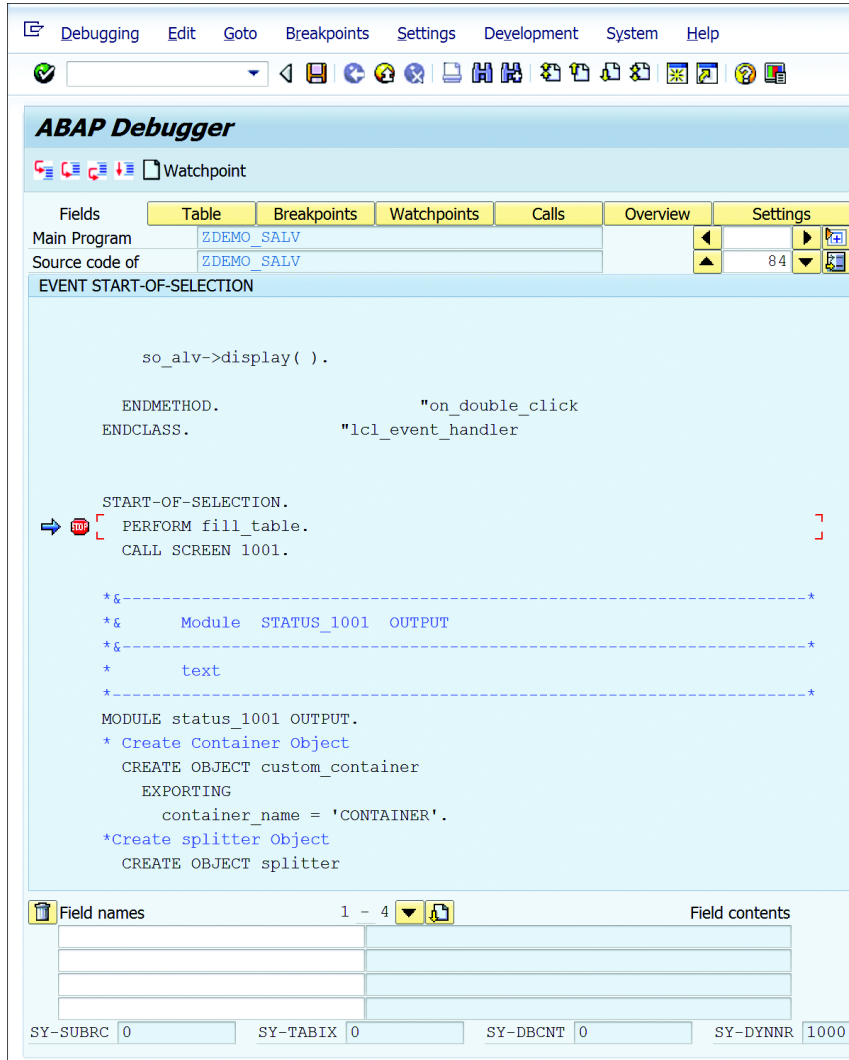


Figure 17.3 Classic Debugger

The top section of the debugger provides various options, as shown in [Figure 17.4](#). Here, you have the option to execute the code step by step, switch to different views, create watchpoints, and scroll through the debugger window. A *watchpoint* allows you to stop the program execution when a field contains a specific value. Watchpoints are useful for debugging based on a field value as opposed to a specific ABAP statement.

The different views on the debugger screen allow you to check information while debugging; you'll learn more about these views in the following subsections. The execution steps allow you to execute the code step by step to troubleshoot issues. The horizontal and vertical scrolling buttons allow you to scroll through the debugger window. The debugger window also displays the program and subprogram name that you're currently debugging.

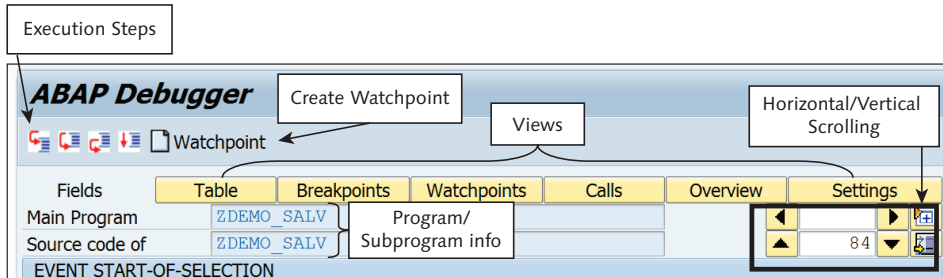


Figure 17.4 Debugger: Top Area

There are four ways to execute the code in debug mode:

- ▶ **SINGLE STEP** (Press **F5**)
This will execute the code line by line. Use this if you want to execute each statement, one-by-one.
- ▶ **EXECUTE** (Press **F6**)
This will execute the complete processing block and go to the next statement after the call location if used at the processing block call location. For example, if you press **F6** at the `PERFORM` statement or `CALL FUNCTION` statement, the debugger will execute the subroutine or function module completely without going inside the block. This is useful if you don't want to check the execution of a processing block and skip to the next statement after the call location. If **F6** is used at any location except a processing block call, it will behave similarly to **SINGLE STEP** and execute each line, one-by-one.
- ▶ **RETURN** (Press **F7**)
This option will exit the processing block if used inside a processing block. For example, if you've entered a processing block by pressing **F5** and after executing a few lines you realize you want exit the processing block because you don't want to analyze further code, you can press **F7** to leave the processing block and go back to the statement immediately after the call location.

► RUN (Press **F8**)

This will go to the next breakpoint or complete the program execution if no further breakpoints are set.

The Classic Debugger screen provides different views, which we'll discuss in the following subsections.

17.1.2 Field View

In the field view, you can assess the contents of different fields. At the bottom of the debugger screen, this view shows two columns, as in [Figure 17.5](#).

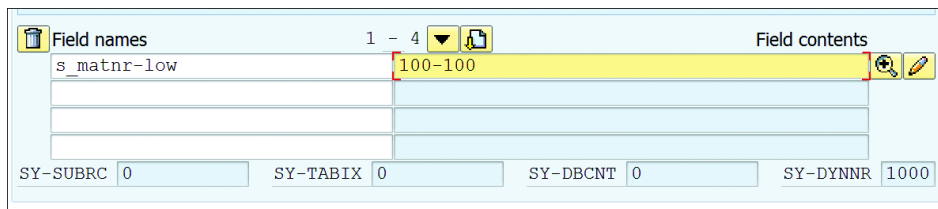


Figure 17.5 Checking Field Contents

You can double-click a field name to copy it to the FIELD NAMES column and see the field contents in the FIELD CONTENTS column. You can also manually type a field name in the FIELD NAMES column. Double-clicking a field name will show the technical attributes of the field. If the field is a structure, it will show the contents of the structure.

Clicking on the pencil icon at the end of the row will allow you to change the contents of the field. For example, if you want to test for a particular value, you can dynamically change the field contents in debug mode by clicking the pencil icon.

At the bottom of the screen, you can see the contents of certain system fields. This will help you quickly assess the status, such as if the `SELECT` was successful (`SY-SUBRC`), or check the number of rows fetched from the database after the `SELECT` was executed (`SY-DBCNT`).

17.1.3 Table View

The table view allows you to check the contents of internal tables, as shown in [Figure 17.6](#). Double-click a line number to see the complete line. You can use the

horizontal scrolling buttons at the top of the debugger to scroll horizontally in the table. Also, you can check a single row of a table by using the index number within square braces; for example, `it_mara[3]` will display the third row of an internal table like a structure.

The CHANGE, INSERT, APPEND, and DELETE buttons will allow you to change the contents of a row, insert a new row, append a new row, or delete a row, respectively. The plus icon in the top-right corner in [Figure 17.6](#) will expand the table view vertically.

1	MATNR	ERSDA	ERNAM	LAEDA	MTART
1	100-100	19941107	BALLER	20120214	HALB
2	100-101	19960305	KUNITZ	20070108	HALB
3	100-110	19941107	BALLER	20030607	ROH
4	100-112	20110310	ZECHA	00000000	ROH

Figure 17.6 Table Contents

17.1.4 Breakpoints View

The breakpoints view (see [Figure 17.7](#)) allows you to see all the breakpoints of the program. You can delete any breakpoint by double-clicking the breakpoint symbol to the left of the respective line.

No.	Breakpoint Type	in (absolute path)
1	Point in program ZTEST_PROGRAM(-)	
2	Point in program ZDEMO_SALV (51)	
3	Point in program ZDEMO_SALV (70)	
4	Point in program ZDEMO_SALV (133)	
5		
6		

Figure 17.7 Breakpoints View

17.1.5 Watchpoints View

The watchpoints view lists all the watchpoints set for the debugging session. Watchpoints allow you to interrupt the program execution when a particular field is updated with a specific value.

For example, say you want to debug the code within an internal table loop when a particular row is copied to the work area. In such a case, you can create a watchpoint to monitor the work area. When the work area is updated with the value, the program execution stops and waits for your interaction.

To create a watchpoint, click the CREATE WATCHPOINT button at the top of the debugger screen (refer back to [Figure 17.4](#)). This will open the CREATE/CHANGE WATCHPOINT dialog box, in which you can provide information about the field that you want to monitor and the value, as shown in [Figure 17.8](#).

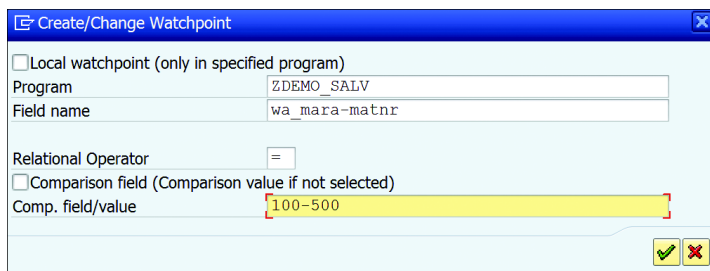


Figure 17.8 Creating Watchpoint

You can create multiple watchpoints and select the logical operator between the watchpoints as an OR condition or an AND condition, as shown in [Figure 17.9](#). Clicking the pencil icon allows you to edit a watchpoint, and clicking the DELETE icon allows you to delete a watchpoint.

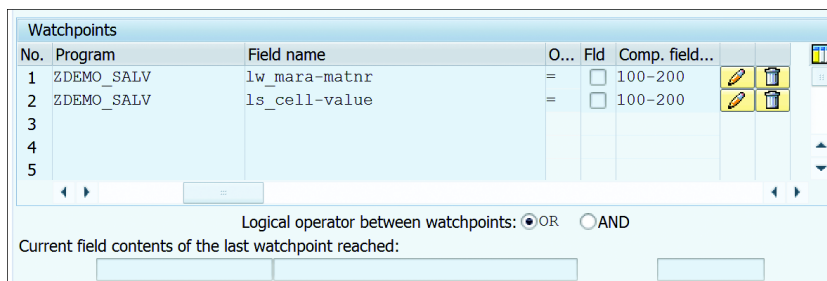


Figure 17.9 Watchpoints View

Once the watchpoint is set, press **F8** and the system will automatically stop at the next breakpoint or when the watchpoint is reached. Watchpoints are useful when you want to troubleshoot based on field values rather than program flow.

17.1.6 Calls View

The calls view provides information about the complete call stack of the session. This allows you to check all the programs loaded in the call stack, as shown in [Figure 17.10](#).

No.	Program	Type	Processing block
11	ZDEMO_SALV	METHOD	ON_DOUBLE_CLICK
10	CL_SALV_EVENTS_TABLE=====	METHOD	RAISE_DOUBLE_CLICK
9	CL_SALV_CONTROLLER_EVENTS	METHOD	RAISE_DOUBLE_CLICK
8	CL_SALV_GRID_ADAPTER=====	METHOD	DOUBLE_CLICK
7	CL_GUI_ALV_GRID=====	METHOD	DISPATCH
6	CL_GUI_CFW=====	METHOD	DISPATCH_SYSTEM_EVENTS
5	SAPFGUICNTL_CFW	FORM	CONTROL_DISPATCH
4	SAPFGUICNTL	FORM	CONTROL_DISPATCH
3	SAPMSSYD	FORM	%_CTL_INPUT
2	SAPMSSYD	MODULE (PAI)	%_CTL_INPUT1
1	ZDEMO_SALV	EVENT	START-OF-SELECTION

Figure 17.10 Call Stack

This view is useful if you want to check all the programs that are loaded into the session while your program is running. Checking the call stack allows you to verify if an expected procedure is loaded or ignored during program execution.

17.1.7 Overview View

The overview view lists all the processing blocks of the current main program, with an arrow pointing to the processing block that is being executed currently, as shown in [Figure 17.11](#). This view allows you to quickly identify the current processing block being executed.

17.1.8 Settings View

The settings view allows you to change the debugger settings and system behavior. There are various settings that can be changed, as shown in [Figure 17.12](#). You can learn more about these settings by referring to the SAP Help documentation

at https://help.sap.com/saphelp_erp60_sp/helpdata/en/c6/617d0ce68c11d2b2ab080009b43351/content.htm.

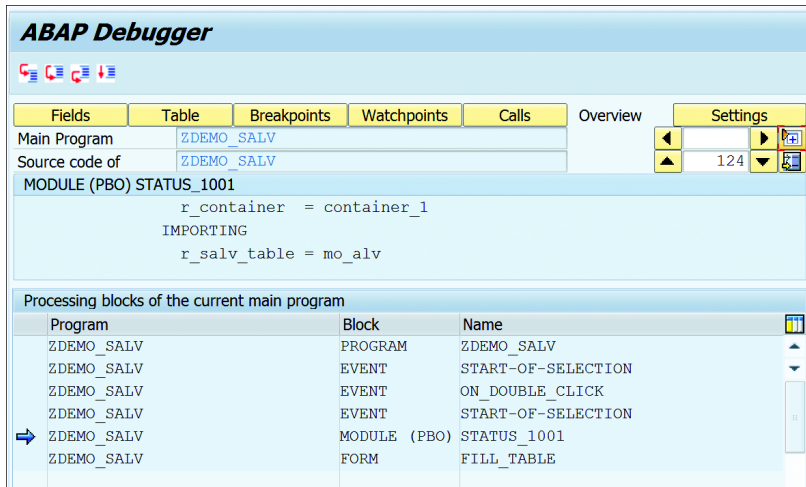


Figure 17.11 Overview View

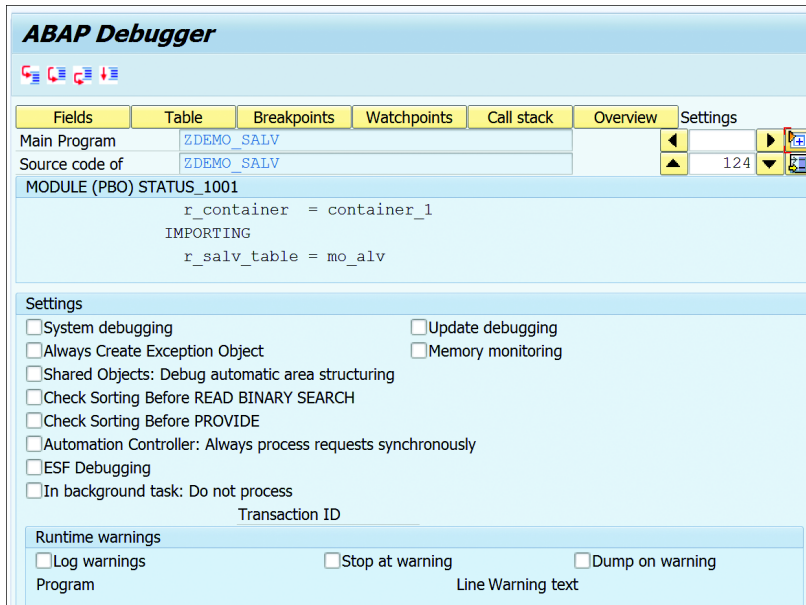


Figure 17.12 Classic Debugger Settings

17.1.9 Additional Features

The menu bar of the debugger screen offers additional features, as shown in [Figure 17.13](#). Under the debugging menu, you can select **CONTROL • GOTO STATEMENT** to jump to a specific cursor position. For example, if you've executed a statement and want to jump back and execute the statement again, you can keep the cursor on the statement and follow the menu path **DEBUGGING • CONTROL • GOTO STATEMENT**, as shown in [Figure 17.13](#). From this menu, you can also choose **SWITCH TO NEW ABAP DEBUGGER**.

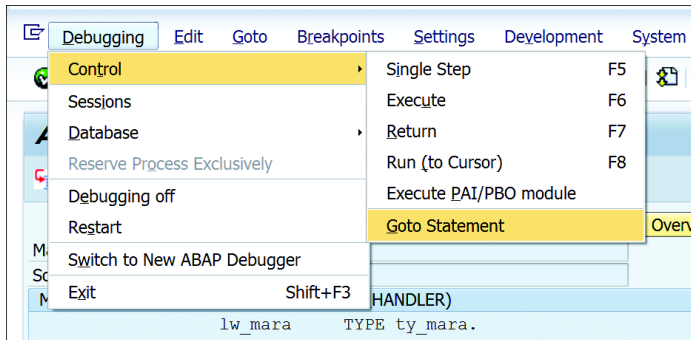


Figure 17.13 Jumping to Statement

You can select **DEBUGGING • DEBUGGING OFF** to turn off debugging and complete the program execution. This will complete the program execution without stopping at any additional breakpoints.

You can select **DEBUGGING • RESTART** to terminate the program execution. This is useful, for example, if you've changed any values in debug mode and don't want the program to update those values in the database.

The **GoTo** menu offers certain interesting options. Select **GoTo • DISPLAY DATA OBJECT** to load different data object views and see various data objects. For example, selecting **GoTo • DISPLAY DATA OBJECT • OBJECT** will open a view in which you can analyze the contents of an object, as shown in [Figure 17.14](#). This object view provides certain specific options to check the contents of the object. For example, you can filter based on the visibility section or display only static attributes or instance attributes of the object.

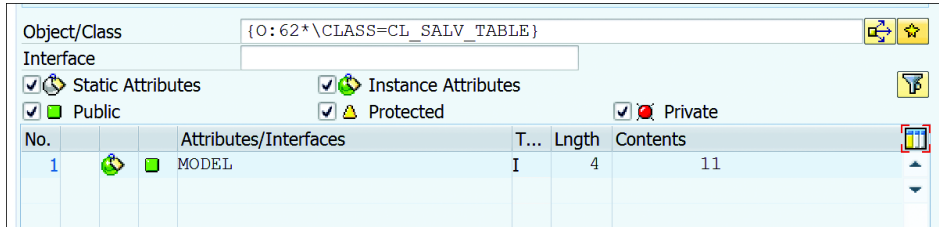


Figure 17.14 Object View

Following the menu path **GoTo • STATUS DISPLAY • MEMORY CONSUMPTION** allows you to check the memory consumption of the program. Meanwhile, following the menu path **BREAKPOINTS • BREAKPOINT AT** allows you to set a breakpoint when a particular statement is executed, a function module is called, and so on, as shown in [Figure 17.15](#). This option is particularly useful when debugging long programs.

For example, if you're getting a runtime error and want to debug what's causing the runtime error or check the status of the program before the exception is triggered, you can check the short dump to identify the statement causing the exception; once you're in debug mode, select the menu **BREAKPOINTS • BREAKPOINT AT • STATEMENT** to enter the statement at which you want to set the breakpoint. Then, the system will stop each time that statement is called. You can use the same **BREAKPOINTS • BREAKPOINT AT** menu to quickly stop at a particular **FUNCTION MODULE** or **METHOD** call.

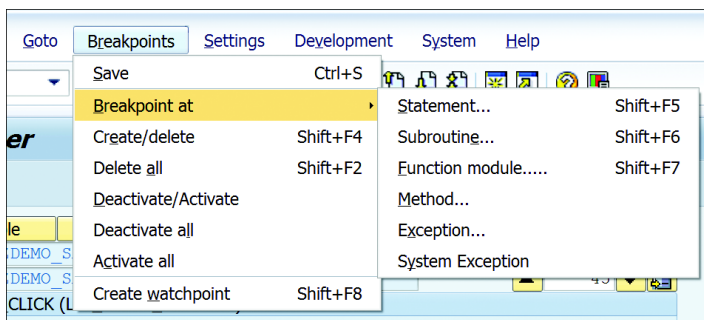


Figure 17.15 Setting Breakpoint at Particular Call

Another menu option worth mentioning is the option to activate system debugging, which allows you to debug system programs (programs of **TYPE S**). For

example, say you're debugging a program and come across a function module call, as shown in [Figure 17.16](#).

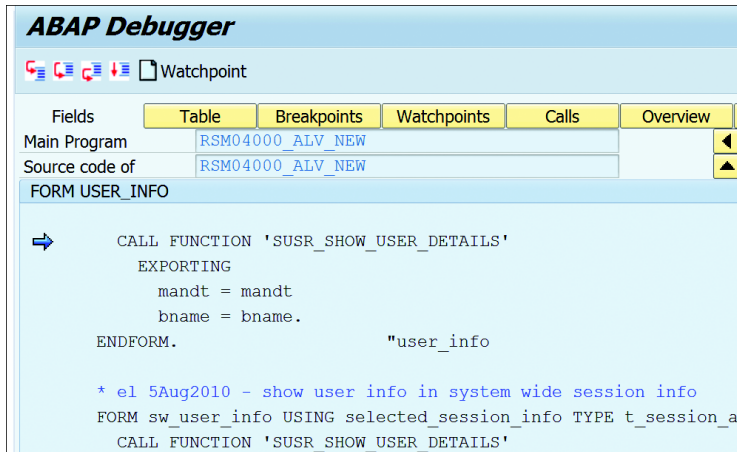


Figure 17.16 Debugging System Program

The `SUSR_SHOW_USER_DETAILS` function module shown in [Figure 17.16](#) belongs to a function group for which the `PROGRAM STATUS` is set to system program (type S), as shown in [Figure 17.17](#). If you press `[F5]` at this function module call, the debugger won't enter the function module. To enter the function module, you first need to activate system debugging. To do so, follow the menu path `SETTINGS • SYSTEM DEBUGGING`. It's turned off by default to allow you to skip debugging system programs, which usually don't pose any issues.

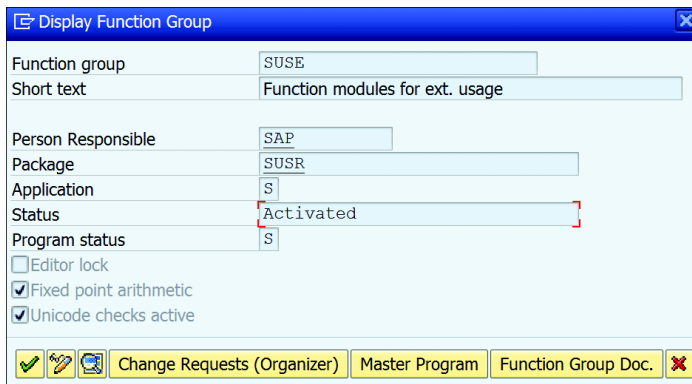


Figure 17.17 Program Status

Other than keeping a breakpoint in the program source code, you also can activate the debugger by typing `/H` in the command bar of the screen. This option is useful if you want to start debugging from the PAI of a certain screen of the application. For example, say you want to analyze the code called when you select a particular button on the screen. Going through the source code to identify the code called for the button action is cumbersome. You easily can avoid manually looking through the source code by activating debugging for the screen and performing the action for which you want to examine the program execution.

Sometimes, you may want to activate debugging for modal dialog windows. Because these windows don't have a command bar, you can't use `/H` to activate the debugger for such windows. To activate the debugger for dialog windows without a command bar, you can create a shortcut for the `/H` command and then drag and drop this shortcut onto the dialog window. To create a shortcut, click the **GENERATE SHORTCUT** icon in the standard toolbar of any screen. This will open a dialog window, as shown in [Figure 17.18](#).

The screenshot shows the 'Create New SAP Shortcut' dialog box with the following fields and values:

- Title: SAP Easy Access
- Type: System Command
- Command: /H
- System Description: ECC
- System ID: DEV
- Start Parameters: (empty)
- Client: 800
- User: (empty)
- Password: (empty) (Use Not Recommended)
- Language: EN - English
- Location: Desktop

Buttons at the bottom: Help, Cancel, < Back, Next >, Finish.

Figure 17.18 Create Shortcut

Under **TYPE**, select **SYSTEM COMMAND** and enter `"/H"` in the **COMMAND** field. For **LOCATION**, select **DESKTOP** and then click **FINISH**. This will create a shortcut on your desktop. You can drag and drop this shortcut onto any modal dialog box to turn on debugging for the window.

17.2 New Debugger

The Classic Debugger loads in the same session as the program. This poses certain technical challenges for debugging ABAP programs. This also restricts developing a new UI design for the debugger. To overcome such challenges, SAP introduced the New Debugger.

In the following sections, we'll look at the UI and tools offered with the New Debugger that differ from the Classic Debugger. We'll also look at the overall layout and the use of sessions.

17.2.1 UI and Tools

The New Debugger provides a feature-rich UI that can be customized to display up to four tools in a desktop. [Figure 17.19](#) shows the New Debugger with the UI customized to display different tools. One tool shows the source code, another tool shows an internal table, the third tool shows an object, and the fourth tool shows variables.

Each tab is called a *desktop* in the New Debugger, and there are twelve total desktops, with each desktop capable of displaying four tools. Each tool has its own options, as shown in [Figure 17.20](#). Using the available options for the tool, you can load new tools in the desktop, replace the tool, swap the tool, maximize the tool, and so on. There are also specific services for the tool that can be accessed using the **SERVICES OF THE TOOL** icon.

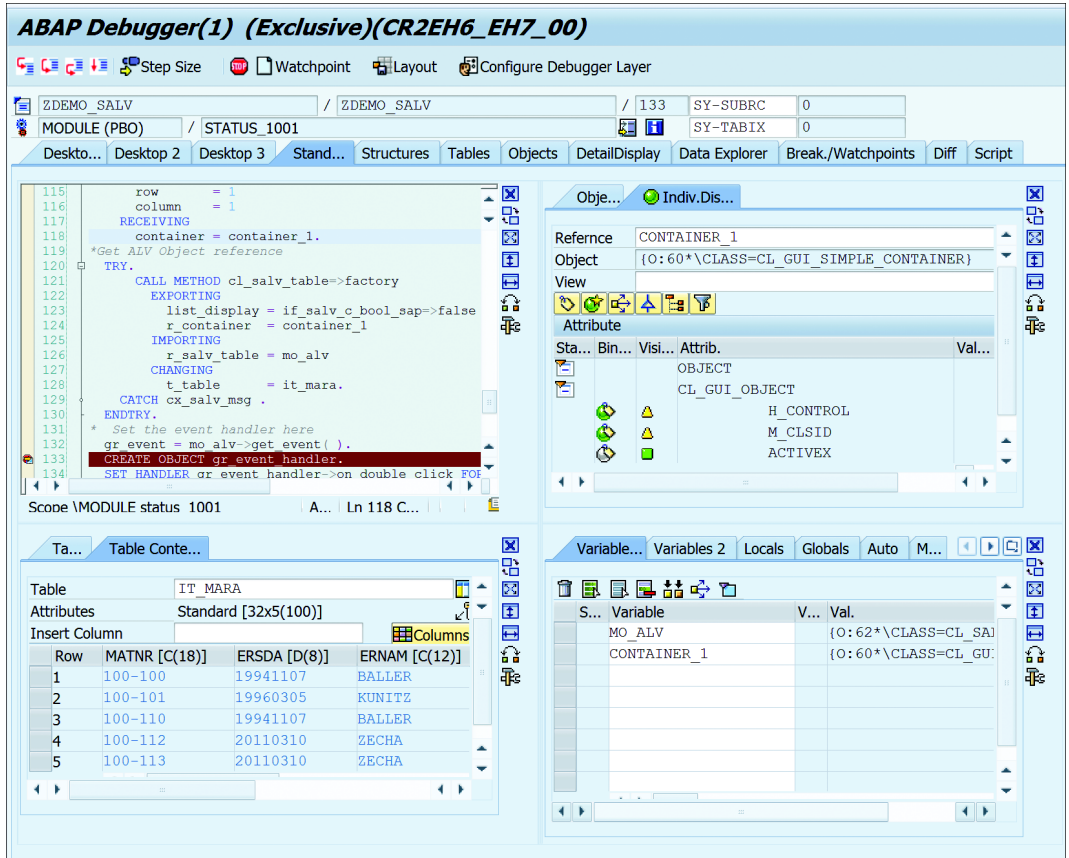


Figure 17.19 New ABAP Debugger



Figure 17.20 Tool Area

Selecting the NEW TOOL option opens a window, as shown in [Figure 17.21](#). As you can see, the New Debugger provides wider tool support than the Classic Debugger. It also provides some special tools—for example, a tool to debug Web Dynpro applications. Select any tool from the NEW TOOL window to add it to the desktop.

The process to run through the code and analyze data objects is similar to that for the Classic Debugger (using [F5](#) to [F8](#) function keys, as discussed in [Section 17.1.1](#)). Compared to the Classic Debugger, the CREATE WATCHPOINT window, apart from variables, allows you to set watchpoints to monitor changes to object attributes.

Following the menu path MISCELLANEOUS • DEBUGGER TIPS will lead you to various tips to use the New Debugger. We recommend that you go through all the tips to feel comfortable using the New Debugger.

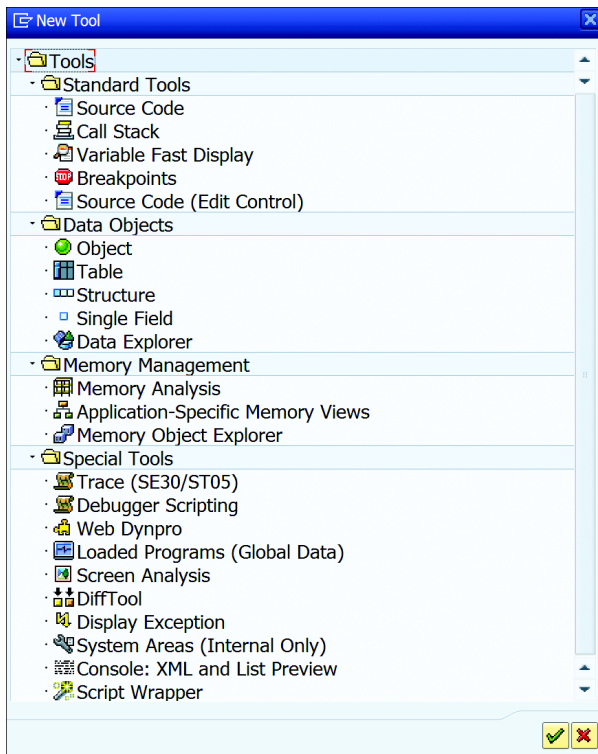


Figure 17.21 New Debugger Tools

17.2.2 Layout and Sessions

You can save the layout of the New Debugger via the menu path **DEBUGGER • DEBUGGER SESSION • SAVE LAYOUT** (see [Figure 17.22](#)). This option is useful if you customized the desktop layout with different tools and want to use the result as your default layout.

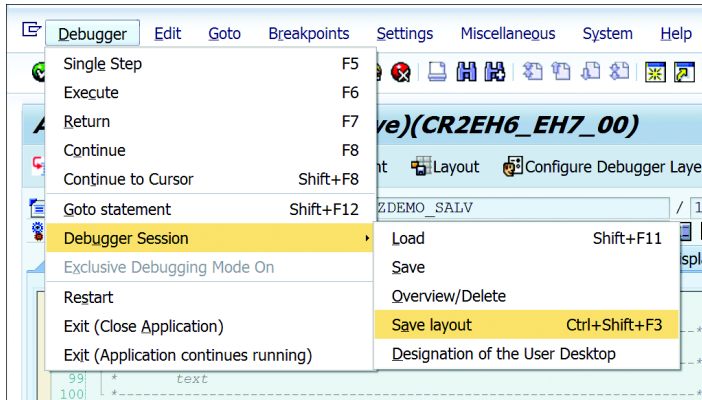


Figure 17.22 Save Layout

You can save the current debugging session via the menu path **DEBUGGER • DEBUGGER SESSION • SAVE**. This will open the **SAVE CURRENT DEBUGGING SESSION** window, as shown in [Figure 17.23](#). You can save the current debug session to a file or to database by selecting the relevant radio button and providing a name for the session. Optionally, you can also add a comment to easily identify a saved session.

Different components of the session, such as layout, breakpoints, watchpoints, settings, options and variables, can be saved (see [Figure 17.23](#)). You can load a saved session via the menu path **DEBUGGER • DEBUGGER SESSION • LOAD**, view the saved sessions and delete them via the menu path **DEBUGGER • DEBUGGER SESSION • OVERVIEW/DELETE**, and change the names of the desktop tabs via the menu path **DEBUGGER • DEBUGGER SESSION • DESIGNATION OF THE USER DESKTOP**.

Because the New Debugger opens in a new session, when the application being debugged becomes ready for input, the external debugger session remains open and the application screen is presented to the user for input. If any further breakpoint is reached, the debugger loads automatically. This ensures that the debugger doesn't lose any layout changes or tool arrangements set previously.

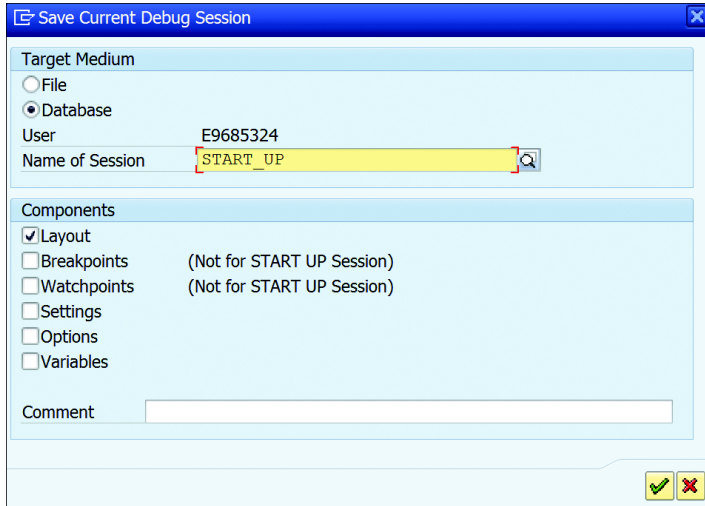


Figure 17.23 Save Current Debug Session

To close the external debugger session when the control is returned to the application, use the `/hx` command in the command bar of the application screen. The debugger can be turned on from any transaction screen using the `/h` command in the command bar of the SAP GUI screen.

17.3 AMDP Debugger

ABAP Managed Database Procedures (AMDP) are a new feature as of SAP NetWeaver AS ABAP 7.5. They enable developers to write database procedures directly into ABAP code. Because the AMDP Debugger deals primarily with the topic of ABAP in SAP HANA we won't dive into much depth in this section.

You can debug AMDPs via the ADT. The AMDP Debugger enables you to debug executed database procedures that run on top of an SAP HANA database.

The following are the prerequisites that must be fulfilled before using the AMDP Debugger:

- ▶ SAP NetWeaver 7.5 or higher
- ▶ SAP HANA SPS 09 or higher
- ▶ ADT 2.51, or higher

The AMDP Debugger provides numerous features. The following list highlights a few of these capabilities:

- ▶ Execute database procedures
- ▶ Debug SQLScript and Llang
- ▶ Analyze table variables
- ▶ Analyze scalar variable values
- ▶ Create and manage breakpoints of AMDP classes in embedded code

In the next two sections, we end the chapter by looking at how the debugger can be used for troubleshooting and as a learning tool.

17.4 Using the Debugger to Troubleshoot

Sometimes, it's difficult to identify problems with programs just by reading the program code. Programs often behave unexpectedly under particular circumstances.

Debugging helps you to check how the program is behaving at a particular moment, under particular circumstances. Sometimes, the problem may not be with the program at all and may be externally generated. For example, the posting program may not be posting the document not due to program error, but due to the wrong data being fed into the program. In such a situation, debugging the program may be the only way to troubleshoot.

Debugging helps troubleshoot any logic errors that are difficult to identify by reading the code. Understanding how to use the debugger tool is just one part; the major skill lies in understanding the program flow and knowing what to debug. If the program is big, you may end up spending hours or even days debugging to identify a problem.

In this section, we'll provide tips that should help you troubleshoot issues:

- ▶ The first step to troubleshooting is to identify where to start. For example, if the program isn't giving the right output, start with the selection logic.
- ▶ Use **F6** to skip debugging unwanted processing blocks. For example, if you come across a function call and you're sure that the problem doesn't lie in the function module, press **F6** to skip debugging the function module call. This saves you a lot of time while debugging.

- ▶ Check program variables while debugging. It may sound obvious, but many people—especially beginners—debug the whole program without even bothering to check if the program variables are initialized with the right data.
- ▶ Check if the program is executing the right control structures. If the program is expected to execute the code in a certain IF condition for the given data, check if that's happening. If the program variables are initialized with unexpected data, the program logic may fail. This is another reason that you should check if the program variables are initialized with expected data.
- ▶ Set and save breakpoints from within the debugger. When executing long programs, you may go through a long list of code before you can zero in on the problem. Setting a breakpoint halfway through may help you to start from where you left off when you rerun the program. However, breakpoints are valid only for the current session, so if you log out from the system or get disconnected, you may have to start from the beginning.
- ▶ Use watchpoints to run through the code faster. Want to check how the program is behaving when the 898th record of the internal table is being processed? Don't press **F5** until you reach the 898th loop; use watchpoints to directly jump to that row.
- ▶ Use the BREAKPOINT AT option (see [Figure 17.24](#)) in the debugger to go directly to a statement or a procedure call if you feel a certain statement or a procedure is causing the issue.

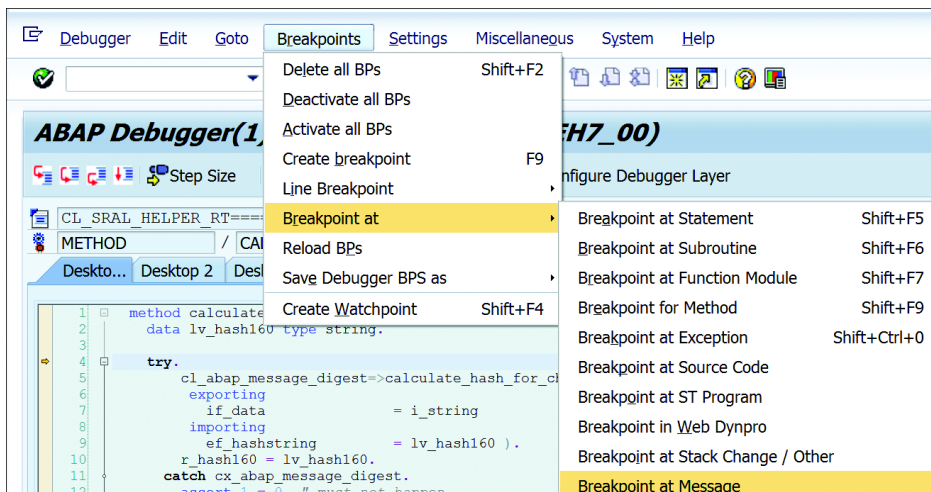


Figure 17.24 Setting Breakpoint

- ▶ Want to check why a particular error message or exception is being thrown by the program? Simply place the breakpoint for that message or exception using the BREAKPOINT AT menu (see [Figure 17.24](#)), and monitor the program variables to identify what is triggering that exception.
- ▶ If you encounter a loop statement, keep a breakpoint outside the loop and press **F8** to continue debugging the program without going through each loop pass.
- ▶ Debug only relevant code. If you're facing issues with a certain task in the application, focus on debugging only the relevant code for that task. Using **/H** to start debugging from that specific screen may help you understand all the code that's being executed for that task.
- ▶ Short dumps are your best friends. If the program is generating a short dump, read it carefully; it will provide invaluable information about the problem.

These tips should help make you comfortable when troubleshooting problems using the debugger. The more you understand ABAP programming, the better you become at debugging.

17.5 Using the Debugger as a Learning Tool

In addition to using the debugger as a troubleshooting tool, you also can use the debugger as a learning tool. Debugging helps you to see what exactly is happening behind the scenes when you execute a program. You don't have to imagine what a particular statement is doing; just debug to see what exactly happens when the statement is executed. Debugging helps you visualize program execution better, which in turn helps you to understand the program flow. Debugging indirectly helps you to understand programming better by relating various actions performed by different ABAP statements.

If you're a beginner, we recommend that you debug every program that you create to help you understand the program flow. For example, if you're using events, check in debug mode to see the order in which the events are triggered. As an exercise, list all the event blocks (such as selection screen events, reporting events, list events, etc.) in an ABAP program, and in debug mode check the sequence in which these events are called.

Developing debugging skills is a continuous process, so the more you debug, the better you become.

17.6 Summary

In this chapter, you learned how to use the debugger tool. We discussed both Classic Debugger and New Debugger. We explored all the options available while debugging programs. We also provided some tips, which you can keep in mind while troubleshooting programs.

In the next chapter, we'll discuss developing forms in SAP. Forms allow you to print documents, and because almost all businesses need forms, creating them is one of the frequent tasks you're likely to undertake.

In this chapter, we'll explain how to create forms using the three primary SAP form technologies. We'll describe the process and provide step-by-step examples.

18 Forms

Forms print application data. Reports are typically used to process data in a user-driven fashion and display the result on the screen. With forms, we process the data from the database and print the output. Because reports also provide the ability to print the output, you may ask, "Why exactly do we need forms?" With reports, you're limited to displaying the data in a tabular or hierarchical format, but forms provide more control over how you print the output.

If you've ever seen an invoice, you may have noticed that the data on the invoice isn't exactly in a tabular format. An invoice typically has many sections (or windows) in which different information is printed. For example, one window on the top right might show invoice details such as the invoice number, payment due date, and so on. Another window on the top left might show customer details and shipping information. The main section in the center of the page might show item details and the like.

Figure 18.1 shows a typical invoice; you can see the invoice data printed across various sections of the page.

Forms generally are used to print document data such as sales orders, invoices, delivery documents, and so on. Sometimes, there is a legal requirement to print these documents in a specific format. With form development, we use a layout designer that helps format the page for printing.

There are three different technologies available to develop forms: SAPscripts, smart forms, and SAP Interactive Forms by Adobe. Each new technology has attempted to make developing and maintaining forms easy. The oldest among them is SAPscripts, which always was and still is a pain to develop and has a lot of limitations. We'll discuss SAPscripts in Section 18.1.

SAP Company Logo

COMMERCIAL INVOICE PAGE 1 OF 1

EXPORTER IDES US INC Dan Gallagher 1230 Lincoln Avenue NEW YORK NY 10019		INVOICE NO. 90031228		DATE 01/18/2002	
ULTIMATE CONSIGNEE DC Werk 1 Altonaer Strasse 98 D-10557 BERLIN		PURCHASE ORDER NO. adv1219		B/L AWB No. NI123456	
PURCHASING CONSIGNEE Deutsche Computer AG Hauptstrasse 67 D-15930 BERLIN		COUNTRY OF ORIGIN Japan		DATE OF EXPORT 12/19/2001	
FORWARDING AGENT		LETTER OF CREDIT INCOTERMS		TERMS OF PAYMENT FOB New York	
REMARKS		2002 without deduction			
ESTIMATED TIME OF ARRIVAL 01/19/2002		CONTAINERIZED (checkbox) <input checked="" type="checkbox"/> Yes <input type="checkbox"/> No		PORT OF EXPORT New York, Port	
		MODE OF TRANSPORT Sea		PORT OF DEST. (checkbox) Bremerhaven Container	
				EXPORTING CARRIER NedLoydd	

ITEM DESCRIPTION OF MERCHANDISE	H.S. NUMBER	QUANTITY	UNIT PRICE	VALUE IN USD
10 Memory 256 MB		100 EA	29.75	2,975.00

LINE TOTAL	2,975.00
MISC. CHARGES (packing, freight, etc.)	
INVOICE TOTAL	2,975.00

CERTIFICATIONS

AUTHORIZED SIGNATURE

Figure 18.1 Typical Invoice

Realizing the limitations of SAPscripts, SAP introduced smart form technology, which simplified the form development to a large extent. We'll cover smart forms in [Section 18.2](#). More recently, SAP has adopted Adobe forms technology, which further simplifies form development. We'll talk about SAP Interactive Forms by Adobe in [Section 18.3](#).

Let's start our discussion of form technology in SAP systems by looking at SAPscripts.

18.1 SAPscripts

SAPscripts is the oldest of the three technologies available for form development. We use the Form Painter tool to develop forms based on SAPscripts. Because SAPscripts is obsolete and replaced with smart forms and Adobe technology, we recommend not using it for new developments. However, SAP will support existing forms developed using SAPscripts for years to come and your company may still be using old forms that were developed in SAPscripts, so understanding how it works is crucial from a maintenance point of view.

In this section, we'll explain the development activities involved with form development using SAPscripts. We'll limit our discussion to concepts that lay the foundations for understanding the newer technologies (smart forms and Adobe forms) or that help you appreciate the advantages that the new technologies bring to the table.

18.1.1 Overview and Layout

From a developer point of view, there are two components involved with form development: the driver (or interface) program, via which the required data is extracted from the database, and the form layout, in which the extracted data is printed. The driver program is an executable program created using ABAP Editor. The form layout is designed using Form Painter (Transaction SE71). A component called a *composer* acts as a central output module, which prepares the text for a certain output device by including all the formatting information.

The composer works behind the scenes and is not visible to the outside. We activate the composer by calling certain function modules in the driver program to process the form. The driver program retrieves the data from the database and the form and controls the output.

Let's get started with form development and discuss the various activities as we go along. For an example, let's print some header and item data from a billing document. The driver program can be executed directly, but in most cases it's called from the application using the message control or linking an output type. We'll discuss message control and output types in more detail later in this section.

For the example, we'll use the driver program to fetch the data from the billing document header and item tables `VBRK` and `VBRP`, respectively. In this driver program, we'll call a form to initiate form processing.

The code in [Listing 18.1](#) shows the selection logic in the driver program.

```
REPORT zdemo_sapscrip_driver.
*Document header data
TYPES : BEGIN OF ty_vbrk,
        vbeln TYPE vbeln_vf,
        fkdat TYPE fkdat,
        kunrg TYPE kunrg,
        netwr TYPE netwr,
        END OF ty_vbrk.
*Document item data
TYPES : BEGIN OF ty_vbrp,
        vbeln TYPE vbeln_vf,
        posnr TYPE posnr,
        arktx TYPE arktx,
        netwr TYPE netwr,
        END OF ty_vbrp.

*Customer data
TYPES : BEGIN OF ty_kna1,
        kunnr TYPE kunnr,
        land1 TYPE land1_gp,
        name1 TYPE name1_gp,
        name2 TYPE name2_gp,
        ort01 TYPE ort01_gp,
        END OF ty_kna1.

DATA : wa_vbrk TYPE ty_vbrk,
        wa_kna1 TYPE ty_kna1,
        it_vbrp TYPE STANDARD TABLE OF ty_vbrp,
        wa_vbrp TYPE ty_vbrp.

PARAMETERS p_vbeln TYPE vbeln_vf.

START-OF-SELECTION.
    PERFORM fetch_data.
    PERFORM print_form.
*&-----*
*&      Form  FETCH_DATA
*&-----*
FORM fetch_data .
    SELECT SINGLE vbeln fkdat kunrg netwr FROM vbrk INTO wa_
vbrk WHERE vbeln EQ p_vbeln.
    IF sy-subrc IS INITIAL.
```

```

        SELECT SINGLE kunnr land1 name1 name2 ort01 FROM kna1 INTO wa_
kna1 WHERE kunnr EQ wa_vbrk-kunrg.
        SELECT vbeln posnr arktx netwr FROM vbrp INTO TABLE it_
vbrp WHERE vbeln EQ wa_vbrk-vbeln.
    ENDIF.
ENDFORM.
*&-----*
*&      Form  PRINT_FORM
*&-----*
FORM print_form .

*Add print form logic

ENDFORM.

```

Listing 18.1 Driver Program Code to Fetch Application Data

In [Listing 18.1](#), we modularized the code by using two subroutines: `fetch_data` and `print_form`. In the `fetch_data` subroutine, we're selecting the header and item data of the billing document into their own internal tables. We're also fetching the customer data from the customer master table `KNA1`. We'll add the form printing logic to the `print_form` subroutine once we develop the form.

As previously mentioned, the form is designed using the Form Painter tool (Transaction SE71). For this example, we'll print the company logo at the top left of the page and the document header data, such as the document number, document date, and net value, at the top right of the page. The payer details are printed in a section below the company logo. The item details are printed in the body of the page.

[Figure 18.2](#) shows a rough layout of the planned form. SAP calls each section of the page a *window*. When designing the form layout, we create different windows on the page in which the planned data will be printed.

In SAPscripts, there are three types of windows that you can create:

► **Constant window**

This window is used if you want to print some constant text on multiple pages. The window size is the same on all the pages and displays the same content on every page. This makes it easy to edit the content, because you only need to edit once to reflect the changes in all the pages. The length of this window is fixed. If you try to print more lines than the window can accommodate, the data will be truncated. You can have any number of constant windows in a form.

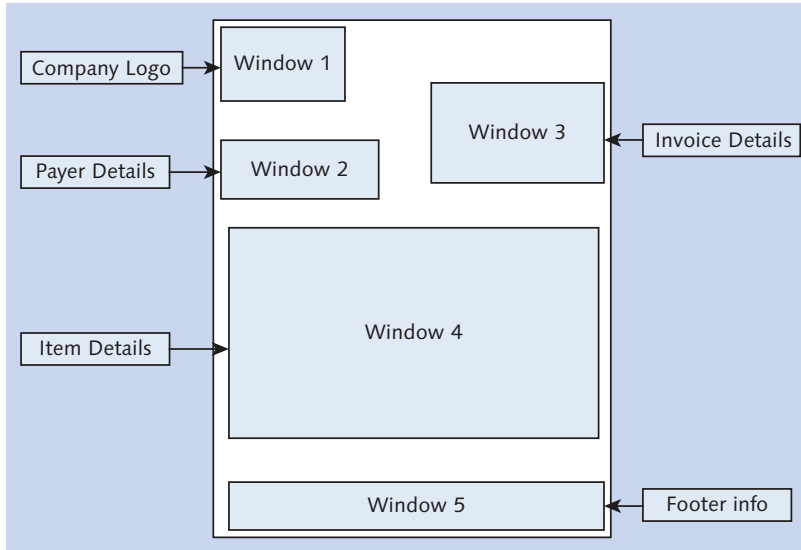


Figure 18.2 Form Layout

► **Variable window**

This window is used when you want to print data that varies in each window. Variable windows can be of different sizes and show different content. Similar to constant windows, the length of a variable window is fixed. If you try to print more lines than the window can accommodate, the data will be truncated. You can have any number of variable windows in a form, and each window is a separate entity.

► **Main window**

The width of the main window is fixed on all pages, but the length of the window isn't fixed. It will automatically trigger a page break and call the next page if the number of lines exceeds the window height. We use this window to print data for which the number of lines are not known at design time. In other words, if the number of lines is dynamic, such as for the item lines of a document, you use a main window. A form should have at least one main window. You can have multiple main windows in a form, but all the main windows share the same code and have the same width throughout the form. A main window consists of a top area and a bottom area, which can be used to print

text elements such as column headings (in the top area) and subtotals (in the bottom area).

Various formatting options, including paragraph and character options, help format the text of a form. Forms can be created in multiple languages so that users from different regions can use a form in their own language. Forms are also client-dependent, so if you create a form in one client and need to use it in another client, you need to manually transport it to the other clients of the system.

Now that we've reviewed the basic layout of a form, in the next section we'll walk through the steps for creating a form layout using SAPscripts.

18.1.2 Creating the Form Layout

In this section, we'll create the form layout as per the design shown in [Figure 18.2](#).

Proceed with the following steps:

1. Open Form Painter using Transaction SE71. On the initial screen of the Form Painter, enter the name of the form and the language (see [Figure 18.3](#)). This language will be the original language of the form and can be later translated to other languages. Click the CREATE button.

Figure 18.3 Form Painter: Initial Screen

- The system shows an informative message indicating that the form isn't available in the current client. Click OK in the message dialog box to create the form. The system shows the CHANGE HEADER screen (based on the subobject selected in previous step), in which some administrative data such as the short description and translations can be maintained. You can also maintain certain basic settings, such as page format, default paragraph format, and so on. We'll revisit this header screen later before activating the form.

Form: Change Header: ZCA_FORM

Pages Windows Page Windows Paragraph Formats Character Formats

Administrative Data Basic Settings

Administration Information

Form ZCA_FORM

Meaning Demo Form for Billing Document

Status New - Not saved

Classification

Package

Client ID 800

Created on [] 00:00:00 By [] Release: []

Changed on [] 00:00:00 By [] Release []

Language Attributes

Language Key EN

Original language EN

Translate

Into All Languages

To Individual Languages

Do Not Translate

Figure 18.4 Form: Change Header

- To activate the layout designer, from the menu select SETTINGS • FORM PAINTER to open the USER-SPECIFIC SETTINGS window. In this window, under the SAPSCRIPT tab, select GRAPHICAL FORM PAINTER (see [Figure 18.5](#)) and click OK. Once the graphical Form Painter is activated, you'll see a LAYOUT button in the toolbar. Click the LAYOUT button to start the graphical Form Painter.

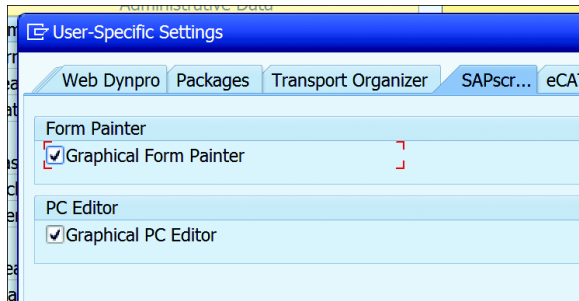


Figure 18.5 Activating Graphical Form Painter

4. The graphical Form Painter opens in a separate screen, with one default main window created (see [Figure 18.6](#)).

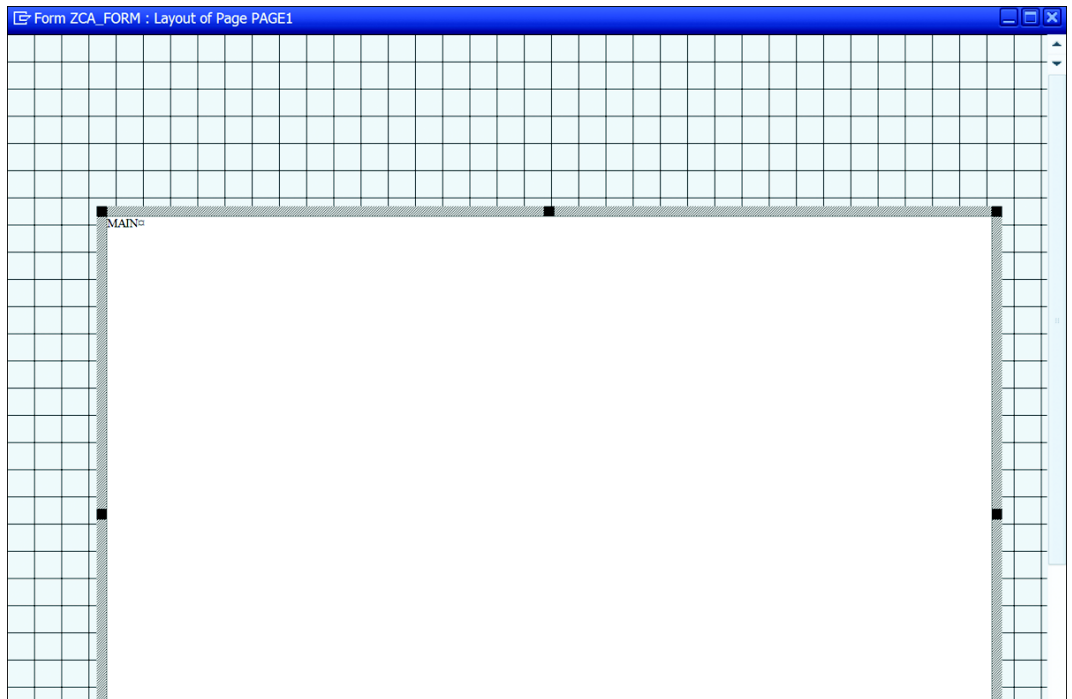


Figure 18.6 Graphical Form Painter

5. You can create additional windows and pages by using the CREATE button in the respective sections of the Form Painter page layout screen, as shown in [Figure 18.7](#). Certain attributes for the page, such as a short description for the page, the next page that should be called in sequence, and so on, can be maintained by selecting the MAINTAIN PAGE ATTRIBUTES button.

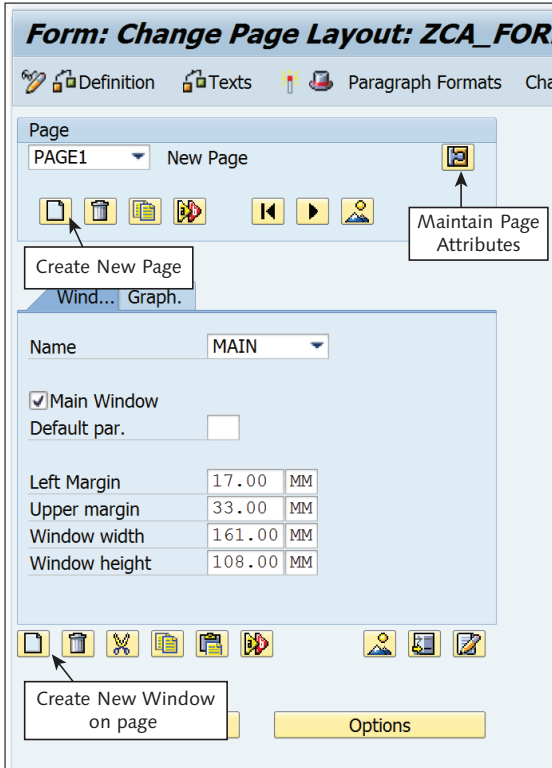


Figure 18.7 Form Painter Page Layout Screen

6. Click the CREATE WINDOW button to create a new window. The window appears in the graphical Form Painter, where you can adjust its size and position (see [Figure 18.8](#)). Rename the window to reflect the content it shows and provide a meaningful short description. Providing a meaningful name and short description for the window is important because it helps other developers easily maintain your form.

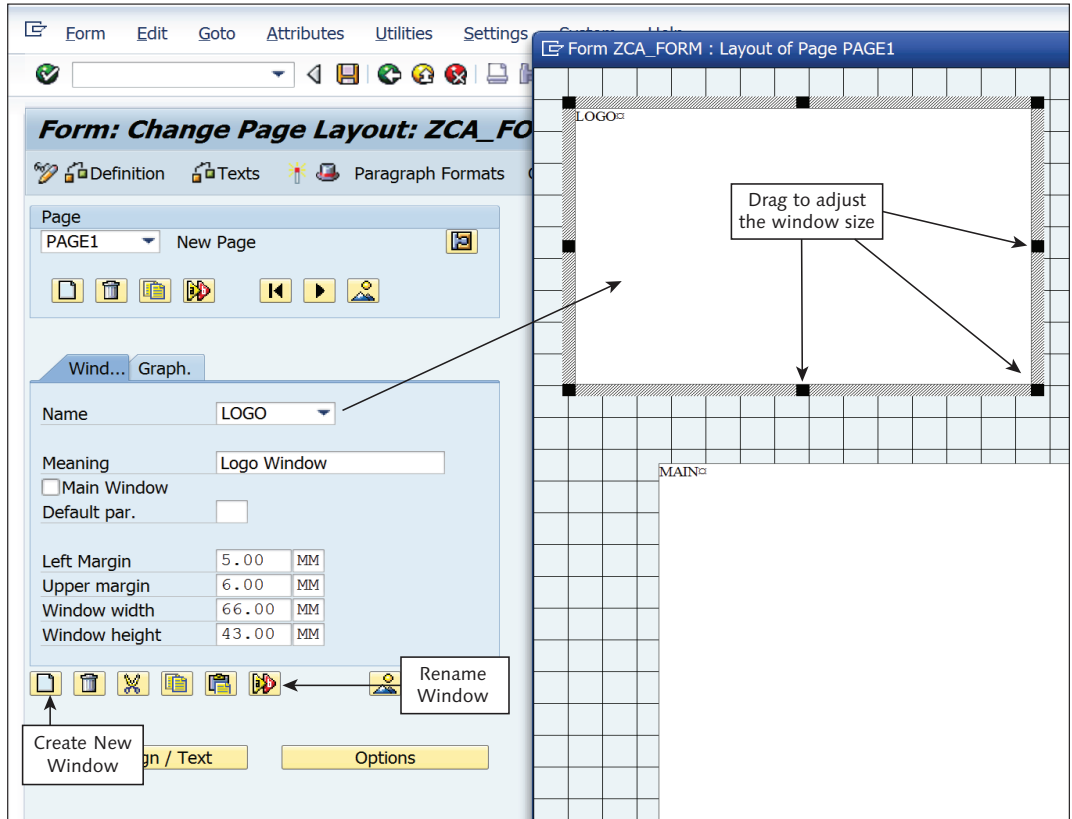


Figure 18.8 Creating New Window

7. Create the other windows on the page as per the layout shown in [Figure 18.2](#). Once done, deactivate the graphical Form Painter (using the menu path **SETTINGS • FORM PAINTER**) to go back to the **CHANGE PAGE WINDOWS** screen, as shown in [Figure 18.9](#). All the windows that you've created for the page will be displayed in this screen. If you want to, you can create multiple pages with different windows.

For example, you may want to print the customer details and invoice details on the first page but don't want them to be repeated on further pages. In such a case, you can create two pages and have the windows to show the customer and invoice details created on the first page only and have the second page consists of only the main window.

In the page attributes (see [Figure 18.7](#)), you can maintain the page sequence accordingly. For example, for PAGE1, the next page will be maintained as PAGE2, and for PAGE2, the next page will be maintained as PAGE2 itself. This way, you can print various information on PAGE1 but print only the item details on subsequent pages. This also helps to reduce the number of pages while printing by ensuring that unnecessary data isn't printed on all pages.

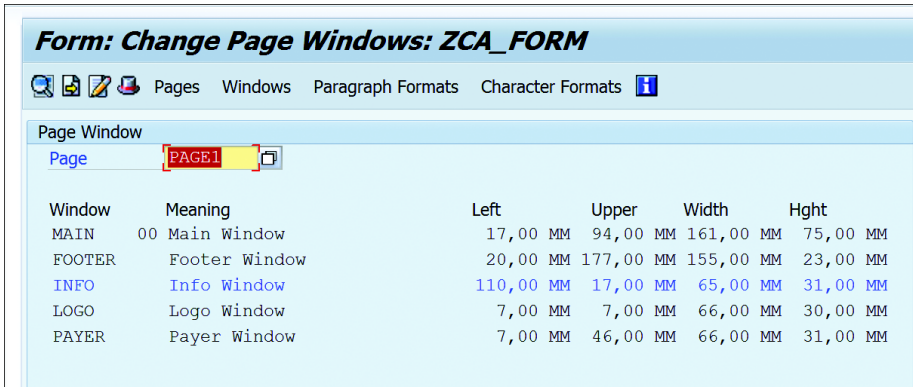


Figure 18.9 Page Windows Screen

8. Click the **WINDOWS** button in the toolbar to display all the windows of the form and change the window types (constant, variable, or main) if required.
9. Maintain **PARAGRAPH FORMATS** and **CHARACTER FORMATS** by selecting the appropriate button on the toolbar. At least one paragraph format should be created and assigned as the default paragraph format for the form. If no specific paragraph format is explicitly maintained for the text, then the system will use the default paragraph format.
10. Click the **PARAGRAPH FORMAT** button to create a paragraph format. At the bottom of the screen, enter a two-character name for the format and press **[Enter]**. Provide a short description (see [Figure 18.10](#)) and maintain other formatting information as required. You can maintain margins, alignment, font, tab spacing, and so on.

Select the **PAGE PROTECTION** checkbox if you want to ensure the paragraph doesn't break across pages. When this checkbox is selected, the system will automatically print the paragraph as a whole on the next page. If there isn't sufficient space on the page, instead of splitting the text between pages, you can create multiple paragraph formats.

Figure 18.10 Paragraph Format

- Once the default paragraph is defined, click the HEADER button (the hat symbol in [Figure 18.9](#) to the left of the PAGES button) in the toolbar (or press **F5**) to assign the default paragraph format for the form. On the FORM: CHANGE HEADER screen shown in [Figure 18.11](#), select BASIC SETTINGS to maintain the PAGE FORMAT, FIRST PAGE that should be called (if the form has multiple pages), default paragraph (DEFAULT PARAGR.), default font (FONT FAMILY), and FONT SIZE. Once these mandatory settings are maintained, the form can be activated. From the menu, select FORM • ACTIVATE to activate the form.

Figure 18.11 Change Header Screen

18.1.3 Maintaining Window Details

So far, we've created the required windows on the page and maintained the required settings to activate the form. In this section, we'll maintain the required details in each window. For this example, we'll print the company logo in the logo window, the billing document details in the info window, the payer details in the payer window, the item details in the main window, and the footer information in the footer window.

Let's work on each window step by step:

1. To print the logo in the logo window, you first need to upload the logo to the document server. To do so, open Transaction SE78 (Graphics Administration) and click the **IMPORT** button on the toolbar, as shown in [Figure 18.12](#). Provide the file path where the logo is saved, provide a name for the logo, and maintain a short description.

Select the appropriate radio button for **TYPE** based on whether the logo is in color or black and white. The logo file should be a bitmap image (BMP). Click **OK** to upload the logo to the document server.

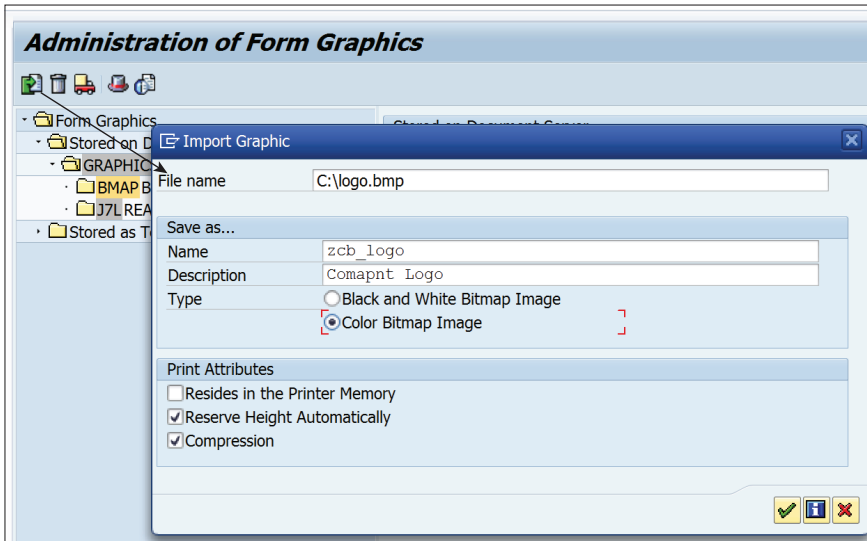


Figure 18.12 Graphics Administration

2. Once the logo is uploaded, go back to the form and click on **PAGE WINDOWS** in the toolbar to list all the windows of the page. Double-click the **LOGO** window

and select the TEXT ELEMENTS button in the toolbar. This will open the SAPscripts editor, as shown in [Figure 18.13](#).

The SAP script editor contains three columns: FORMAT, LEFT MARGIN, and ROW TEXT. The FORMAT and ROW TEXT columns are editable; the LEFT MARGIN column simply indicates if the text starts at the beginning of the line.

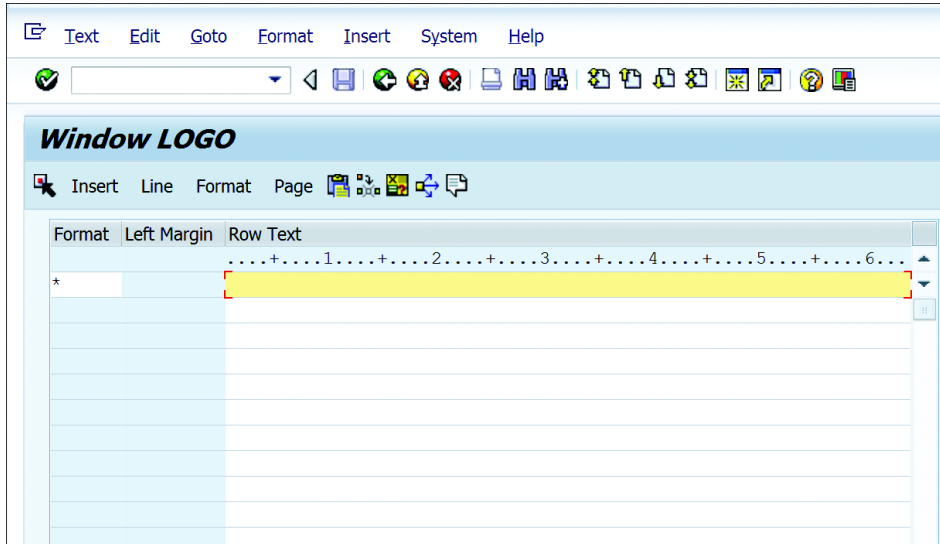


Figure 18.13 SAP Script Editor

3. In the FORMAT column, maintain the line format, such as the paragraph format. The system will use this information to format the output while printing. If the paragraph format is maintained, it will signify the beginning of a new paragraph, and all the lines immediately below it that have an empty value in this field will belong to this paragraph. The system will print these lines as continuous text.

Some of the possible entries in this field are listed in [Table 18.1](#).

Symbol	Name	Meaning
XX	PARAGRAPH FORMAT	Two-character paragraph format. The paragraph format can use the letters A to Z and numbers 0 to 9. The first character should be a letter.

Table 18.1 Formatting Options

Symbol	Name	Meaning
*	DEFAULT PARAGRAPH FORMAT	Entering "*" in this field will apply the default paragraph format.
/	NEW LINE	The line will be outputted on a new line. The formatting options of the paragraph will be retained.
/:	CONTROL COMMAND	Specifies a control command. The lines with this format are not outputted but treated as a control statement. Use this when using control commands such as IF statements.
/*	COMMENT LINE	A statement with this format won't be processed.
=	EXTENDED LINE	This format option signifies that the previous line is extended to this row. You can split your single text into multiple rows in the SAP script editor by using this symbol. All the extended lines will be treated as a single line when outputted.
/=	LIKE EXTENDED LINE	Similar to an extended line, but outputted in a new line.
(RAW LINE	No formatting will be applied, and the output is issued as is.
/(LIKE RAW LINE	Similar to a raw line, but printed in a new line.

Table 18.1 Formatting Options (Cont.)

In the ROW TEXT column, you can enter the text that should be printed in the output. You can use any control statement by entering "/:" as the format for the line. To print the value of a program variable, you need to preface and follow the variable name with an ampersand symbol (&)—for example, &V_MATNR&. If the & symbol isn't used, then the system will treat the variable name as text and print the variable name in the output, instead of printing its content.

4. You want to print a logo in this window, so insert the logo that you uploaded in the previous step by keeping the cursor in the ROW TEXT column and following the menu path INSERT • GRAPHICS. Here, select the STORED ON DOCUMENT SERVER tab and provide the name of the logo and its type, as shown in [Figure 18.14](#). Optionally, you can also specify the image resolution to be used. If the logo isn't already uploaded to the document server, you can click the IMPORT button on this window to upload the logo. Once the details are maintained, click OK to include the logo.

You can also type the BITMAP 'LOGO_NAME' OBJECT GRAPHICS ID BMAP TYPE BCOL DPI 200 command with /: as the format, because this is a control command. Once done, click BACK to go back to the PAGE WINDOWS screen.

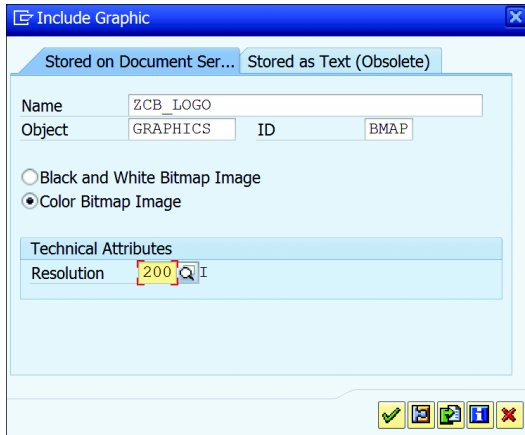


Figure 18.14 Insert Graphic

- Now, you need to print the billing document number, billing date, and net value in the info window. To do so, open the SAP script editor for the window and maintain the text as shown in Figure 18.15. Here, you're printing the field label and the field name that contains the data. In the driver program, you have to ensure that the `wa_vbrk` work area contains the data before this window is processed by the composer. (You'll see how to activate the composer and write the data when we enhance the driver program from Listing 18.1.) Note the different formats used for each line in the FORMAT column. Enter "*" to use the default paragraph format and "/" to print on a new line.

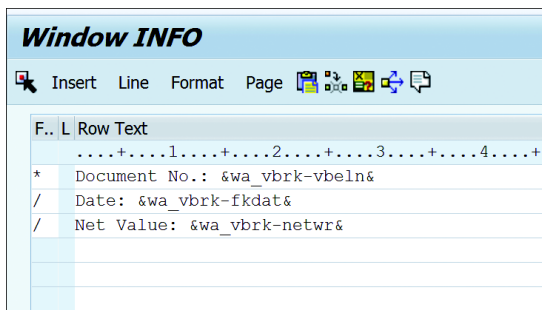


Figure 18.15 Printing Document Info

6. To print payer details, open the SAP script editor for the payer window and maintain the text as shown in [Figure 18.16](#). Here, you're printing the data from the `wa_knal` data object of the driver program.

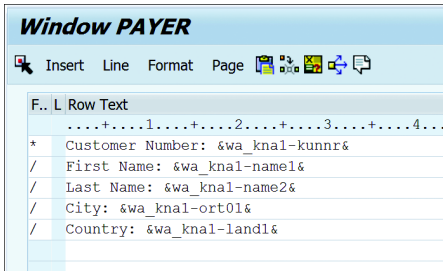


Figure 18.16 Payer Window

7. Print the item details of the document in the main window (window of type MAIN). A main window is divided into three areas: the TOP, BODY, and BOTTOM areas. When the MAIN window is called in the driver program, the composer automatically takes care of printing the text in the top area above the body. Once the body contents are outputted, the composer will print the text in the bottom area.

The top area and bottom area are triggered by the text in the body. If the body is empty, then the text in the top and bottom areas will not be printed. The composer will remember the contents of the top and bottom areas and will print them on all pages. Among other things, you can use the top area to print column headings and the bottom area to print subtotals. The top and bottom areas are specified in the form using the `/:TOP...ENDTOP` and `/:BOTTOM...ENDBOTTOM` control commands, as shown in [Figure 18.17](#).

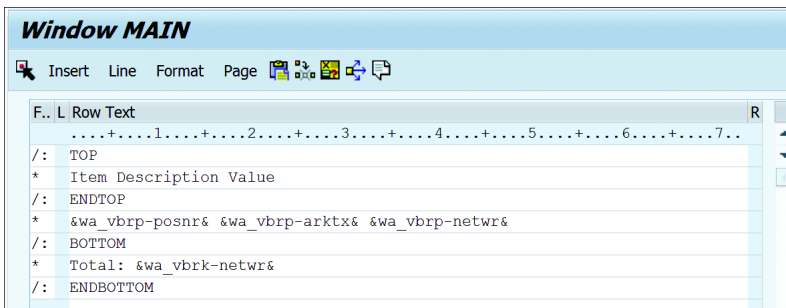


Figure 18.17 Main Window

8. Finally, input some text in the footer window. For the footer, print a disclaimer that can be edited by the user directly. When you maintain the text directly in the form, the user has to ask an ABAP programmer to edit the text each time he wants to modify the text. To let the user maintain the text directly instead, you can use standard text. Standard text is maintained separately using Transaction SO10 or by navigating to TOOLS • FORM PRINTOUT • SAPSCRIPT • STANDARD TEXT in the SAP MENU of the SAP EASY ACCESS screen.

To create a standard text, on the standard text initial screen, enter the TEXT NAME, TEXT ID, and the LANGUAGE (see [Figure 18.18](#)) and click CREATE. For standard text, the TEXT ID is ST. Clicking CREATE will open the SAP script editor (or PC editor, based on your personal settings) for you to maintain the text.

Standard Text		
Text Name	ZCA_TEXT	
Text ID	ST	Standard text
Language	EN	

Display Change Create

Figure 18.18 Standard Text Initial Screen

9. Once the standard text is maintained, you can include it in the form. To include the standard text, open the footer window and follow the menu path INSERT • TEXT • STANDARD TEXT. Enter the details of the standard text and click OK. You can also type the command `INCLUDE XXXX OBJECT TEXT ID ST LANGUAGE YY` directly in the ROW TEXT column. Here, XXXX is the standard text name and YY is the language.
10. Once all the windows are edited, you can perform a print test by selecting UTILITIES • PRINTING TEST to see how the layout will look when printed.

This completes the form development. Now, we need to process the form in the driver program, which we'll discuss next.

18.1.4 Processing Forms with Function Modules

To process a form, SAP provides various function modules. Three important function modules are `OPEN_FORM`, `WRITE_FORM`, and `CLOSE_FORM`, used to open the form for writing, write the data to the form, and close the form processing, respectively.

Let's go ahead and enhance the code from [Listing 18.1](#) to add the form processing logic to the `print_form` subroutine, as shown in [Listing 18.2](#). Here, we'll loop the item table `it_vbrp` that contains the item records. Within the loop, using the `AT FIRST` control break statement, we'll call the `OPEN_FORM` function module, to which the form name and language are supplied. We can also pass other information to this function module, such as print options.

```
*&-----*
*&      Form  PRINT_FORM
*&-----*
FORM print_form .
  LOOP AT it_vbrp INTO wa_vbrp.
    AT FIRST.
      CALL FUNCTION 'OPEN_FORM'
        EXPORTING
          form      = 'ZCA_FORM'
          language = sy-langu.
    ENDAT.

    CALL FUNCTION 'WRITE_FORM'
      EXPORTING
        type  = 'BODY'
        window = 'MAIN'.

  AT LAST.
    CALL FUNCTION 'CLOSE_FORM'.
  ENDAT.
ENDLOOP.
ENDFORM.
```

Listing 18.2 Code to Print Form

Once the form is opened, we call the `WRITE_FORM` function module to write the data to the form. To this function module, we pass the window name and the area (`TOP`, `BODY`, or `BOTTOM`). When the body of the main window is called, the composer will automatically call the top and bottom areas if they're maintained in the form. All the variables that are visible at this point in the program flow will be automatically transferred to the form. Finally, we close the form in the last loop pass by calling the `CLOSE_FORM` function module under the `AT LAST` control break statement.

Because we're calling the `MAIN` window within a loop, the complete text in the main window body will be executed in each loop pass. If you don't want certain text to be executed in each loop pass, then you can further divide the main window body into different sections using elements.

An element is maintained in the form using the format /E (see [Figure 18.19](#)) and is called from the driver program by passing the element name to the ELEMENT formal parameter of the WRITE_FORM function module, as shown in [Listing 18.3](#).

/E	ITEM
*	Item data
/E	TAX
*	Tax data

Figure 18.19 Using Elements

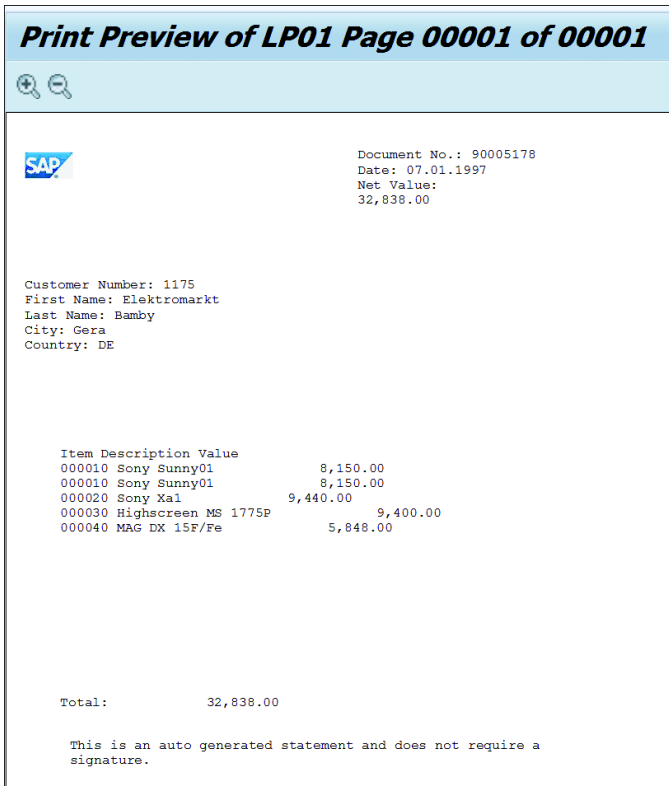
```
CALL FUNCTION 'WRITE_FORM'
  EXPORTING
    element = 'ITEM'
    type    = 'BODY'
    window = 'MAIN'.
```

Listing 18.3 Calling Element

We're now ready to execute the driver program and see the print preview. When the driver program is executed, the system will prompt you to provide the output device name, because you're not passing the printer information in the OPEN_FORM function module. Enter "LP01" (logical printer) as the output device and click the PRINT PREVIEW button, as shown in [Figure 18.20](#).

Figure 18.20 Print Dialog

The output should look similar to [Figure 18.21](#). As you can see, even though the form is printing all the data, the text isn't aligned properly.



Print Preview of LP01 Page 00001 of 00001

SAP

Document No.: 90005178
Date: 07.01.1997
Net Value:
32,838.00

Customer Number: 1175
First Name: Elektromarkt
Last Name: Bamby
City: Gera
Country: DE

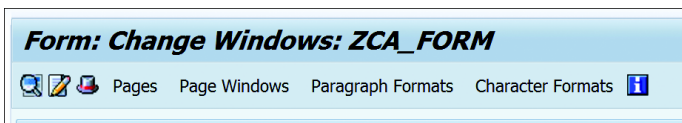
Item Description	Value		
000010 Sony Sunny01	8,150.00		
000010 Sony Sunny01	8,150.00		
000020 Sony Kai	9,440.00		
000030 Highscreen MS 1775P		9,400.00	
000040 MAG DX 15F/Fe		5,848.00	
Total:			32,838.00

This is an auto generated statement and does not require a signature.

Figure 18.21 Form Output

Let's align the text properly and also print the column headings in the main window in bold by using paragraph formats. Follow these steps:

1. Create a paragraph format in the form by selecting the PARAGRAPH FORMAT button in the toolbar (see [Figure 18.22](#)). Under TABS, maintain the tab positions as shown in [Figure 18.23](#).



Form: Change Windows: ZCA_FORM

Pages Page Windows Paragraph Formats Character Formats

Figure 18.22 Toolbar

Number	Tab position	Alignment
1	3.00 CM	LEFT
2	6.00 CM	LEFT
3	9.00 CM	LEFT

Figure 18.23 Maintaining Tab Positions

2. Click the FONT button and select ON for the BOLD option, as shown in [Figure 18.24](#). For this example, use this paragraph format for column headings.

	On	Off	Retain
Bold	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Italic	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Underlined	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>

Figure 18.24 Maintain Font for Paragraph Format

3. To create a new paragraph format for item records, enter the paragraph format name and press **Enter**.
4. The system will ask whether you want to create a new paragraph format or copy the previous paragraph format. Select COPY to use the same tab positions for item records as well. Make sure to turn off BOLD font for this copied element.
5. Once the paragraph formats are created, edit the main window text elements to use these paragraph formats, as shown in [Figure 18.25](#). To use the tab positions, insert a double comma (, ,) between words. Each double comma takes one tab position. Note that the net value field is printing leading spaces in [Figure 18.21](#). You can get rid of the leading spaces by using the formatting option (C) with the field, as shown in [Figure 18.25](#) for the wa_vbrp-netwr variable.

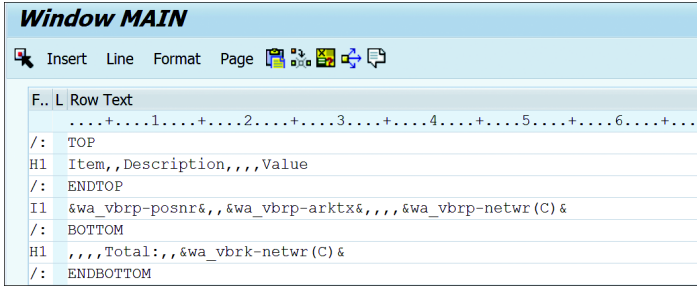


Figure 18.25 Using Tab Positions

Formatting Options

You can check all the available formatting options in the SAP Help portal at http://help.sap.com/saphelp_nw74/helpdata/en/4e/34031063de02c2e10000000a15822b/frameset.htm.

6. Similarly, edit the text elements of other windows to use tab positions and adjust the window widths if required. The final output should look similar to [Figure 18.26](#).

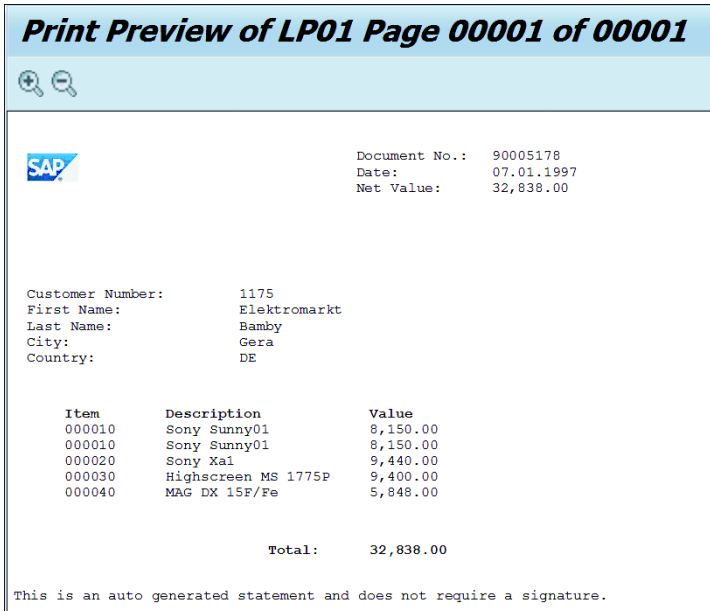


Figure 18.26 Final Formatted Output

You can format the item data as a table using the `BOX` control command where you can manually specify the positions. Similarly, you can format the address using the `ADDRESS` control command. As mentioned earlier, it's a lot of work to format SAPscripts forms, because you need to manually maintain the text positions and size, similar to classical reports that generate the list using `WRITE` statements. Visit http://help.sap.com/saphelp_nw74/helpdata/en/4e/37991b862135aae10000000a42189c/content.htm for information about all the control commands that can be used, including the `BOX` command and the `ADDRESS` command.

Forms are seldom triggered by executing the driver program directly. Instead, they're called from the application directly using the output type. Output types are set up by the functional consultant in which details such as transmission medium (print, EDI, etc.), processing routines, and so on are maintained.

We can configure the driver program and form for the output type via Transaction NACE. Proceed with the following steps:

1. In Transaction NACE, select the application area and click on the `OUTPUT TYPES` button in the toolbar (see [Figure 18.27](#)) to display all the output types available for the application area.

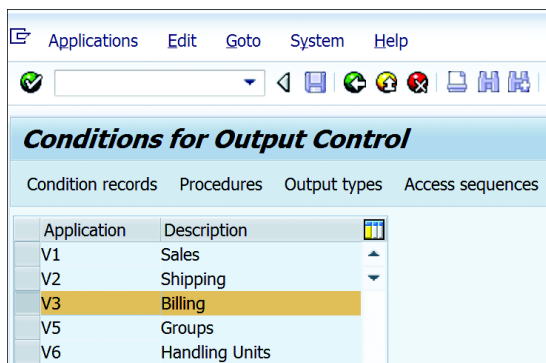


Figure 18.27 Output Types

2. Select the output type that should be configured, and double-click the `PROCESSING ROUTINES` folder, as shown in [Figure 18.28](#).

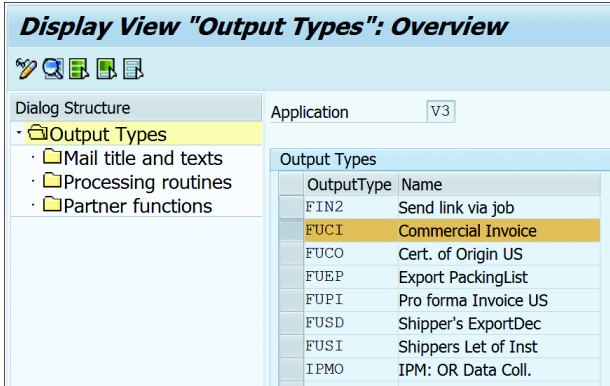


Figure 18.28 Display View of Output Types

3. Under the PROCESSING ROUTINES section, maintain the PROGRAM name, FORM ROUTINE name, and FORM name (see Figure 18.29). The form routine name should be the subroutine that acts as an entry point to the driver program. This subroutine should have two formal parameters—`return_code` and `us_screen`—as follows:

```
FORM entry_fuci USING return_code us_screen.
```

If there is any error during form processing in the driver program, the `return_code` parameter can be updated with a value of 1 to inform the calling program that the form processing resulted in an error. If the `return_code` is 0, the form is processed successfully. The `us_screen` parameter will have the value X if the form is called for print preview, in which case the spool doesn't need to be generated.

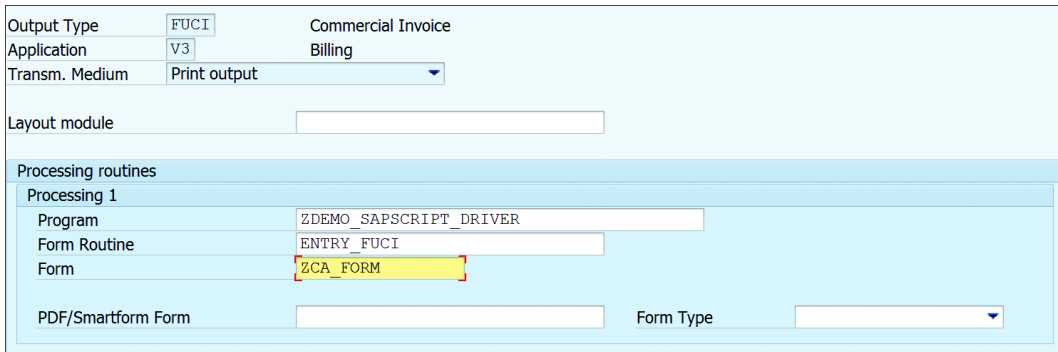


Figure 18.29 Configuring Output Type

- Once the document is processed using an output type in the application document, the processed output proposal is recorded in table `NAST`. Within the driver program, we can access the `NAST` structure by defining it using the `TABLES` statement. The `OBJKY` field will contain the document number.

Listing 18.4 shows the coding changes required to call the structure using the output type.

```
TABLES NAST.
FORM entry_fuci USING return_code us_screen.
  PERFORM fetch_data.
  PERFORM print_form.
ENDFORM.
*&-----*
*&      Form  FETCH_DATA
*&-----*
FORM fetch_data .
  SELECT SINGLE vbeln fkdat kunrg netwr FROM vbrk INTO wa_
vbrk WHERE vbeln EQ NAST-OBJKY.
ENDFORM.
```

Listing 18.4 Code Using NAST Structure

- In the application document, the output can be issued by configuring the output type in the document header. For example, for a billing document, open Transaction `VF02` and follow the menu path `GoTo • Header • Output` to maintain the output type. Once the output type is configured for the document, you can issue the output by following the menu path `Billing Document • Issue Output To`.

Now that we've looked at the older technology used for creating forms, in the next section we'll discuss a more recent form technology: smart forms.

18.2 Smart Forms

For all practical purposes, smart forms can be considered a successor to SAPscripts. *Smart forms* provide various advantages and address most of the challenges faced with SAPscripts.

The advantages of using smart forms over SAPscripts include the following:

- ▶ Smart forms reduce the development effort by more than 50 percent compared to SAPscripts.

- ▶ Smart forms provides a complete GUI to develop and adapt forms.
- ▶ Smart forms requires less programming knowledge, because the data from the SAP system can be configured directly. Thus, a user with good understanding of the tool and rudimentary programming knowledge can carry out modifications using SAP GUI.
- ▶ Various elements of the form can be easily recreated using copy and paste or drag and drop.
- ▶ Input elements can be integrated easily with HTML output, thus enabling a form to be used in Internet applications.
- ▶ It takes very little effort to create tables and format output with smart forms compared to SAPscripts.

However, as previously mentioned, SAPscripts will still be supported for years to come, and companies that have already invested in SAPscripts need not migrate existing SAPscripts to smart forms just yet. SAP also provides a migration tool if you wish to migrate existing SAPscript forms to smart forms. For all new developments, smart forms should be used.

We'll walk you through smart forms by developing the same form we used in the earlier example in [Section 18.1](#). This should help you appreciate the ease of development smart forms adds. We'll also use the same driver program, with certain changes to make it compatible with smart forms.

18.2.1 Overview and Layout

You build smart forms using the Form Builder tool. You can access Form Builder using Transaction SMARTFORMS or by navigating to TOOLS • FORM PRINTOUT • SMART FORMS in the SAP MENU on the SAP EASY ACCESS screen. Form Builder provides a GUI to develop and maintain smart forms.

When you develop a form in Form Builder and activate it, a function module is generated that can be called from any application program, similar to a regular function module. An interface can be defined for a smart form, and through that interface data can be supplied from the driver program.

The logic of the form can be mapped entirely in the smart form. Form Builder provides various tools to develop a smart form, such as Form Painter to design the layout, Table Painter to design table output, and so on. We'll explore all the various tools within Form Builder in the next few subsections.

The form logic is represented by a tree structure in which individual nodes consisting of various form elements are created. Form Builder is divided into three areas (see [Figure 18.30](#)):

► **Navigation area**

The navigation area displays the nodes in a tree structure. For example, in the navigation area you can define a table node to display the data in a tabular format, define an address node to display the data in a country-specific address format, and so on.

► **Maintenance area**

Once an element is selected in the node, its attributes can be maintained in the maintenance area. For example, once a table node is selected in the navigation area, the properties for the table node, such as number of columns, column width, column headings, and table data, can be maintained in the maintenance area.

► **Form Painter**

Form Painter allows you to design the layout of the page. Similar to SAPscripts, Form Painter lets you quickly design the page layout to position the windows using drag and drop.

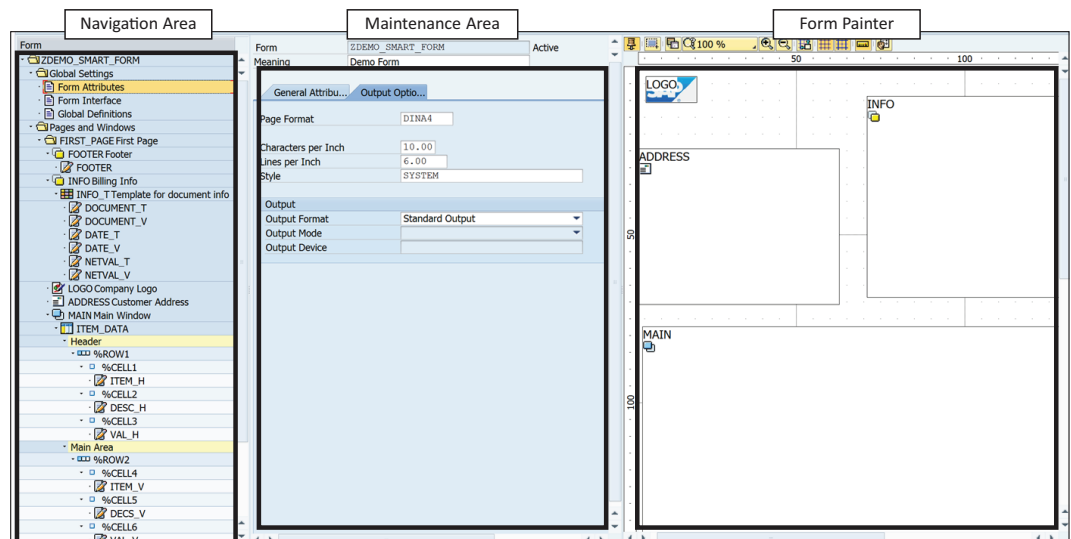


Figure 18.30 Form Builder

One of the major disadvantages of SAPscripts is its lack of a syntax checker. In Form Builder, you can easily check individual nodes or the complete form. You can also perform data flow analysis to check if all fields have defined values.

Similar to SAPscripts, smart forms consists of pages and windows. You can create multiple windows on a page, and each page can have different windows. We can design the page using Form Painter to position windows at different places on the page. We create form elements as nodes with attributes within a window. You also can create a draft page, which acts as a template from which print pages can be derived.

As in SAPscripts, you can create separate pages to represent the first page and subsequent pages. The first page typically has more windows to show data that doesn't need to be printed in the subsequent pages. The next page can be manually triggered from within the main window or called automatically when the data in the main window exceeds the length of the window.

You can create two types of windows with smart forms:

► **Secondary window**

A secondary window has a fixed length, and the text will be truncated if the length of the text exceeds the window height. You can have the same secondary window (with the same name) on multiple pages to print the same text on multiple pages.

► **Main window**

A form can have only one main window. A main window has a dynamic length, and the system continues printing in the next page if the text exceeds the window height. The main window triggers the page break automatically when the text exceeds the window height. The width of the main window should be same on all pages, but the height can vary.

Each smart form has certain global settings that should be maintained in areas such as the following:

► **FORM ATTRIBUTES**

Here, we maintain form attributes such as page format, default style, translations, and so on.

► **FORM INTERFACE**

Here, we define the interface of the form, via which the driver program can supply the data. The interface defined here forms the parameter interface of the

generated function module when the smart form is activated. Defining this interface is similar to defining the interface of a function module using Function Builder.

► **GLOBAL DEFINITIONS**

In this section, we define global variables or constants that can be used within the form. For example, if you defined an internal table in the form interface, then a work area for that internal table can be defined in the GLOBAL DEFINITIONS section, which then can be used to loop or read the internal table within the form nodes.

The GLOBAL DEFINITIONS section contains various tabs in which you can define local data types, field symbols, form routines, and so on. Any code to initialize the form can be written in the INITIALIZATION tab. If the structures defined in the interface or global definitions have currency or quantity fields, then the reference fields can be maintained in the CURRENCY/QUANTITY FIELDS tab.

We can create various elements in a smart form: text, tables, templates, addresses, graphics, and so on. Each element node has options that can be defined under the corresponding tab in Form Builder. Let's walk through how to work with the elements of a smart form while developing the form to meet our requirements. We'll use the same example as we used in the previous section in which we developed a SAPscripts form to print billing document information.

On the initial screen of Form Builder (Transaction SMARTFORMS), select the FORM radio button, enter the name of the form, and click CREATE.

18.2.2 Maintaining the Global Settings

As previously discussed, a smart form has three unique global settings that must be maintained: form attributes, form interface, and global definitions. In the following subsection, we'll discuss each of these settings.

Form Attributes

The FORM ATTRIBUTES section consists of the GENERAL ATTRIBUTES and OUTPUT OPTIONS tabs. Under the GENERAL ATTRIBUTES tab, we can maintain translations. In the OUTPUT OPTIONS tab, we maintain different output options, such as page format, default style, output format, and so on, as shown in [Figure 18.31](#).

Form	ZDEMO_SMART_FORM	Active
Meaning	Demo Form	
General Attribu... Output Optio...		
Page Format	DINA4	
Characters per Inch	10.00	
Lines per Inch	6.00	
Style	SYSTEM	
Output		
Output Format	Standard Output	
Output Mode	▼	
Output Device	▼	

Figure 18.31 Form Attributes

By default, the `STYLE` option is set to `SYSTEM`. You can define your own custom smart style and assign it to the form; you'll learn about creating smart styles in [Section 18.2.3](#). Paragraph formats and character formats are maintained in the style. Once the style is assigned to the form, we can use formats defined in the style in text nodes.

Form Interface

Under `FORM INTERFACE`, we define the interface parameters for the form. Defining the interface is similar to function modules in that we can maintain `IMPORTING PARAMETERS`, `EXPORTING PARAMETERS`, `TABLE PARAMETERS`, and `EXCEPTIONS`, as shown in [Figure 18.32](#).

Some default parameters are predefined in the interface; these control printing, archiving, mailing, and so on. For this example, we'll import the billing document header and item data into the form from the driver program. The interface parameters should be referred to global structures in the ABAP Data Dictionary.

For the header, we've defined a `ZBILL_HEADER` structure in ABAP Data Dictionary, as shown in [Figure 18.33](#). The structure consists of `VBELN`, `FKDAT`, `NETWR`, and `ADRNR` as the components. The `ADRNR` component will have the address number of the customer that we'll be fetching from the customer master table. This address number can be used in the form directly to print the customer data.

Parameter Name	Type Assignment	Associated Type
ARCHIVE_INDEX	TYPE	TOA_DARA
ARCHIVE_INDEX_TAB	TYPE	TSPDARA
ARCHIVE_PARAMETERS	TYPE	ARC_PARAMS
CONTROL_PARAMETERS	TYPE	SSFCTRLOP
MAIL_APPL_OBJ	TYPE	SWOTOBJID
MAIL_RECIPIENT	TYPE	SWOTOBJID
MAIL_SENDER	TYPE	SWOTOBJID
OUTPUT_OPTIONS	TYPE	SSFCOMPOP
USER_SETTINGS	TYPE	TDBOOL
IM_HEADER	TYPE	ZBILL_HEADER
IM_ITEM	TYPE	ZBILL_T_ITEM

Figure 18.32 Form Interface

Similarly, we've defined a ZBILL_T_ITEM table type in ABAP Data Dictionary, the line type for which has the components VBELN, POSNR, ARKTX, and NETWR for item data. In the form interface, we've defined a `im_header` structure, which is of type `zbill_header`, and an internal table `im_item`, which is of type `zbill_t_item` (see Figure 18.32).

Dictionary: Display Structure

Structure: ZBILL_HEADER Active
 Short Description: Billing Header

Attributes: Attribu... Compon... Entry help/check Currency/quantity fields

Component	Typing Method	Component Type	Data Type	Length	Deci...	Short Description
VBELN	Types	VBELN VF	CHAR	10		0 Billing Document
FKDAT	Types	FKDAT	DATS	8		0 Billing date for billing index and printout
NETWR	Types	NETWR	CURR	15		2 Net Value in Document Currency
ADRNR	Types	ADRNR	CHAR	10		0 Address

Figure 18.33 Header Structure

Global Definitions

Global variables and constants for the form are defined in GLOBAL DEFINITIONS. This section consists of multiple tabs to define objects such as variables, constants, field symbols, data types, and currency and quantity fields under their respective tabs. You can also maintain the initialization code under the INITIALIZATION tab to initialize the form fields. The FORM ROUTINES tab allows you to maintain subroutines that can be called from within the form. The definitions in this

section are global within the form and can be accessed from any element of the form. To define a variable, enter the variable name, type assignment, and associated type in the respective columns under the GLOBAL DATA tab (see [Figure 18.34](#)).

For this example, we'll be looping data in the internal table `im_item` and printing it in the main window. Because we need a work area to loop the internal table, define a work area under GLOBAL DEFINITIONS, as shown in [Figure 18.34](#). The work area should be of the line type of the internal table. To define a variable, enter the name

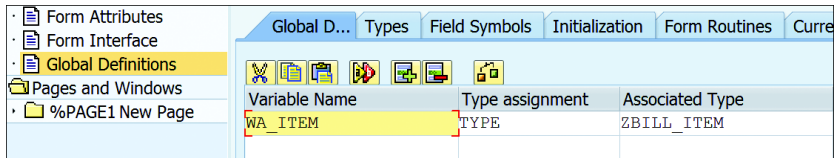


Figure 18.34 Defining Variable under Global Definitions Node

18.2.3 Maintaining Elements

As previously mentioned, we can create elements in a smart form, and each element node has options that can be defined under tabs in Form Builder.

You can maintain element attributes under the GENERAL ATTRIBUTES tab and output options under the OUTPUT OPTIONS tab. If you want the node to appear only when certain conditions are met, you can maintain the conditions under the CONDITIONS tab. Using the context menu on the node, you can create additional nodes and elements within the node.

In the following subsections, we'll discuss the different elements that can be maintained for smart forms.

Windows

Windows are sections on the page in which text and data can be positioned. A page can have multiple secondary windows but only one main window. You can adjust the size of the window or position the window using drag-and-drop functionality in Form Painter. Windows can overlap one another.

Under the PAGES AND WINDOWS folder, a default page with one main window is created by default. You can double-click the page to rename it.

To create a window on the page, right-click the page and select **CREATE • WINDOW** from the context menu, as shown in [Figure 18.35](#). Using the context menu, you can cut, copy, paste, or delete a node. Under the **CREATE** submenu, you have the options to create a new **PAGE**, **WINDOW**, **GRAPHIC**, or **ADDRESS**. We'll use these options one by one in following subsections.

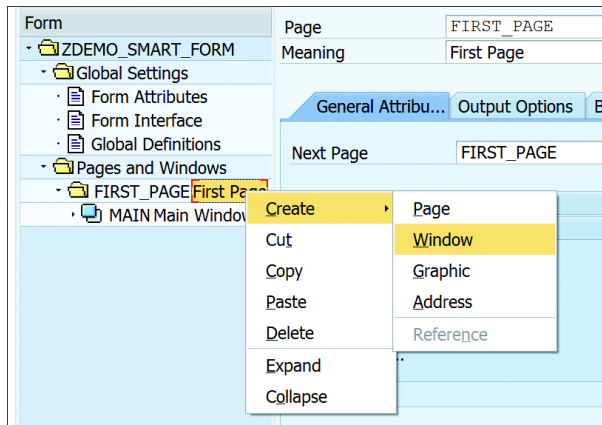


Figure 18.35 Context Menu for Page Node

At this point, the new window should appear in Form Painter towards the right (see [Figure 18.36](#)). If you don't see Form Painter, click the **FORM PAINTER** button in the toolbar. Form Painter opens in a split screen, which can be dragged to adjust the width.

You can rename the window to reflect the text it displays. For this example, we'll use this window to print the document information, so let's name it **INFO**. Maintain the window name in the **WINDOW** field and its short description in the **MEANING** field, as shown in [Figure 18.36](#). We also need one more window to print the footer text; go ahead and create one more window and position it below the main window in Form Painter.

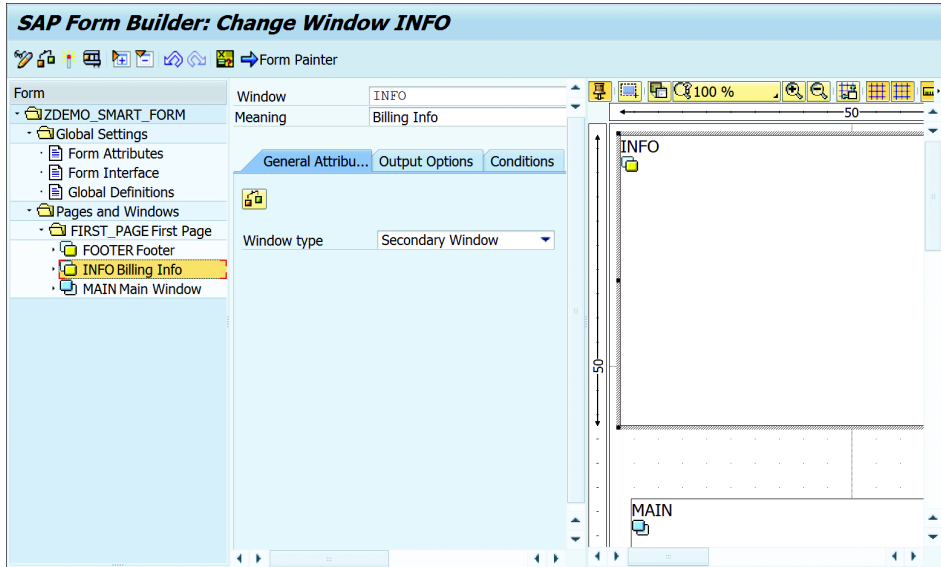


Figure 18.36 Window Creation

Graphics

Graphic elements allow you to quickly display images stored in the document server, such as a company logo. You can create a graphic element within a window or directly on the page.

To create a graphic element, from the context menu of the node, select **CREATE • GRAPHIC**. In this node, we'll display a company logo; therefore, let's name the node **LOGO**.

Under the **GENERAL ATTRIBUTES** tab, enter the logo details, as shown in [Figure 18.37](#). Here, we're using the same logo that we uploaded previously to the document server (see [Section 18.1](#)).

Optionally, you can use the **CONDITIONS** tab to restrict when this node is displayed. For example, if you're using the same form for multiple plants and want to print a different logo for each plant, then you can create multiple graphic nodes, and for each node you can maintain the plant number as the condition. In the **CONDITIONS** tab, you can compare a variable of the form with a value.

General Attributes	
Name	ZCB_LOGO
Object	GRAPHICS
ID	BMAP
<input type="radio"/> Black and White Bitmap Image (BMON) <input checked="" type="radio"/> Color Bitmap Image (BCOL) <input type="radio"/> Determine Dynamically (BMON, BCOL)	
Technical Attributes	
Resolution	200

Figure 18.37 Logo Attributes

Addresses

Address elements use Business Address Services (BAS) to print an address. This helps you avoid manually querying the address table to retrieve and print an address. The address element also automatically formats the address as per the postal regulations of the sender's country. In BAS, the address is identified by the address number, so all we have to do is enter the address number, either statically or dynamically. To dynamically enter the address number, include the variable name between ampersands (&).

For this example, we're passing the address number in the header structure of the form interface. We'll use this address number in the address node, as shown in [Figure 18.38](#).

Address	ADDRESS
Meaning	Customer Address
General Attributes	
Type	<input checked="" type="radio"/> Organization Address (1) <input type="radio"/> Personal Address (2) <input type="radio"/> Workplace Address (3) <input type="radio"/> Determine Dynamically (1,2,3)
Address Number	&IM_HEADER-ADRNR&
Person Number	
Additional Address Specifications	
Output Starts with Paragraph	<input checked="" type="checkbox"/>

Figure 18.38 Address Element

At this point, the page should have five windows, as shown in [Figure 18.39](#). Adjust the window positions in Form Painter as per the layout we used in [Section 18.1.1](#) (see [Figure 18.2](#)).

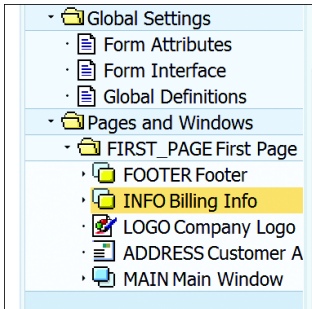


Figure 18.39 Nodes Inside Page

Templates

Templates allow you to format output in rows and columns. Using templates, we can display data in a tabular format. However, templates differ from table elements. Table elements are dynamic in length, whereas templates have a fixed length. We define the row type for a template in which individual column positions are maintained. We can define multiple rows for a template and link individual text, graphic, or address elements within the template to a specific row and column of the template.

To create a template, proceed with the following steps:

1. Right-click a node and select **CREATE • TEMPLATE** from the context menu.
2. In the template node, click the **DETAILS** button under the **TEMPLATES** tab to define the line type for the template (see [Figure 18.40](#)).
3. In the line type, define the number of columns in a line. The total width of the columns shouldn't exceed the width of the window. You can define the number of rows using the **FROM** and **TO** fields. If different rows need different column lengths, you can define multiple row types and assign each row type to a row by entering the row number in the **FROM** and **TO** fields. You can enter the same row number in both the **FROM** and **TO** fields to assign the row type to a single row.

4. You can define borders and shading for the template under the OUTPUT OPTIONS tab.

Name	From	To	Reference	Height	U.. 1.	U.. 2.	U.. 3.	U.. 4.	U.. 5.
R1	1	3		10.00	MM	3.00	CM	4.25	CM

Figure 18.40 Defining Row Type for Template

5. Once the template is defined, you can add multiple elements to the template. For each element in the template, assign the column and row number corresponding to where the element should be displayed in the template. You can assign the column and row numbers under the element's OUTPUT STRUCTURE area, as shown in [Figure 18.41](#).

Output structure	
Line	1
Column	1

Figure 18.41 Assigning Row and Column for Element within Template

Text

Text elements are used to display text in the form. We can display the value of a variable within the text by using the field name within ampersands. We can use four different text types within the text node: text elements, text modules, include texts, and dynamic texts.

Text can be maintained directly within the text element node or referred to a text module. *Text modules* are created separately by selecting the TEXT MODULE radio

button on the initial screen of Form Builder. Text modules allow you to reuse the text in multiple forms. *Include texts* allow you to include the standard texts that have been created for SAPscripts (as we used for the footer in the example in [Section 18.1](#)). *Dynamic texts* are directly passed in the form interface from the driver program.

With text elements, you can use the PC editor or the SAP script editor (based on your personal settings). When using the PC editor, click the INSERT FIELD button to insert a field (the field name should be maintained between ampersands). You also can use the SAP script editor as discussed in the previous section. You can open the SAP script editor by clicking the EDITOR icon (see [Figure 18.42](#)).

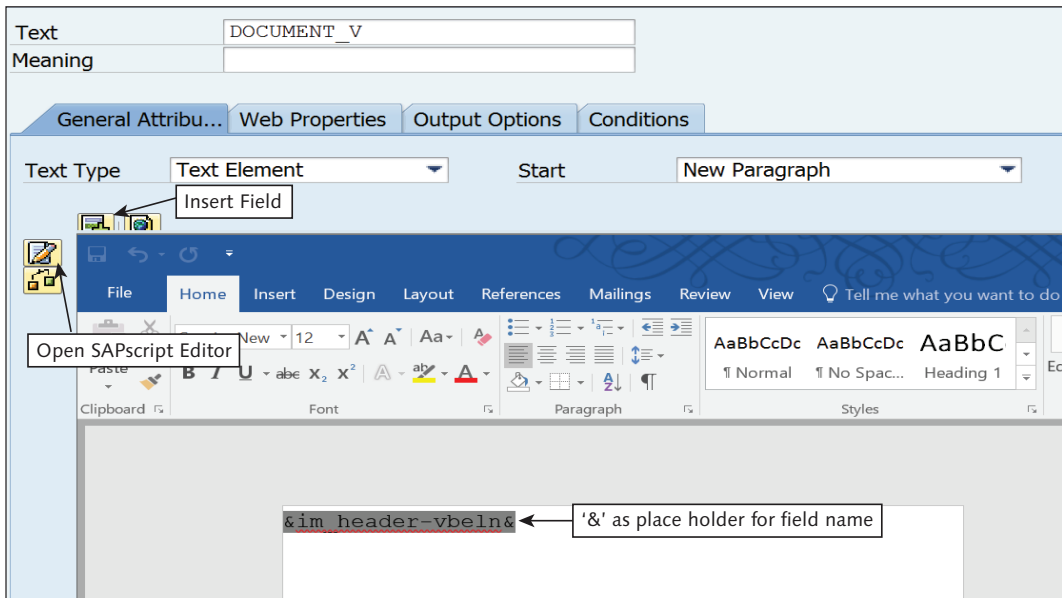


Figure 18.42 Text Element

For this example, we'll create six text elements within the template (see [Figure 18.43](#)), because we plan to include the print document number, document date, and net value with their respective labels in a two-column layout. For each text element, we'll assign the line number and column number of the template under the output options of the text element.

In this example, as shown in [Figure 18.43](#), the DOCUMENT_T text element contains the label, DOCUMENT_V contains the value, and so on.

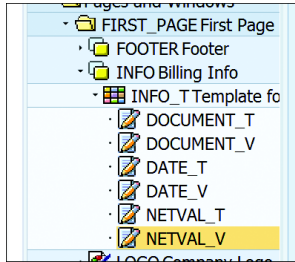


Figure 18.43 Text Nodes under Template

To create a text element, proceed with the following steps:

1. Right-click the template node and select **CREATE • TEXT** to create a text element under the template node.
2. Under the **GENERAL ATTRIBUTES** tab for the text element, you can choose the type of text you want to use, such as a text element, text module, include text, or dynamic text. The default type is text element, for which you can directly enter the text and insert fields.
3. Similarly, in the footer window, we plan to print the footer text maintained in a standard text that we created in the previous section. To include the standard text in the footer window, create the text element and select **INCLUDE TEXT** for the text TYPE. Enter the **TEXT NAME**, **TEXT OBJECT**, **TEXT ID**, and **LANGUAGE**, as shown in [Figure 18.44](#). The text object for standard text is **TEXT**.

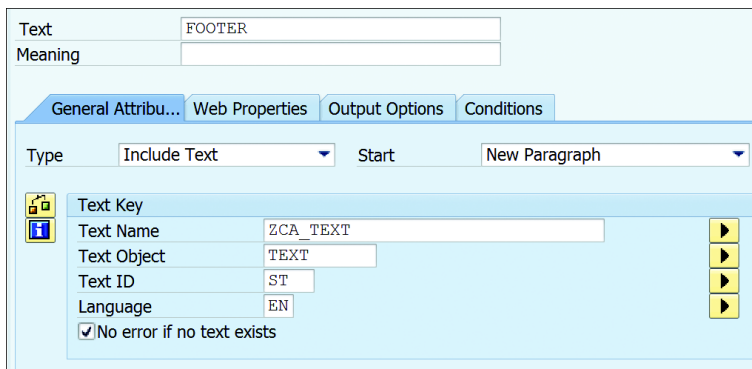


Figure 18.44 Including Standard Text

Table

You can use a *table* element you've created to print data from an internal table. You define a row type (or line type) for the table element similar to how it's defined for a template. In the row type, maintain the number of columns and their widths.

The table element consists of three areas (see [Figure 18.45](#)):

- ▶ **HEADER**
In the HEADER area, you can print the column headings.
- ▶ **MAIN AREA**
In the MAIN AREA, you can print the records of the internal table.
- ▶ **FOOTER**
The FOOTER area is where you can print any subtotals.

The table element consists of different tabs. In the TABLE tab, you define the line type by clicking the DETAILS button. In the DATA tab, you can loop the internal table. The DATA tab also provides support to mention a WHERE condition and sort criteria (see [Figure 18.45](#)). You can maintain borders and shading for the tables in the OUTPUT OPTIONS tab. You can perform certain calculations, such as totals, mean values, and count, under the CALCULATIONS tab.

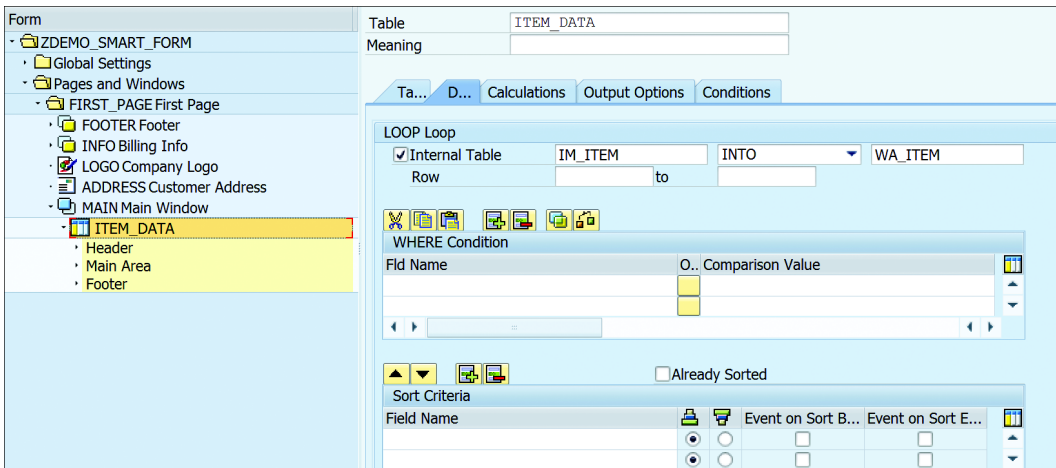


Figure 18.45 Table Element

To print text within a table, proceed with the following steps:

1. Right-click the relevant area (HEADER, MAIN AREA, or FOOTER) and select CREATE • TABLE LINE, as shown in [Figure 18.46](#).

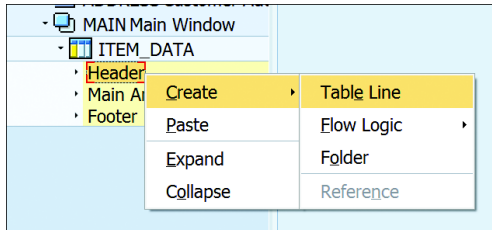


Figure 18.46 Create Table Line

2. Under the OUTPUT OPTIONS tab, select the LINE TYPE that was previously defined for the table (see [Figure 18.47](#)). The number of cells created within the node is based on the number of columns defined in the line type ([Figure 18.48](#)). In each cell, you can create the text element using the context menu to print the text or field.

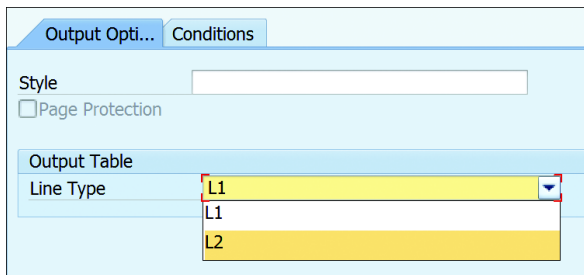


Figure 18.47 Table Line

3. Once you create the table lines for HEADER, MAIN AREA, and FOOTER, you'll see the respective cells added under each section, as shown in [Figure 18.48](#). You can use different line types for each of these sections depending on your requirements. For this example, we defined the line type L1 for the table with three columns and used the same line type for all three sections (header, main area, and footer), so you'll see three cells being added under each section.

Within each cell, create text elements to print the data. You're printing column headings in the header section, so maintain the text directly in the respective

text elements. For this example, enter "Item" in the text element of CELL1, "Description" in the text element of CELL2, and "Amount" in the text element of CELL3.

4. Print the internal table data in the main area. In order to print data from the internal table, loop the internal table into a work area under the DATA tab of the table node (see [Figure 18.45](#)). Within the text element of the cells in the main area, list the work area fields that should be printed. For this example, in the first cell of the main area, print the `WA_ITEM-POSNR` field; in the second cell of the main area, print the `WA_ITEM-ARKTX` field; and in the third cell of the main area, print the `WA_ITEM-NETWR` field.
5. Print the subtotals in the footer section. For this example, leave the first cell in the footer section empty to maintain the table alignment and print the text "Total" in the second cell and the `IM_HEADER-NETWR` field in the third cell. The `IM_HEADER` structure is defined in the form interface and will receive the data from the driver program when the form is called.

MAIN Main Window		
ITEM_DATA		
Header		
- %ROW1		
-	□	%CELL1
	·	ITEM_H
-	□	%CELL2
	·	DESC_H
-	□	%CELL3
	·	VAL_H
Main Area		
- %ROW2		
-	□	%CELL4
	·	ITEM_V
-	□	%CELL5
	·	DECS_V
-	□	%CELL6
	·	VAL_V
Footer		
- %ROW3		
-	□	%CELL7
-	□	%CELL8
	·	TOTAL_H
-	□	%CELL9 New Column 9
	·	TOTAL_V

Figure 18.48 Table Cells

Program Lines

Using the program lines element, you can write ABAP code within the form. This is useful if you want to perform simple calculations or query the database to fetch additional data. To create program lines, right-click a node and select **CREATE • FLOW LOGIC • PROGRAM LINES**. There is no restriction on what code you can write within program lines, but we recommend limiting the code to performing basic operations, such as executing conversion routines. The data selection for the form should always be delegated to the driver program instead of using program lines to fetch the data in the form.

The variables for which values are used within the program lines should be specified as input parameters for the program lines element. The variables for which values are changed within the program lines and should be available outside the program lines element must be specified as output parameters for the program lines element, as shown in [Figure 18.49](#). Here, we're calling the `CONVERSION_EXIT_MATN1_OUTPUT` function module to convert the material number in `WA_ITEM-MATNR` to an external format.

The converted value is copied to `V_MATNR`. In this example, we're maintaining `WA_ITEM-MATNR` as the input parameter for the program lines element as it's being used inside the program lines. The value in `WA_ITEM-MATNR` doesn't change after the code is executed. However, because we're changing the value in `V_MATNR`, we're maintaining this field as the output parameter for the program lines element.

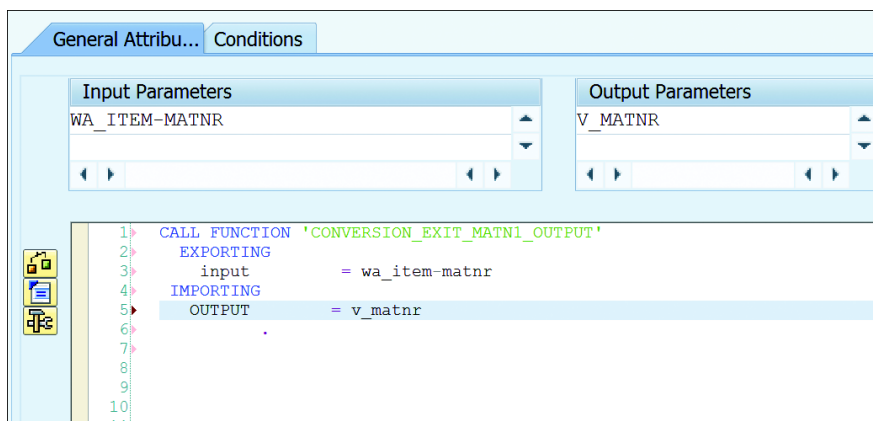
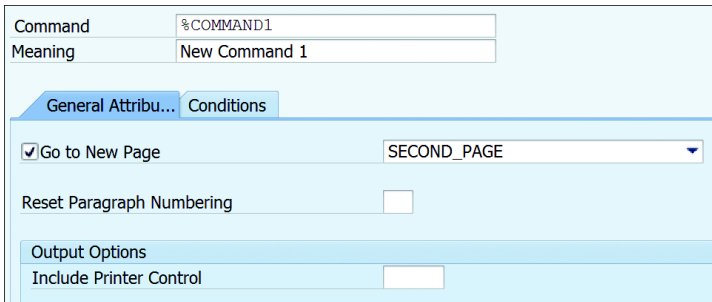


Figure 18.49 Program Lines

Commands

The *command node* is used to call a new page or to execute other special commands such as numbering, printer control, and so on. This node is mostly used in the main window to trigger a page break, as shown in [Figure 18.50](#).

To create a command node, right-click the node in which you want to insert the command node and select **CREATE • FLOW LOGIC • COMMAND**.



Command	%COMMAND1
Meaning	New Command 1
<div style="display: flex; border-bottom: 1px solid black;"> <div style="border-right: 1px solid black; padding: 2px;">General Attribu...</div> <div style="padding: 2px;">Conditions</div> </div>	
<input checked="" type="checkbox"/> Go to New Page	SECOND_PAGE
Reset Paragraph Numbering	<input type="checkbox"/>
<div style="border: 1px solid black; padding: 2px;">Output Options</div>	
Include Printer Control	<input type="checkbox"/>

Figure 18.50 Command Node

Loop

You can use a loop node to process the nodes within the loop node repeatedly. Similarly to a table node, you can loop an internal table with the loop node and access its data within the elements of the node. The table node can have a layout defined that isn't available in the loop node. To create a loop node, right-click the node in which you want to insert the loop node and select **CREATE • FLOW LOGIC • LOOP**.

You can have a table node within a loop node as an inferior node and vice versa. This allows you to loop multiple tables that are linked with a foreign key as a nested loop. For example, if you want to loop the item table for each record of the header table, you can define a loop node within the header table node, as shown in [Figure 18.51](#).

Loop: %LOOP1
Meaning: New Loop 1

Calculations | Events | Output Options | Conditions

LOOP Loop

Internal Table: IT_ITEM INTO WA_ITEM
Row: _____ to _____

WHERE Condition

Fld Name	O.. Comparison Value
VBELN	WA_HEADER-VBELN

Sort Criteria

Already Sorted

Field Name	Event on Sort B...	Event on Sort E...
	<input checked="" type="radio"/> <input type="radio"/>	<input type="checkbox"/> <input type="checkbox"/>
	<input checked="" type="radio"/> <input type="radio"/>	<input type="checkbox"/> <input type="checkbox"/>
	<input checked="" type="radio"/> <input type="radio"/>	<input type="checkbox"/> <input type="checkbox"/>
	<input checked="" type="radio"/> <input type="radio"/>	<input type="checkbox"/> <input type="checkbox"/>

Figure 18.51 Loop Node

Styles

Styles allow you to define your own paragraph and character formats. A style is defined independently of the form, and you can use one style in multiple forms. The following steps walk you through defining a style:

1. On the initial screen of Form Builder (Transaction SMARTFORMS), select the STYLE radio button and enter the name of the style, as shown in [Figure 18.52](#). Click CREATE.
2. The style consists of three nodes: HEADER DATA, PARAGRAPH FORMATS, and CHARACTER FORMATS, as shown in [Figure 18.53](#). In the header data, you maintain the default paragraph format and other settings that will be used as defaults if no paragraph format or character format is explicitly assigned to the text.

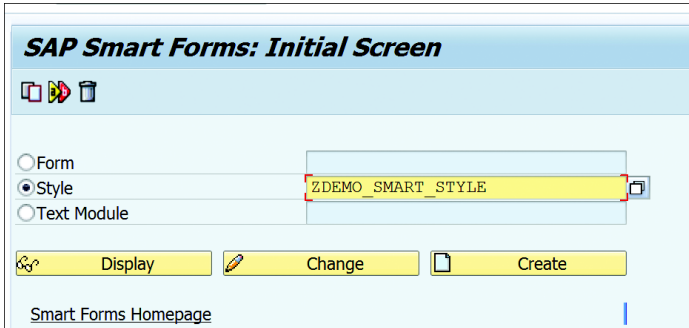


Figure 18.52 Form Builder Initial Screen

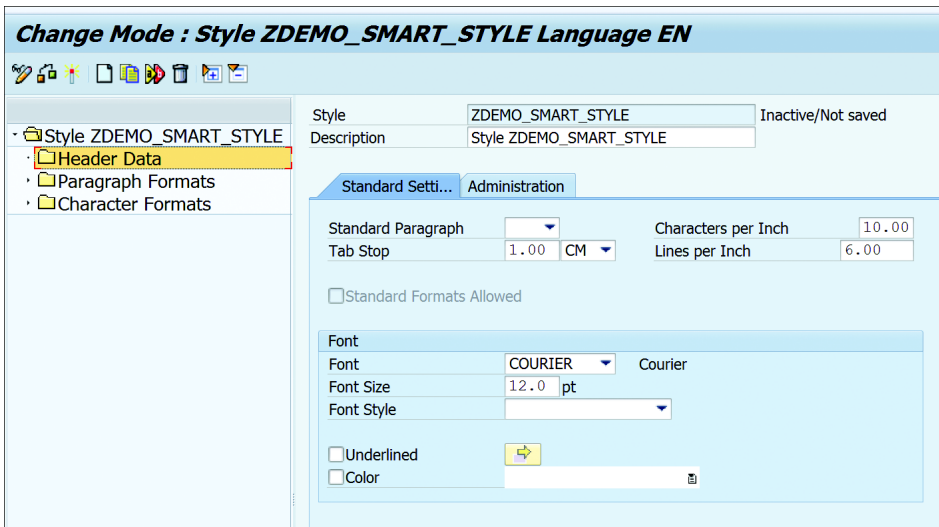


Figure 18.53 Style Change Screen

3. You can create any number of paragraph and character formats within the style. To create a paragraph format, right-click the PARAGRAPH FORMATS node and select CREATE NODE from the context menu, as shown in [Figure 18.54](#).
4. In the CREATE NODE window (see [Figure 18.55](#)), enter the paragraph format name and click the green checkmark (or press).

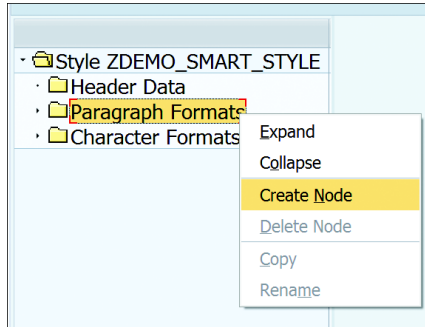


Figure 18.54 Create Paragraph Format

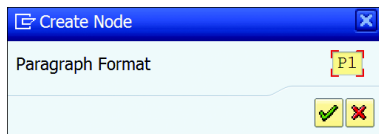


Figure 18.55 Create Node

5. In the PARAGRAPH FORMAT window (see [Figure 18.56](#)), you can maintain various details, such as indentation and spacing, fonts, tab positions, and so on. All of the options are self-explanatory. The paragraph format can be assigned to the text in a smart form, which will then derive the formatting from the assigned paragraph format. We'll explain how to use the paragraph format in just a moment.

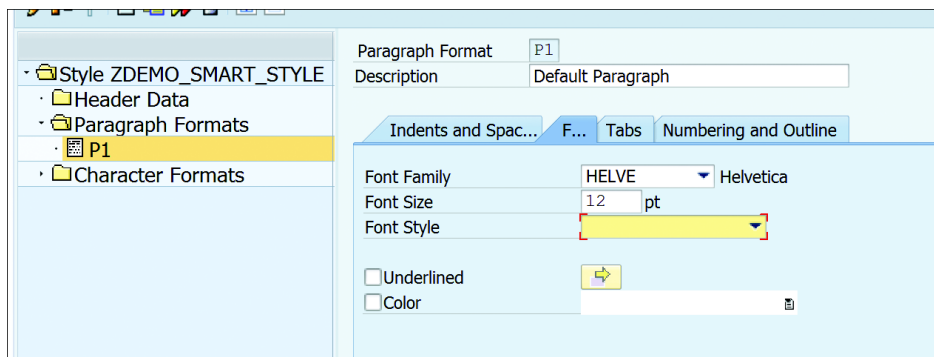


Figure 18.56 Maintain Paragraph Format

6. Similarly, to create character formats right-click the CHARACTER FORMATS node and select CREATE NODE from the context menu. For character formats, you can maintain formatting that is applied to specific words within the text. The paragraph format is used for the complete paragraph or line, whereas the character format is used for words within a line.

Figure 18.57 shows the CHARACTER FORMAT screen in which you can maintain the required font.

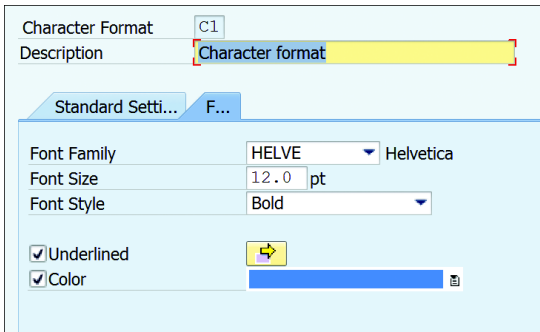


Figure 18.57 Character Format

7. Once you define at least one paragraph format, assign the default paragraph format for the style in the HEADER DATA node by selecting one of the paragraph formats from the dropdown list for the STANDARD PARAGRAPH field, as shown in Figure 18.58. Once the default paragraph format for the style is assigned, click the ACTIVATE button in the toolbar to activate the style.

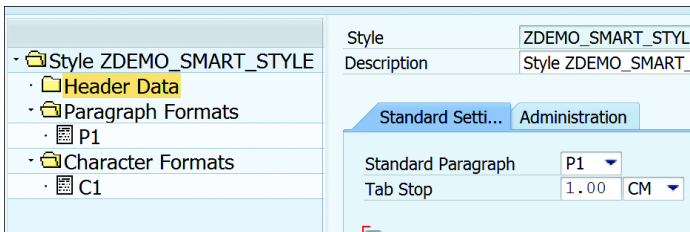


Figure 18.58 Assigning Default Paragraph

Now that you know how to create custom styles and maintain paragraph and character formats, let's look at how to use these custom styles in a smart form.

You can assign the style for a smart form under FORM ATTRIBUTES by entering the name of the style in the STYLE field, as shown in [Figure 18.59](#). This style will be used for the entire smart form.

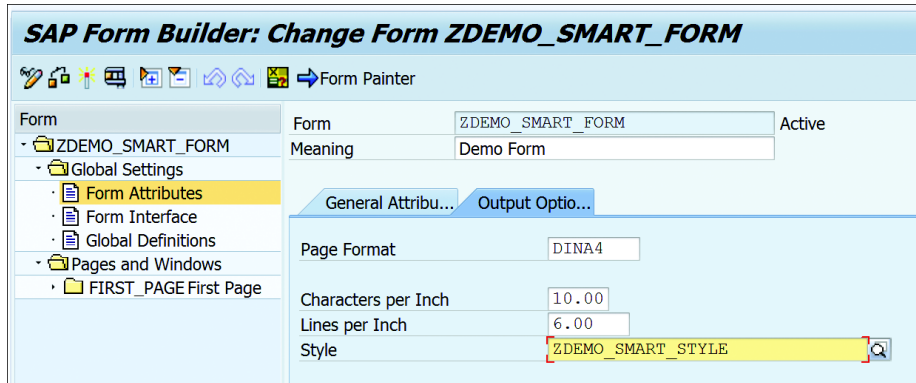


Figure 18.59 Assigning Style to Smart Form

If you want to use different styles for different nodes, you can maintain the style for the node under the node's OUTPUT OPTIONS tab, as shown in [Figure 18.60](#).

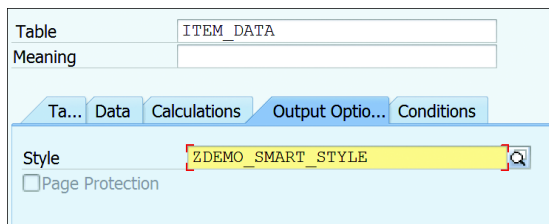


Figure 18.60 Maintaining Style for Node

To use the paragraph format and character format from the style, open the TEXT ELEMENT node and switch to the SAP script editor by clicking the EDITOR button (see [Figure 18.61](#)). In the SAP script editor, you can use the formats similar to those used in SAPscripts. In the first column, you maintain the paragraph format that is applied to the complete line. To use the character format, use the character format between the <> and </> tags, as shown in [Figure 18.61](#).

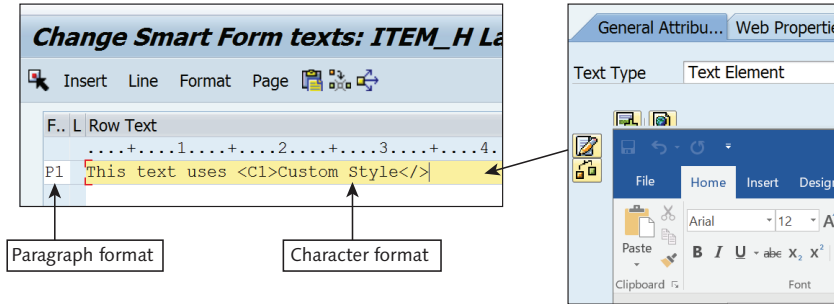


Figure 18.61 Using Paragraph and Character Formats

18.2.4 Driver Program

Once all the nodes of the form are defined, you can activate the form by clicking the **ACTIVATE** button in the toolbar. Once the form is activated, it generates a function module, which can be called from the driver program. You can check the name of the generated function module via the menu path **ENVIRONMENT • FUNCTION MODULE NAME**.

We don't generally call a smart form directly by the function module name but instead call it dynamically using the `SSF_FUNCTION_MODULE_NAME` function module. This is important, because the function module name generated by the smart form is dynamic. When this smart form is transported to another system, the other system can generate a totally different name for the function module. If you hard-code the generated function module name in your driver program, it may not work in the new system after transportation.

[Listing 18.5](#) shows the code in the driver program. The data selection logic is similar to that in the SAPscripts driver program, but we're calling the smart form as a function module directly in the driver program instead of using `open_form`, `write_form`, or `close_form` function modules to print the form.

```
REPORT zdemo_smart_driver.
TYPES : BEGIN OF ty_vbrk,
         vbeln TYPE vbeln_vf,
         fkdat TYPE fkdat,
         kunrg TYPE kunrg,
         netwr TYPE netwr,
       END OF ty_vbrk.

TYPES : BEGIN OF ty_vbrp,
         vbeln TYPE vbeln_vf,
```

```

        posnr TYPE posnr,
        arktx TYPE arktx,
        netwr TYPE netwr,
    END OF ty_vbrp.

TYPES : BEGIN OF ty_kna1,
        kunnr TYPE kunnr,
        adrnr TYPE adrnr,
    END OF ty_kna1.

DATA : wa_vbrk   TYPE ty_vbrk,
        wa_kna1  TYPE ty_kna1,
        it_vbrp  TYPE zbill_t_item,
        wa_vbrp  TYPE zbill_item,
        st_header TYPE zbill_header.

PARAMETERS p_vbeln TYPE vbeln_vf.

START-OF-SELECTION.
    PERFORM fetch_data.
    PERFORM print_form.

*&-----*
*&      Form  FETCH_DATA
*&-----*
FORM fetch_data .
    SELECT SINGLE vbeln fkdat kunrg netwr FROM vbrk INTO wa_
vbrk WHERE vbeln EQ p_vbeln.
    IF sy-subrc IS INITIAL.
        SELECT SINGLE kunnr adrnr FROM kna1 INTO wa_kna1 WHERE kunnr EQ wa_
vbrk-kunrg.
        SELECT vbeln posnr arktx netwr FROM vbrp INTO TABLE it_
vbrp WHERE vbeln EQ wa_vbrk-vbeln.
    ENDIF.
MOVE-CORRESPONDING : wa_vbrk to st_header, wa_kna1 to st_header.
ENDFORM.
*&-----*
*&      Form  PRINT_FORM
*&-----*
FORM print_form .
    DATA fm_name TYPE RS38L_FNAM.
    CALL FUNCTION 'SSF_FUNCTION_MODULE_NAME'
        EXPORTING
            formname           = 'ZDEMO_SMART_FORM'
        IMPORTING
            FM_NAME           = fm_name.

CALL FUNCTION fm_name
    EXPORTING

```

```

    im_header          = st_header
    im_item            = it_vbrp
EXCEPTIONS
  FORMATTING_ERROR    = 1
  INTERNAL_ERROR      = 2
  SEND_ERROR          = 3
  USER_CANCELED       = 4
  OTHERS              = 5

```

```

IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.
ENDFORM.

```

Listing 18.5 Smart Form Driver Program

In [Listing 18.5](#), we're consolidating the header data from tables `vbrk` and `kna1` into the `st_header` structure. This allows us to keep the form interface simple and easy to read, as opposed to passing multiple structures.

In the `print_form` subroutine, we're calling the `SSF_FUNCTION_MODULE_NAME` function module, to which we're passing the smart form name; the function module will return the corresponding generated function module name for the smart form. We're then calling this function module dynamically and passing the header and item data.

[Figure 18.62](#) shows the smart form output.

Print Preview of LP01 Page 00001 of 00001

Archive Print and Archive

SAP

Document No: 90005178
Date: 07.01.1997

Firma
Elektromarkt Bamby
Adlerstrasse 452
D-07545 GERA

Net Value: 32,838.00

Item	Description	Amount
000010	Sony Sunny01	8,150.00
000020	Sony Xa1	9,440.00
000030	Highscreen MS 1775P	9,400.00
000040	MAG DX 15F/Fe	5,848.00
	Total	32,838.00

This is an auto generated statement and does not require a signature.

Figure 18.62 Smart Form Output

As you may have noticed, we created the form quite easily without having to write any code or spend any time formatting the output.

In this section, we walked through how to use smart forms. In the next section, we'll look at using SAP Interactive Forms by Adobe.

18.3 SAP Interactive Forms by Adobe

SAP, in partnership with Adobe, introduced SAP Interactive Forms by Adobe in 2005 with the release of SAP NetWeaver 2004. The partnership aims to integrate Adobe's design expertise into the SAP NetWeaver platform. The integration takes advantage of the feature-rich Adobe LiveCycle Designer tool, which enables users to create form layouts, and combines it with SAP's expertise in form development from SAPscripts and smart forms. While Adobe continues to improve the core technology, SAP focuses on improving the integration and the frameworks based on the new technology.

To work with Adobe forms, you need the SAP system to be configured with Adobe Document Server (ADS). Adobe LiveCycle Designer and Adobe Acrobat Reader should also be installed on your system. Adobe Acrobat Reader is available as a free download from Adobe's website, but Adobe LiveCycle Designer needs to be downloaded from the SAP Service Marketplace at *service.sap.com*.

We won't discuss the technical configurations to set up ADS or Adobe LiveCycle Designer here, because it's beyond the scope of this chapter. For more information on ADS configuration, visit <http://scn.sap.com/docs/DOC-52256>. In this section, we'll focus on form development using Adobe Forms. If you haven't practiced developing smart forms in the previous section yet, we highly recommend that you do so before proceeding with Adobe forms. This should help you follow the steps in this section more easily, because many of the tasks involved with Adobe forms are similar to those for smart forms. After all, smart forms were used as the outline for integrating Adobe forms with SAP applications.

We'll cover the following high-level steps in this section to develop Adobe forms:

1. Create an interface.
2. Map the context for the form.
3. Design the form layout.

The form consists of the context and layout; the interface is defined separately. The interface is used in the form context to map the form fields.

We'll use the same billing document example we've used throughout this chapter to develop an Adobe form.

18.3.1 Form Interface

Unlike smart forms, in which the form interface is part of the form and is defined from within the form, with Adobe forms we define the interface separately. This allows you to use the same interface for multiple forms. The where-used list on the interface shows all the forms that use the interface. This helps determine all the dependent forms that need to be adapted when the interface is extended.

The form interface must be defined before we can create the form. Proceed with the following steps to create the interface:

1. To use Object Navigator to create the interface, select **CREATE • FORM OBJECT • INTERFACE** from the context menu within the package. Alternately, you can use Form Builder. [Figure 18.63](#) shows the initial screen of Form Builder.

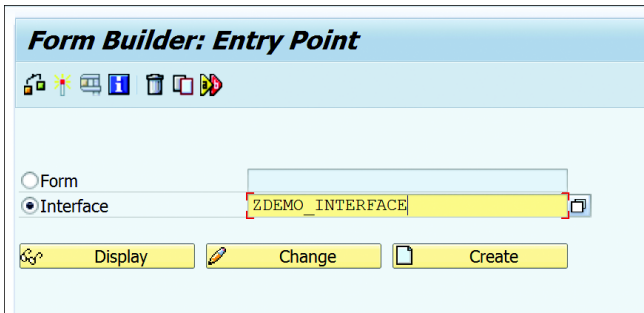


Figure 18.63 Form Builder Initial Screen

2. On the **FORM BUILDER: ENTRY POINT** screen, select the **INTERFACE** radio button, provide a name for the interface, and click the **CREATE** button.
3. You'll see the **CREATE INTERFACE** window, as shown in [Figure 18.64](#). In this window, provide a short description and select the **INTERFACE TYPE**.

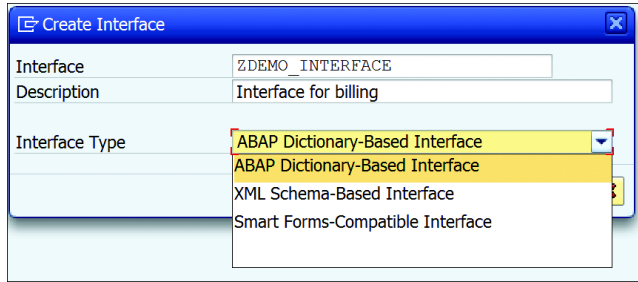


Figure 18.64 Create Interface Window

There are three types of interfaces you can create:

- ▶ **ABAP DICTIONARY-BASED INTERFACE:** This is the most frequently created interface. With this interface, you define the interface parameters by referring to the ABAP Data Dictionary objects, such as structures, table types, and data elements. This interface allows you to define the import parameters, export parameters, and exceptions (similar to a function module), as shown in [Figure 18.65](#). Similar to a smart form, you can also define global definitions, initialization code, and currency/quantity fields. We'll use this interface for this example.

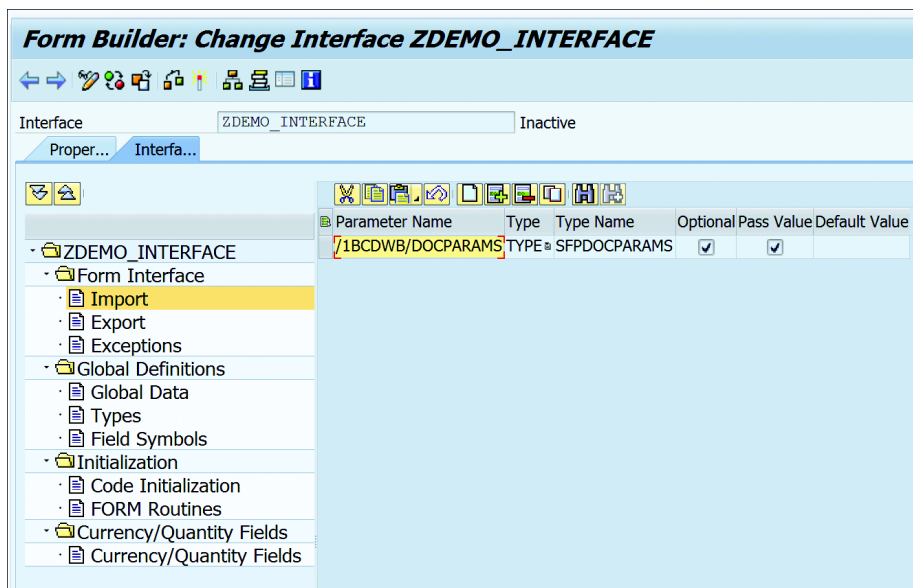


Figure 18.65 ABAP Dictionary-Based Interface

- ▶ **XML SCHEMA-BASED INTERFACE:** This interface is used when the application processes XML data instead of ABAP Dictionary data—for example, with Web Dynpro ABAP, which defines its own data context. The XML schema-based interface definition looks similar to [Figure 18.66](#). In the XML SCHEMA SOURCE field, you see two options, FILE and GENERATED. You can't use the GENERATED option; it's used when the interface is generated using an API from a Web Dynpro ABAP application. When manually creating an interface, select the FILE option only.

In the FILE field, provide the path of the XML file and click the LOAD button to load the file.

The XML schema, import/export parameters, and exceptions are displayed in the INTERFACE tab. However, you can't make any changes.

The screenshot shows the SAP configuration screen for an interface. At the top, the interface name is 'ZDEMO_INTERFACE1' and its status is 'Inactive'. Below this, there are two tabs: 'Proper...' and 'Interface'. The 'Interface' tab is active. The main area is divided into several fields: 'Description' (a text area with a yellow highlight), 'Interface Type' (a dropdown menu set to 'XML Schema-Based Interface'), 'XML Schema Source' (a dropdown menu set to 'File'), and 'File' (a text input field). A yellow 'Load' button is positioned to the right of the 'File' field.

Figure 18.66 XML Schema-Based Interface

- ▶ **SMART FORMS-COMPATIBLE INTERFACE:** This interface type allows you to define an interface that is compatible with applications that have used smart forms previously. The interface defines the parameters that are similar to smart forms; for example, in [Figure 18.67](#), you can see that the TABLES option is added to the FORM INTERFACE node and to the default importing parameters of the smart form.

This allows you to replace the smart form called with an Adobe form in the application program without having to change the interface parameters.

4. For this example, select ABAP DICTIONARY-BASED INTERFACE. The interface, as shown in [Figure 18.65](#), consists of a tree structure on the left, which is used for navigation between different definitions. The right-hand side provides the work area to define the parameters, global definitions, and so on. As you can

see in [Figure 18.65](#), there is a default /1BCDWB/DOCPARAMS parameter defined of type SFPDOCPARAMS. The SFPDOCPARAMS structure consists of various components, as shown in [Figure 18.68](#).

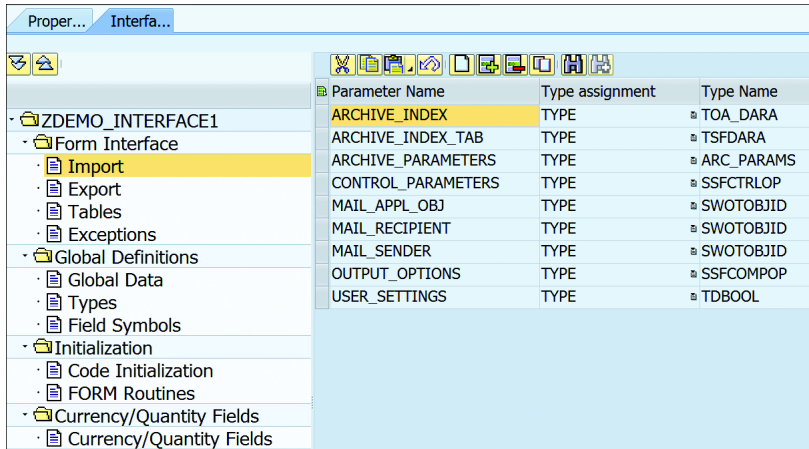


Figure 18.67 Smart Forms-Compatible Interface

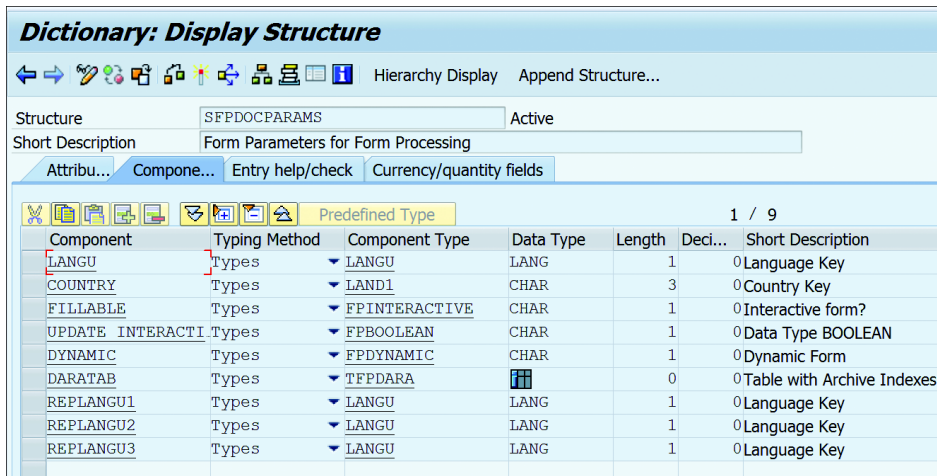


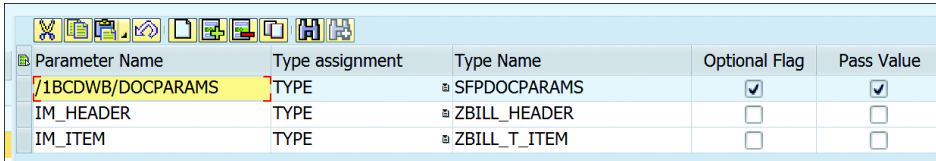
Figure 18.68 Components of Structure SFPDOCPARAMS

[Table 18.2](#) explains the components of the SFPDOCPARAMS structure.

Component	Meaning
LANGU	Form language
COUNTRY	The country key along with language, used for formatting output
FILLABLE	Flag to indicate if the form is to be created as an interactive form
DYNAMIC	Used when form is interactive
DARATAB	Archiving information
REPLANGU1	Alternate language if the form is not available in the requested language
REPLANGU2	Another alternate language
REPLANGU3	Another alternate language

Table 18.2 Component of Structure SFPDOCPARAMS

5. The PARAMETERS section provides a toolbar on top with buttons to CREATE, INSERT, DELETE, CUT, COPY, PASTE, and SEARCH parameters. In the example form, import the header and item data from the driver program, similar to the example used in [Section 18.2](#). To define the importing parameters, click the CREATE button and maintain the parameters as shown in [Figure 18.69](#).



Parameter Name	Type assignment	Type Name	Optional Flag	Pass Value
1BCDWB/DOCPARAMS	TYPE	SFPDOCPARAMS	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
IM_HEADER	TYPE	ZBILL_HEADER	<input type="checkbox"/>	<input type="checkbox"/>
IM_ITEM	TYPE	ZBILL_T_ITEM	<input type="checkbox"/>	<input type="checkbox"/>

Figure 18.69 Importing Parameters

You can't define any export parameters manually. However, you can define your own exceptions. This interface will be used to generate the interface of the generated function module when the form is activated.

- Under GLOBAL DEFINITIONS, define the global data objects for the form, local type definitions, and field symbols (see [Figure 18.70](#)).
- Under INITIALIZATION, write the code to initialize the form fields. You can also write the code to fetch any additional data that is required but not supplied by the driver program. However, we recommend delegating all data selection to the driver program. The FORM ROUTINES section allows you to maintain sub-routines, just like in a regular ABAP program.

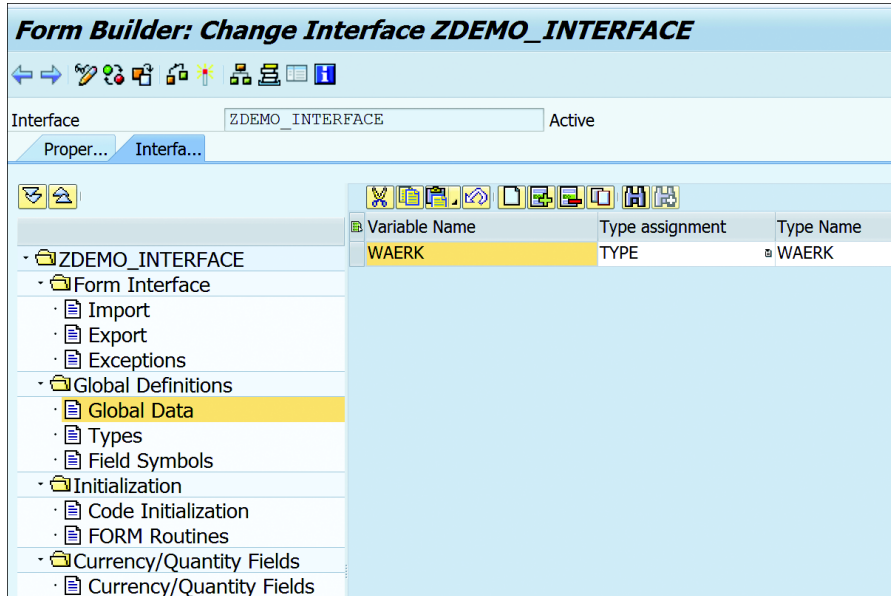


Figure 18.70 Global Definitions

8. The CURRENCY/QUANTITY FIELDS section (see [Figure 18.71](#)) allows you to define reference fields for currency and quantity fields. The reference fields allow the form to apply output formatting for the fields. For example, it may show decimals when printing the currency in US dollars, but when printing Japanese Yen, it might suppress the decimals. If the reference fields exist in the same structure for the parameters or global definitions, the program will use that structure automatically.
9. If the reference fields use an external reference, then you need to manually define the reference fields in the interface. For example, the `IM_HEADER` importing parameter in the form interface contains a `NETWR` currency field. In the structure, you're using an external reference from table `VBRK` for this field, as shown in [Figure 18.72](#). Because the reference field doesn't belong to the same structure, `ZBILL_HEADER`, you need to manually define reference fields in the form interface. This is also true for the `IM_ITEM` parameter for the interface.

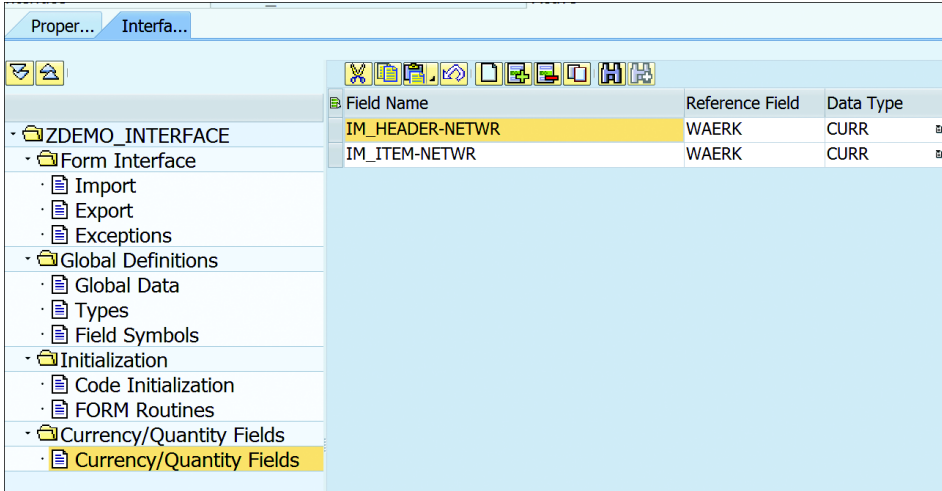


Figure 18.71 Currency/Quantity Fields I

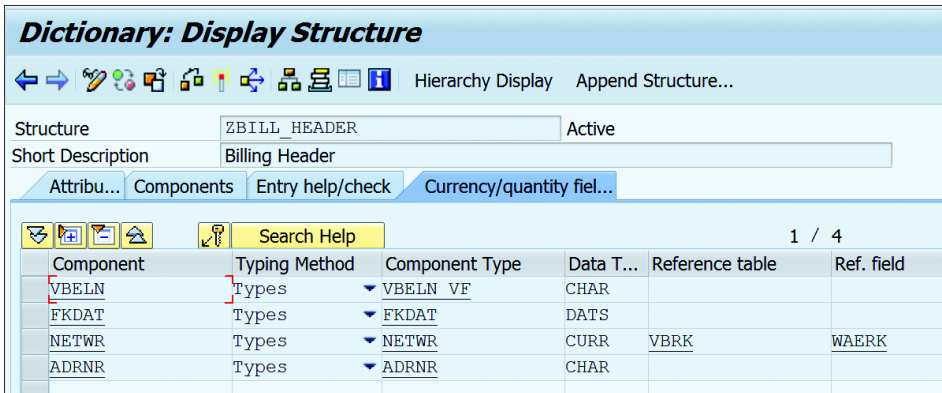
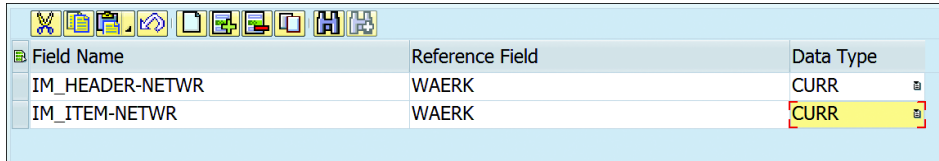


Figure 18.72 External Reference to Currency Field

- To maintain the reference fields, first define a variable of type `CUKY` in `GLOBAL DATA` (see [Figure 18.70](#)) and then use it as a reference field in the `CURRENCY/QUANTITY` fields section, as shown in [Figure 18.73](#). For this example, we defined a `WAERK` global variable of type `WAERK`.

In the `CURRENCY/QUANTITY FIELDS` section, use this global variable as the reference field for the `NETWR` field in both the header and item structure. Under `DATA TYPE`, select `CURR` or `QUAN` based on the type of field.



Field Name	Reference Field	Data Type
IM_HEADER-NETWR	WAERK	CURR
IM_ITEM-NETWR	WAERK	CURR

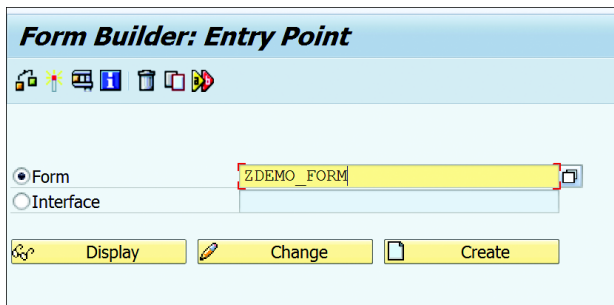
Figure 18.73 Currency/Quantity Fields II

Once all the required information is maintained, activate the interface.

18.3.2 Form Context and Layout

Now that we've defined the interface, we can create the form linking the interface. Proceed with the following steps:

1. On the Form Builder initial screen, select the FORM radio button, enter the name of the form, and click the CREATE button, as shown in [Figure 18.74](#).

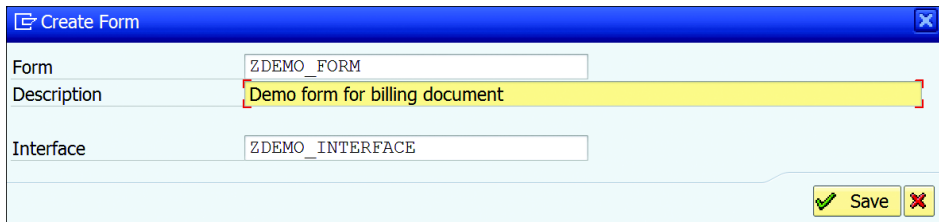


Form Builder: Entry Point

Form Builder interface showing the 'Form' radio button selected and the form name 'ZDEMO_FORM' entered in the text field. Below the text field are three buttons: 'Display', 'Change', and 'Create'.

Figure 18.74 Form Builder: Entry Point

2. In the CREATE FORM window, provide a short description and the interface name you just created, as shown in [Figure 18.75](#).



Create Form

Form: ZDEMO_FORM

Description: Demo form for billing document

Interface: ZDEMO_INTERFACE

Buttons: Save, X

Figure 18.75 Create Form

This will open the form in which you can map the context and design the layout, as shown in [Figure 18.76](#). The screen contains three tabs:

- ▶ **PROPERTIES:** The PROPERTIES tab shows general information about the form, such as the form interface name, who it's created by, its original language, and so on.
- ▶ **CONTEXT:** The CONTEXT tab allows you to map the interface components with the context.
- ▶ **LAYOUT:** The LAYOUT tab is where you design the form. You need Adobe LiveCycle Designer installed and configured with SAP GUI to work in the LAYOUT tab. The Adobe LiveCycle Designer patch for SAP GUI can be downloaded and installed from SAP Service Marketplace at <https://websmp107.sap-ag.de/>. Note that you'll need download authorizations to download any SAP software from SAP Service Marketplace.

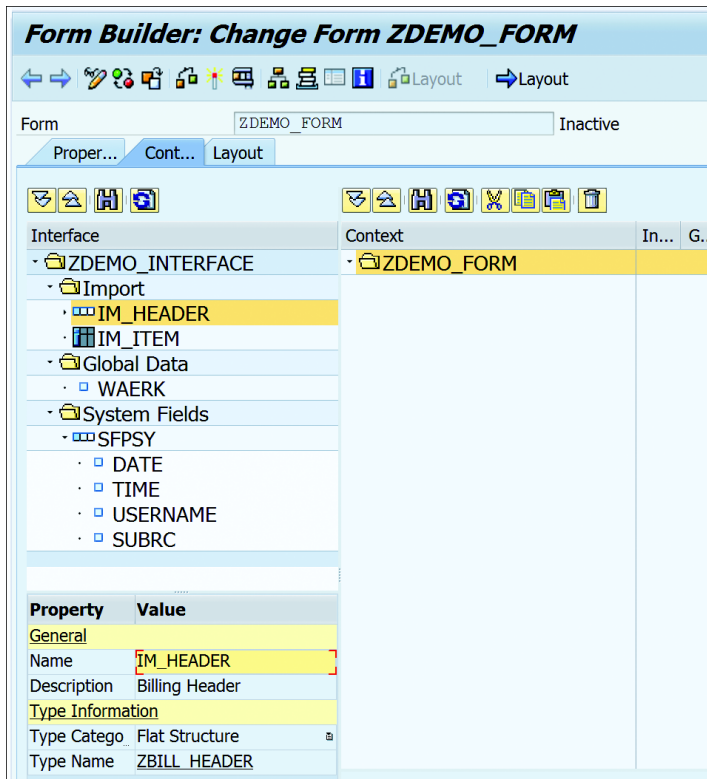


Figure 18.76 Form Editor

In the following subsections, we'll look at using first the CONTEXT tab and then the LAYOUT tab.

Context

The CONTEXT tab is divided into two areas: On the left-hand side is an interface that includes the parameters and global data defined in the interface. It also includes the SFPSY structure, which consists of the DATE, TIME, USERNAME, and SUBRC system fields. They are initialized at runtime and can be used in the form—for example, to print the current date, time, and so on. You can double-click any component to see more information about the component, such as its data type.

On the right-hand side is the context. By default, the context is empty and shows an empty parent folder. This parent folder acts as the root folder under which we can define various nodes. All the nodes that can be added to the context are more or less similar to the smart form. For example, similar to smart forms, we can add graphic nodes, table nodes, address nodes, and so on.

The easiest way to add nodes is to drag the field/structure/table from the interface and drop it in the context. Another way to add the node is to right-click on the root folder or existing node in the context and select the node from the context menu, as shown in [Figure 18.77](#). As you can see, most of the nodes are similar to those for a smart form.

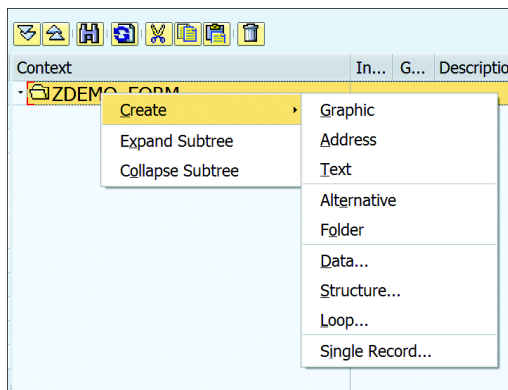


Figure 18.77 Adding Nodes to Context

Let's go through each node, one-by-one, while developing the form to meet the example requirements.

Graphic

A graphic node allows you to print a logo or any image, such as a scanned copy of a signature, for example. To add and edit a graphic node, follow these steps:

1. Right-click the root folder and select **CREATE • GRAPHIC**. This will add the GRAPHIC node under the root folder, as shown in [Figure 18.78](#).

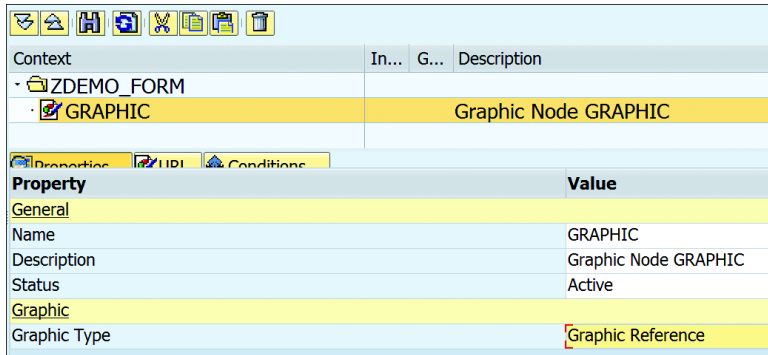


Figure 18.78 Graphic Node

2. When you select a node, you'll see the **PROPERTY** window at the bottom of the screen. In the **PROPERTY** window, you can maintain the properties for the node. For all nodes, except the alternatives node, a **CONDITIONS** button appears at the top of the window. The **CONDITIONS** button allows you to conditionally display the node, similar to the **CONDITIONS** tab for the nodes in smart forms.
3. In the **PROPERTY** window for the **GRAPHIC** node, you can maintain the **NAME** for the node and the **DESCRIPTION**. The **STATUS** field can be set to **ACTIVE** or **INACTIVE**. If you no longer want to use the node or want to temporarily deactivate it, set it to **INACTIVE**.
4. The **GRAPHIC TYPE** field allows you to set how graphics are processed. Here, you can select from one of the two options. You can load a graphic from a URL by selecting the **GRAPHIC REFERENCE** option or select the **GRAPHIC CONTENT** option to map a field (of type `string` or `xstring`) that contains binary data about the graphic. If a field of type `string` is used, the data should be base 64 encoded.
5. The **GRAPHIC REFERENCE** option is selected by default. To provide the URL when this option is selected, click the **URL** button at the top of the properties window and enter the URL of the image, as shown in [Figure 18.79](#).

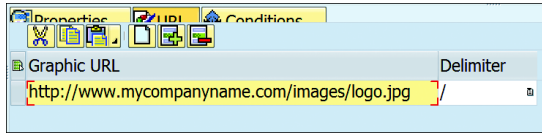


Figure 18.79 Logo URL

6. If you want to make some part of the URL dynamic, you can create one more row with an appropriate delimiter selected under the DELIMITER field. For example, if all the images are stored in the *images* folder and you want to assign the image name dynamically at the end, you can do so by entering the static part of the URL in the first row and maintaining the field name that contains the image name in the second row, with a forward slash (/) as the delimiter, as shown in [Figure 18.80](#).

In [Figure 18.80](#), the `v_image` field contains the name of the image, which can be updated dynamically in the form logic.

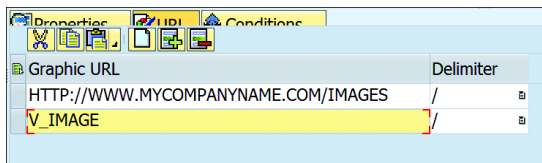


Figure 18.80 Dynamic URL

You can also upload the image to the form while designing the layout, which we'll discuss in the next section.

Address

To create and edit the address node, proceed with the following steps:

1. Select **CREATE • ADDRESS** from the context menu. The address node is similar the address node in a smart form. With the address node, you can use BAS, which stores and formats the address.
2. Based on the address type, you need to assign the address number and/or the person number. If the address type is **ORGANIZATION**, **COMPANY**, then you need to assign the address number. If the address type is **PRIVATE ADDRESS** or **OFFICE ADDRESS**, then you need to assign both the address number and the person number. The system will automatically take care of formatting the address as per the postal requirements of the location of the address.

3. For this example, assign the `ADRNR` field of the `IM_HEADER` structure to the ADDRESS NUMBER, as shown in [Figure 18.81](#). You also need to assign the `SENDING COUNTRY`.

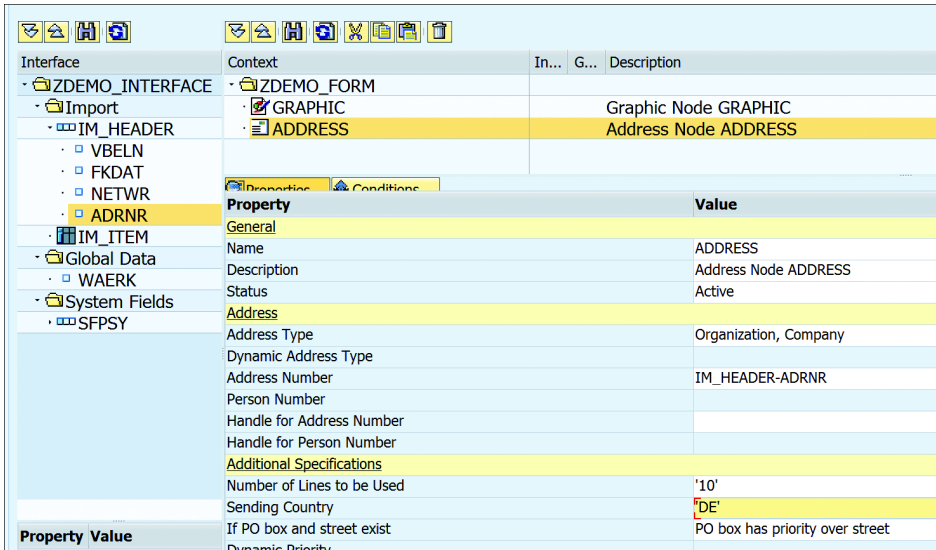


Figure 18.81 Address Node

Structure

The easiest way to add a structure node is to drag the structure from the interface and drop it in the context. Another way is to select `CREATE • STRUCTURE` from the context menu of the root folder. This will open a window in which you can input the structure name and click OK. The structure will be added to the context, as shown in [Figure 18.82](#).

By default, all the fields of structure will be active. If you don't want to use a field of the structure, then you can right-click that field and deactivate it from the context menu. The system won't use deactivated fields at runtime, and no data is transferred to them. This is useful if you added a structure with many fields but only want to use certain fields in the form. This will save the memory used by the form. You'll see a lock symbol for each field of the structure. This symbol indicates that the fields of the structure can't be deleted independently; you only can make them inactive.

Context	In...	G...	Description
· ZDEMO_FORM			
· GRAPHIC			Graphic Node GRAPHIC
· ADDRESS			Address Node ADDRESS
· IM_HEADER			Billing Header
· VBELN		🔒	Billing Document
· FKDAT		🔒	Billing date for billing index and printou
· NETWR		🔒	Net Value in Document Currency
· ADRNR		🔒	Address

Property	Value
General	
Name	IM_HEADER
Description	Billing Header
Status	Active
Structure	
Field	IM_HEADER

Figure 18.82 Structure Node

LOOP

To add an internal table, you can either drag and drop the internal table from the interface to the context or create a loop node manually. Selecting CREATE • LOOP will open a window in which you can enter the internal table name. Once the internal table is added, it'll be displayed in the context as shown in [Figure 18.83](#). A loop node allows you to show tabular data in a form, and you can nest multiple tables using a loop node.

Context	Item Table Type
· IM_ITEM	
· DATA	
· VBELN	🔒 Billing Document
· POSNR	🔒 Item number of the SD document
· ARKTX	🔒 Short text for sales order item
· NETWR	🔒 Net Value in Document Currency

Property	Value
General	
Name	IM_ITEM
Description	Item Table Type
Status	Active
Loop	
Table	IM_ITEM
From Line	
To Line	
Already sorted	

Figure 18.83 Loop Node

For example, if you have two tables with a foreign key relationship, then you can loop one table inside another and use the `WHERE` condition to link the foreign key relationship. A loop node provides buttons at the top of the `PROPERTIES` window to specify the `WHERE` condition and sort sequence.

Data

A data node allows you to add a single field to the context. To create a data node, you can either drag and drop the field from the interface to the context or select `CREATE • DATA` from the context menu of the root folder/node.

Text

A text node allows you to display text in a form. Similar to smart forms, you can use a text module, include text, or dynamic text with the text node by selecting the appropriate type in the `TEXT TYPE` field, as shown in [Figure 18.84](#). Based on the text type selected, the screen will show the appropriate fields to input the text details. The text object name should be maintained between single quotes.

Property	Value
General	
Name	TEXT
Description	Text Node TEXT
Status	Active
Text	
Text Type	Text Module
Text Module	
Text Name	
Text Language	

Figure 18.84 Text Node

Alternative

An alternative node allows you to check for a specific condition and accordingly assign an alternative value. It consists of two subnodes: true and false. When you create an alternative node, you set the condition for the node by clicking the `ALTERNATIVE CONDITIONS` button, as shown in [Figure 18.85](#). If the condition is true, the node assigned to the true node is used; otherwise, the node inside the false node is used.

In [Figure 18.85](#), the condition we're checking is whether `IM_HEADER-FKDAT` is initial. If the condition is true, then we print the system field `DATE`; otherwise, we print the `FKDAT` field of `IM_HEADER`. You can set multiple conditions by using `AND` or `OR` as an operator. You can also use other relation operators, such as `=` (equal to), `>` (greater than), `?` (not equal to), and so on.

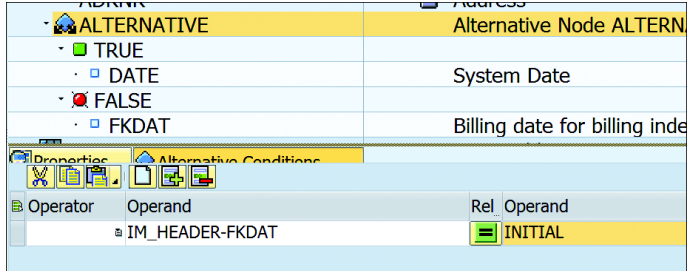


Figure 18.85 Alternative Node

Single-Record

The single-record node can be used within a loop node to read a single record of another internal table. This is similar to using a `READ` statement within the `LOOP` statement. When you add a single-record node, you specify the `WITH KEY` condition by clicking the `WITH KEY CONDITIONS` button, as shown in [Figure 18.86](#). We can use the `SUBRC` field of system structure `SFPSY` to check if the `READ` operation was successful.

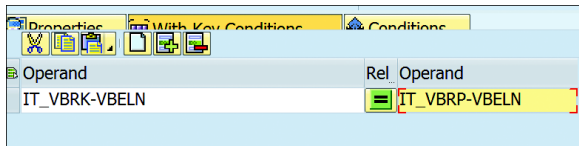


Figure 18.86 With Key Condition for Single-Record Node

This concludes our discussion of context. Once the context is designed, click the `LAYOUT` tab, discussed ahead, to design the layout of the form.

Layout

The `LAYOUT` tab loads Adobe LiveCycle Designer and consists of three sections, as shown in [Figure 18.87](#).

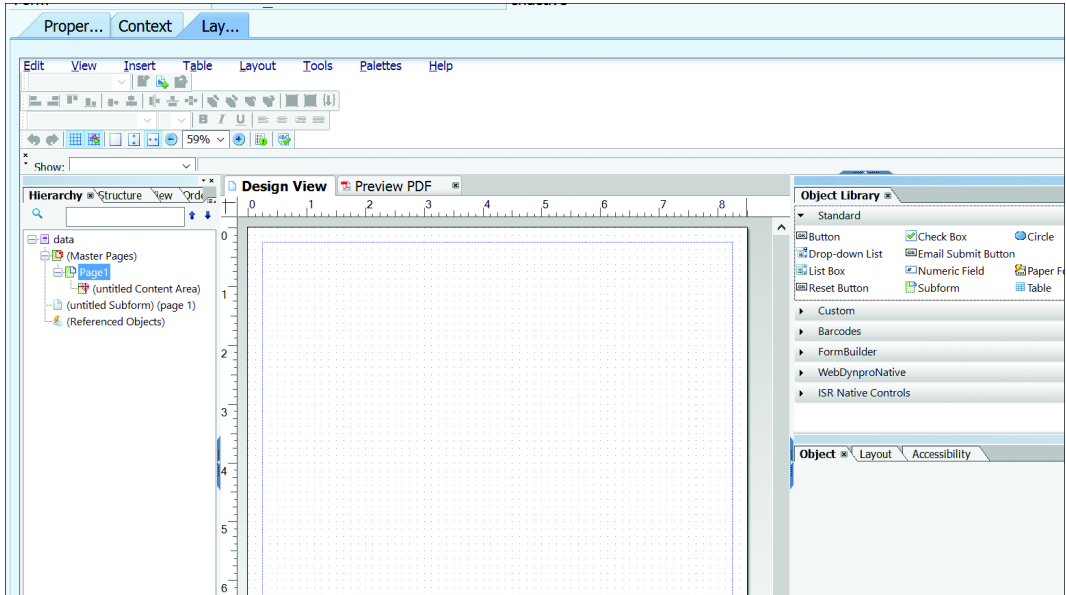


Figure 18.87 Form Layout

The left section contains different tabs. The middle section is the design view, where we add form fields. The PREVIEW PDF tab allows you to quickly preview the form.

The right section contains the library palette and the OBJECT palette. The library palette consists of different form fields organized into multiple groups. Below the library palette, we maintain the properties for form fields in the OBJECT palette. We add the form fields from the library palette to the design view. To add a field, you can simply drag the field from the group and drop it in the design view. Once the field is added, we bind it to the context from the OBJECT palette.

Let's create an image field first. Proceed with the following steps:

1. To add the image field, drag the image field from the STANDARD GROUP under the OBJECT LIBRARY and drop it in the design view, as shown in [Figure 18.88](#).
2. Click on the image field in the design view to set the object properties. In the OBJECT palette, as shown in [Figure 18.89](#), you can set a caption for the image and provide the image URL and sizing.

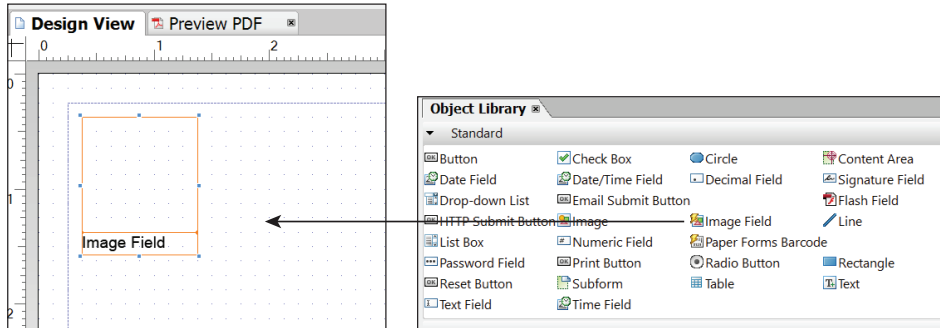


Figure 18.88 Adding Image Field

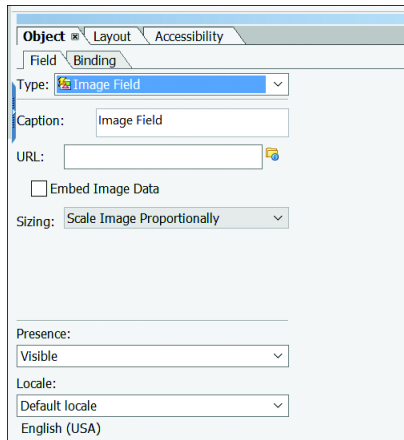


Figure 18.89 Object Palette

3. If you want to bind the image field with the graphic node from the context, you can do so in the BINDING tab of the OBJECT palette. Binding to the graphic node of the context allows you to dynamically pass the image URL. Click the drop-down icon next to the DATA BINDING field and select the GRAPHIC node, as shown in [Figure 18.90](#). Once bound, the field will display the node name preceded by a dollar sign (\$) symbol.

You can also click within the image field in design view to upload the image from your local PC.

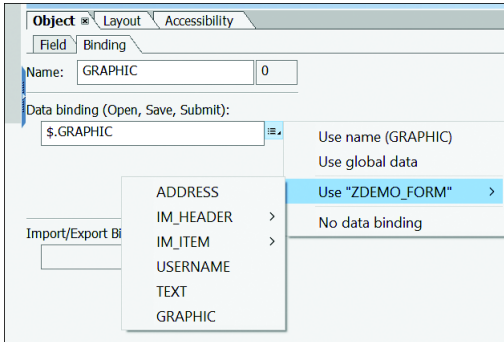


Figure 18.90 Binding Field to Context

- An easier way to add the image field to the design view is to simply drag the GRAPHIC node from the DATA VIEW to the DESIGN VIEW area, as shown in [Figure 18.91](#). This will automatically bind the field.

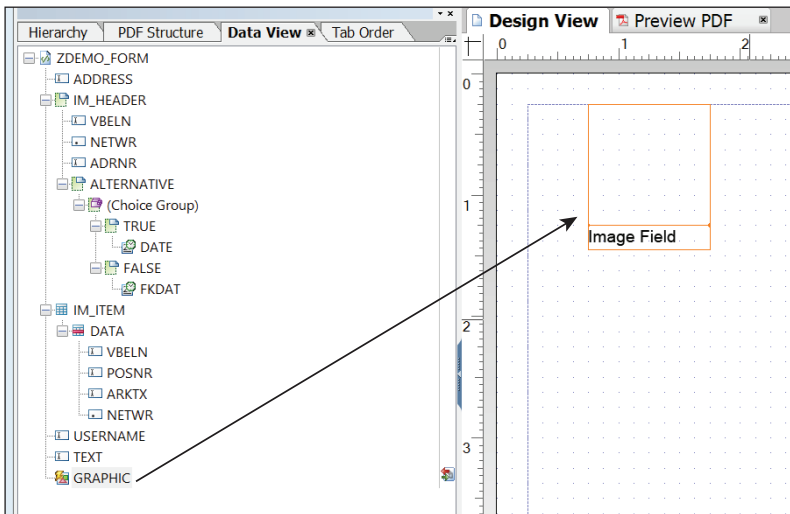


Figure 18.91 Adding Field from Data View

- Similarly, you can add other fields to the design view by selecting the respective object from the OBJECT LIBRARY or by dragging the field from the data view. When adding from the OBJECT LIBRARY, select the relevant field type based on the data type of the field to which it is bound. For example, to show NET VALUE on the form, you select DECIMAL FIELD from the OBJECT LIBRARY and bind it to

NETWR field from the context. If you use the wrong field, the system will automatically prompt you to convert the field when binding.

- When adding fields of a structure or table, you can directly drag the structure from the data view instead of manually adding individual fields. You can also insert the fields using the INSERT menu, as shown in [Figure 18.92](#).

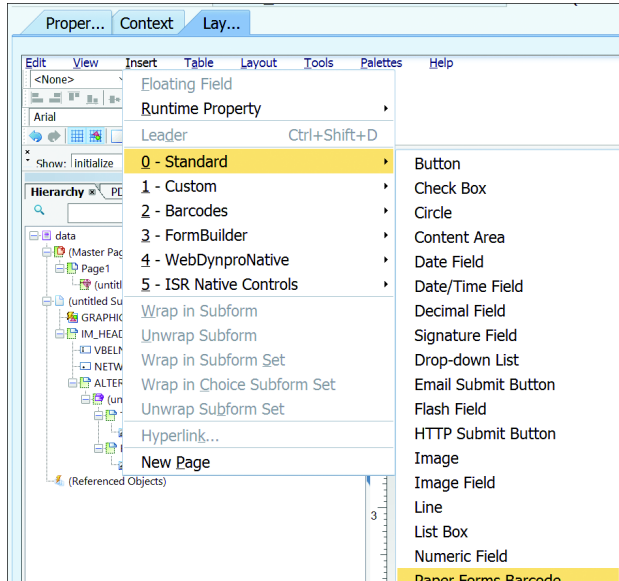


Figure 18.92 Inserting Fields from Menu

Once all the fields are added to the form, activate it. To pass data to the form, we call it from the driver program. Calling the form is similar to the process for a smart form.

18.3.3 Driver Program

When the form is activated, a function module is generated. We call this generated function module in the driver program to pass the data to the form. Similar to smart forms, when the form is transported, the corresponding function module is generated in the target system. This function module may have a different name in the target system than in the current system. Therefore, we use the `FP_FUNCTION_MODULE_NAME` function module to identify the corresponding function module name for the form.

The code for the driver program is similar to the that for a smart form, as shown in [Listing 18.5](#). However, in the `print_form` subroutine of [Listing 18.5](#), we replace the code as shown in [Listing 18.6](#).

```
*&-----*
*&      Form  PRINT_FORM
*&-----*
FORM print_form.
CALL FUNCTION 'FP_FUNCTION_MODULE_NAME'
  EXPORTING
    i_name          = c_formname "ZDEMO_FORM
  IMPORTING
    E_FUNCNAME      = gv_formname.

DATA gs_outputpar TYPE SFOUTPUTPARAMS.
gs_outputpar-reqnew = abap_true.

CALL FUNCTION 'FP_JOB_OPEN'
  CHANGING
    ie_outputparams = gs_outputpar
  EXCEPTIONS
    cancel          = 1
    usage_error    = 2
    system_error   = 3
    internal_error = 4
    OTHERS         = 5.

CALL FUNCTION gv_formname
  EXPORTING
    zbill_header = st_header
    im_item       = it_vbrp
  EXCEPTIONS
    USAGE_ERROR    = 1
    SYSTEM_ERROR   = 2
    INTERNAL_ERROR = 3
    OTHERS         = 4
  .
IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.

CALL FUNCTION 'FP_JOB_CLOSE'.
ENDFORM.
```

Listing 18.6 Calling Form from Driver Program

In [Listing 18.6](#), we're calling the `FP_FUNCTION_MODULE_NAME` function module to get the name of the generated function module for the form. Before this gener-

ated function module is called, we call the `FP_JOB_OPEN` function module to open the job and generate a new spool. Once the job is opened, we call the generated function module to which the data from the driver program is transported.

At the end, we call the `FP_JOB_CLOSE` function module to close the job. Executing the driver program should print the form. The driver program and the form can be assigned to the output type using Transaction NACE, as previously discussed.

18.3.4 Downloading the Form as a PDF

Instead of printing the form, if you want to download the form as a PDF or send the form output back to the application program for further processing (e.g., to send as an email attachment), you can do so by importing the form data. The code in [Listing 18.7](#) shows how to import the form data.

```
CALL FUNCTION 'FP_FUNCTION_MODULE_NAME'
  EXPORTING
    i_name      = 'ZDEMO_FORM'
  IMPORTING
    e_funcname = gv_fmname.

DATA : gs_outputpar TYPE sfpoutputparams,
      gs_formoutput TYPE fpformoutput.

gs_outputpar-nodialog = abap_true.
gs_outputpar-getpdf   = abap_true.

CALL FUNCTION 'FP_JOB_OPEN'
  CHANGING
    ie_outputparams = gs_outputpar
  EXCEPTIONS
    cancel           = 1
    usage_error     = 2
    system_error    = 3
    internal_error  = 4
    OTHERS          = 5.

DATA gs_docparams TYPE sfpdocparams.
gs_docparams-fillable = abap_true "Makes the PDF interactive
CALL FUNCTION gv_fmname
  EXPORTING
    /1bcdwb/docparams = gs_docparams
    zbill_header      = st_header
    im_item            = it_vbrp
  IMPORTING
    /1bcdwb/formoutput = gs_formoutput "Returns PDF!
```

```

EXCEPTIONS
  usage_error      = 1
  system_error    = 2
  internal_error   = 3
  OTHERS          = 4.
IF sy-subrc <> 0.
* Implement suitable error handling here
ENDIF.

CALL FUNCTION 'FP_JOB_CLOSE'.

```

Listing 18.7 Importing Form Data

In [Listing 18.7](#), we're setting the `nodialog` and `getpdf` flags for the output parameters of the `FP_JOB_OPEN` function module. The generated function module exports the raw data, which we're importing into the `gs_formoutput` structure of the program. The `gs_formoutput` structure contains a component called `PDF` that will contain the PDF output. We can set the `fillable` flag in the `gs_docparams` structure to make the PDF interactive. The `FP_UTILITIES` include program provides various routines that can be used to download the PDF.

18.4 Summary

In this chapter, we explored all the different ways to develop forms. We have seen the old technologies and how SAP has improved form development by investing in new technologies. Improvements in technologies over the years have made form development both easy and intuitive.

We started our discussion with SAPscripts and you learned how form design is separated from application logic. We also discussed how to develop a driver program to get the required data and use Form Builder to design the page layout to display the data.

We discussed smart forms, which make form development quite easy compared to SAPscripts. We also discussed various nodes that can be used in smart forms for different purposes.

Finally, we looked at how SAP has leveraged its experience in developing smart form technology to integrate Adobe forms. Adobe forms take form development to the next level. We only scratched the surface here; all the details of Adobe forms are beyond the scope of this chapter.

In the next chapter, we'll discuss interface development. In today's connected world, interfaces play a vital role for any system. SAP has implemented many technologies and many open standards to help SAP systems interface with external systems.

This chapter focuses on interface development. Each section is devoted to a different type of interface and explains both the concepts behind the interface and walks through step-by-step examples of working with the interface.

19 Interfaces

In today's connected world, interfaces play a vital role in exchanging information between different systems, both in heterogeneous landscapes within an organization and in the outside world. Interfaces are used for wide variety of reasons, such as transferring data from a legacy system to an SAP system, providing an application programming interface (API) to perform actions in the SAP system, and more.

Interfaces allow external developers to access and process information in an SAP system without having to understand the underlying business processes or technical implementation. For example, a travel agency that wants to design a booking engine to make airline reservations can implement its website using any technology and use the interface provided by the airline to request seat availability or to process bookings. The travel agency doesn't need to know the technical aspects of the airline's system, nor do the developers in the travel agency need to learn the programming language used in the airline's system. The only thing they need to know is how to use the API provided by the airline and pass the required details to the API.

Some interfaces may be required for one-time use—such as a one-time data transfer from a legacy system—and others for frequent use—such as posting orders from business partners. The data may need to be transferred in real time or at frequent intervals. To cover various scenarios based on business requirements, SAP provides different technologies and tools for interface development.

At the end of the day, all interfaces achieve the same task of posting external data into the SAP system and retrieving data from the SAP system and making it available for external systems.

In this chapter, we will discuss the different methods and tools that can be used to develop various types of interface programs. We begin in [Section 19.1](#), where we discuss direct and batch input for batch data communication (BDC). In [Section 19.2](#), we look at how Business Application Programming Interfaces (BAPIs) can be used to transfer data into SAP.

From there, in [Section 19.3](#), we look at the various data exchange formats (i.e., EDI/ALE/IDocs), before turning our attention to migrating data from legacy systems using the Legacy System Migration Workbench (LSWM) in [Section 19.4](#). In this section, we'll include step-by-step instructions of the migration process.

In [Section 19.5](#) and [Section 19.6](#), we discuss web services and OData services, respectively. First, we'll explore the framework of a web service and the steps for consuming a web service. Then, we'll look at the process of creating OData services, including data model definition, service maintenance, and service implementation.

[Section 19.7](#) walks through serialization and deserialization via XSL transformations and [Section 19.8](#) offers a brief overview of data representation via XML and JSON. Finally, we end the chapter in [Section 19.9](#), with a discussion on ABAP channels.

19.1 Batch Data Communication

SAP provides different methods to transfer data into an SAP system, such as via BAPIs, batch input, or direct input. BAPIs are standardized programming interfaces that provide external access to SAP data and business processes, which we'll discuss in [Section 19.2](#).

Batch input is a standard technique used to move large volumes of data into an SAP system. The advantage of batch input is that all the transaction-level checks can be performed on the data as if it were entered online, and the data is updated through the transactions.

Direct input is similar to batch input in that all the transaction-level checks can be performed on the data. However, with direct input, the data is updated into the database directly as opposed to via transactions. All these data transfer methods allow you to move data from external systems into your SAP system, either as one-time transfers or as regular periodic transfers.

Almost any data transfer project involves the following steps:

1. Identifying the data that needs to be transferred

The first step is to identify the data that should be transferred into an SAP system—for example, material master data from a legacy system. Once the data is identified, it should be exported from the legacy system into a flat file. The exported data should be cleaned up to remove any inconsistencies or any old data. You can clean up all the data that was temporarily created and is no longer used, such as one-time customers' data or any materials that no longer exist or don't need to be transferred.

2. Identifying the frequency of transfer

The next step is to identify the frequency of data transfer. Based on this, you can identify which data transfer technique is most suitable. For example, if you need to transfer the data only once, a batch input or a direct input method may be more suitable. If the data needs to be transferred frequently and in real time, then using a BAPI might be more suitable.

3. Identifying the conversion logic

External data should be converted to an SAP-specific format. For example, if the material number is stored as a ten-character field in the legacy system, it may need to be converted to an eighteen-character field (SAP format) before transferring the data into the SAP system. This is typically done via a transfer program.

4. Processing the data transfer program

A transfer program uploads the flat file and processes the data to be transferred into the SAP system. Once the data is processed and converted to SAP format, it will be posted in the SAP system.

5. Handling any errors

Any errors that arise during data transfer need to be addressed. For example, if there are errors in the data, the errors should be corrected in the source file and reprocessed.

6. Verifying the transferred data

After successful transfer of the data, it needs to be verified to ensure data consistency. For example, if we transferred material master data from a legacy system to an SAP system, the material data should be verified in the SAP system for consistency and completeness after the successful transfer.

As mentioned, there are two methods we use to transfer mass data: direct input and batch input. The direct input method relies on certain function modules to validate the data, whereas a batch input program relies on the screens of the corresponding transaction to validate the data.

These methods are suitable for transferring mass data for one-time transfer or periodic transfer, but they're not suitable if real-time data transfer is required. We typically use these methods when moving data from a legacy system into an SAP system just before go-live.

In the following subsections, we'll discuss the direct and batch input methods in greater detail.

19.1.1 Direct Input

SAP provides certain direct input programs that can be used to move data from an external system into an SAP system. These direct input programs typically use function modules to validate the data and update it in the database. The function modules subject the data to the same level of validations as a corresponding transaction. For example, when you create a material master record using Transaction MM01, the transaction validates the data you input on the screen (e.g., material group, unit of measure, etc.).

When the material master data is transferred using the corresponding direct input program, the direct input program subjects the data to the same validations that are performed by Transaction MM01.

For example, a material master record transferred using the direct input program RMDATIND will be subjected to same set of validations as a material created using Transaction MM01. Transaction MM01 allows you to create a material manually, whereas the direct input program can be used for bulk transfer.

Table 19.1 lists the direct input programs provided by SAP for different application areas.

Program	Application	Description
RFBIBL00	FI	Used for entering accounting documents and clearing open items
RMDATIND	MM	Transfers material master data

Table 19.1 Direct Input Programs

Program	Application	Description
RVAFSS00	SD	Creates billing documents
RAALTD11	AM	Direct data import for asset accounting
RKEVEXT0	CO-PA	Transfers external data to CO-PA

Table 19.1 Direct Input Programs (Cont.)

A direct input program is faster than a batch input program because it doesn't depend on transaction screens to validate the data. Direct input programs are executed in the background and support error reporting. In case of any errors, the data transfer can be resumed after correcting the errors. A background job can be scheduled to run the programs at regular intervals.

Direct input programs are run in the background and managed using Transaction BMV0 (see [Figure 19.1](#)). Running direct input programs in the background is the preferred method for production runs, but you can also execute these programs in the foreground for testing the data transfers.

The program documentation for each of the direct input programs provides more information on their usage.

Figure 19.1 Direct Input Management

19.1.2 Batch Input

Unlike direct input programs, batch input programs rely on the corresponding transactions to validate the data. Batch input programs allow you to capture the steps involved in processing the data and automate this process.

For example, to create a material, you'd go through the following steps:

1. Go to Transaction MM01, which will take you through a series of screens in which you input certain details on each screen. For this example, input the MATERIAL, INDUSTRY SECTOR, and MATERIAL TYPE on the initial screen of Transaction MM01, as shown in [Figure 19.2](#).

Figure 19.2 Initial Screen of Transaction MM01

2. On the next screen, select the view to input further data, as shown in [Figure 19.3](#).

Figure 19.3 Second Screen of Transaction MM01

3. The system will present the selected view, and you can input further details of the material, such as its description, MATERIAL GROUP, BASE UNIT OF MEASURE, and so on, as shown in [Figure 19.4](#).
4. Once the information is entered, click the SAVE button to create a material.

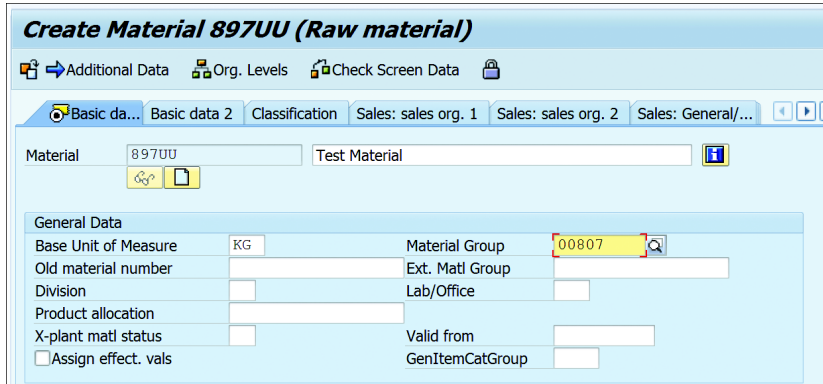


Figure 19.4 Third Screen of Transaction MM01

To create another material, you repeat the same steps with a different set of data. If you want to create only a few materials, it's easy to create materials manually via this transaction. But what if you want to create hundreds or thousands of materials? We can automate this process by creating a batch input program. We'll walk through the steps for doing so in the next subsection.

Creating a Batch Input Program

A batch input program requires you to capture the screen and field information in an internal table with a special structure known as `BDCDATA` (BDC here indicates *batch data communication*).

The `BDCDATA` structure is shown in [Figure 19.5](#).

Component	Typing Method	Component Type	Data Type	Length	Deci...	Short Description
PROGRAM	Types	BDC PROG	CHAR	40		0BDC module pool
DYNPRO	Types	BDC DYNR	NUMC	4		0BDC Screen number
DYNBEGIN	Types	BDC START	CHAR	1		0BDC screen start
FNAM	Types	BDC FNAM 4	CHAR	132		0Field name
EVAL	Types	BDC EVAL	CHAR	132		0BDC field value

Figure 19.5 BDCDATA Structure

The BDCDATA structure consists of five components, as detailed in [Table 19.2](#).

Field Name	Meaning
PROGRAM	Program name of the screen to which the data is transferred
DYNPRO	Screen number of the program (specified in the PROGRAM field above)
DYNBEGIN	Flag to indicate new screen
FNAM	Field name of the screen
FVAL	Value to be transferred to the field

Table 19.2 BDCDATA Structure Components

The basic concept behind developing a batch input program is that you capture the program name and screen field names of the transaction and map the values that should be passed to the screen fields using table BDCDATA. For example, if you look at the initial screen of Transaction MM01 and use the **[F1]** help on the MATERIAL field to look at the technical information, you can find the technical details of the field as shown in [Figure 19.6](#).

Screen Data	
Program Name	SAPLMGMM
Screen Number	0060

GUI Data	
Program Name	SAPLMGMM
Status	MENU

Field Data	
Table Name	RMMG1
Table category	Structure
Field Name	MATNR
Search Help	=
Data Element	MATNR
DE Supplement	1
Parameter ID	MAT

Field Description for Batch Input	
Screen Field	RMMG1-MATNR

Figure 19.6 Technical Information about Screen Field

Notice that the program name is `SAPLMGMM`, the screen number is `0060`, and the screen field name is `RMMG1-MATNR`. Using this information, we can map the BDC-DATA structure, as shown in [Listing 19.1](#).

```
PROGRAM ZDEMO_BATCH_INPUT.
Data: it_bdcdata TYPE STANDARD TABLE OF bdcdata,
      wa_bdcdata TYPE bdcdata.
START-OF-SELECTION.
perform bdc_dynpro      USING 'SAPLMGMM' '0060'.
perform bdc_field      USING 'RMMG1-MATNR' '567ASHG'.
PERFORM bdc_field      USING 'RMMG1-MBRSH' 'P'.
PERFORM bdc_field      USING 'RMMG1-MTART' 'ROH'.
PERFORM bdc_dynpro     USING 'SAPLMGMM' '0070'.
.....
FORM bdc_dynpro USING program dynpro.
  CLEAR wa_bdcdata.
  wa_bdcdata-program = program.
  wa_bdcdata-dynpro = dynpro.
  wa_bdcdata-dynbegin = 'X'.
  APPEND wa_bdcdata TO it_bdcdata.
ENDFORM.
FORM bdc_field USING fnam fval.
  CLEAR wa_bdcdata.
  wa_bdcdata-fnam = fnam.
  wa_bdcdata-fval = fval.
  APPEND wa_bdcdata TO it_bdcdata.
ENDFORM.
```

Listing 19.1 Filling BDCDATA Structure

In [Listing 19.1](#), we defined an internal table `it_bdcdata` with the line type as BDC-DATA. We defined two subroutines, `BDC_DYNPRO` and `BDC_FIELD`. In the `BDC_DYNPRO` subroutine, we're filling the program name and screen number. In the `BDC_FIELD` subroutine, we're filling the field name and field value. The `PERFORM` statements for both these subroutines are passing two actual parameters.

For example, the `PERFORM BDC_DYNPRO` statement is passing the program name and screen number as literals, and the `PERFORM BDC_FIELD` statement is passing the field name and field value. In our example, we're passing the literals for the field value, but the literal will be replaced with the field value uploaded from the sequential file. We're flagging `dynbegin` as `X` in the `BDC_DYNPRO` subroutine, because this subroutine is called only when new screen information needs to be passed.

Once the internal table is filled with all the required screen and field information, we can pass the data by using either the `CALL TRANSACTION` method or the `SESSION` method. We'll look at these methods in the next two subsections.

CALL TRANSACTION

In the `CALL TRANSACTION` method, `BDCDATA` is passed to the transaction using the `CALL TRANSACTION USING` statement. The internal table that contains `BDCDATA` is passed with the statement. The transaction can be executed in foreground mode, background mode, or foreground on error mode.

When a transaction is executed in *foreground mode*, the transaction runs in the foreground, displaying all the screens. Running a transaction in foreground mode is useful when doing a test run, because it helps verify that the fields are mapped correctly with the right data.

When a transaction is executed in *background mode*, the transaction completely runs in the background and no screens are displayed. Any messages raised by the transaction during the execution will be transferred to the batch input program at the end.

Running a transaction in background mode is the preferred method for production runs. If *foreground on error* mode is used, the transaction runs in the background unless any error messages are raised by the transaction. If an error message is raised, the transaction screen that has caused the error is displayed to the user.

Data can be updated synchronously or asynchronously. With synchronous updates, the system waits for the previous transaction to commit, while with asynchronous updates, the system will start processing the next transaction without waiting for the previous transaction to commit.

For example, if you're updating a sales order header and item data, synchronous updates can ensure that the sales order item data is updated only after the header data is committed to the database. If you use asynchronous updates, the sales order item data may commit to the database without waiting for the header data to commit first. For this reason, an asynchronous update is faster than a synchronous update.

The `CALL TRANSACTION` method doesn't support an automated error log or resuming on error. Error handling should be performed in the program. You can call one transaction at a time, and the data is processed synchronously.

SESSION

In the `SESSION` method, we generate a session with all the data. The session is then processed separately. The session can be processed manually or at regular intervals by scheduling the program `RSBDCSUB` as a background job. This method automatically creates an error log in case of any errors.

The error transactions can be processed, after correcting the errors, by resuming the session. The `BDC` program opens the session (using function module `BDC_OPEN_GROUP`), writes the data to the session (using function module `BDC_INSERT`), and closes the session (using function module `BDC_CLOSE_GROUP`). The previous session should be closed before opening a new session.

The `SESSION` method can be used to transfer data to multiple transactions and perform a synchronous database update. For example, you can use the `SESSION` method to post the header and item data of a document and have the system update the item data after updating the header data.

Transactions should always be updated in background mode; foreground mode should be used only for testing the screen sequence.

Both the `CALL TRANSACTION` method and the `SESSION` method require you to pass the screen and field information of the transaction using table `BDCDATA`. However, capturing all the screen and field information manually is a tedious job. To make this easy and generate a batch input program, SAP provides a tool called Transaction Recorder (Transaction `SHDB`).

In the next subsection, we'll discuss the steps for generating a batch input program with Transaction `SHDB`.

Creating a Batch Input Program Using the Transaction Recorder

Using Transaction Recorder, we can simulate the sequence of steps performed by the transaction to capture all the required information for table `BDCDATA`. Transaction Recorder also supports generating the batch input program using this information.

The following steps walk through generating a batch input program via Transaction Recorder:

1. On the initial screen of Transaction SHDB (TRANSACTION RECORDER: RECORDING OVERVIEW), as shown in [Figure 19.7](#), click the NEW RECORDING button.

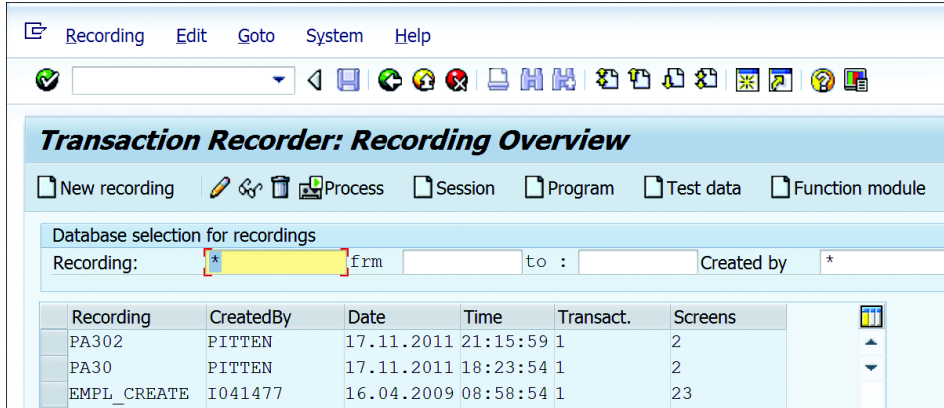


Figure 19.7 Transaction Recorder Initial Screen

2. In the CREATE RECORDING dialog box, as shown in [Figure 19.8](#), provide a name for the recording and enter the transaction code for which the recording should be performed. You can also set the UPDATE MODE to update the data. There are three update modes available, as listed in [Table 19.3](#).

Update Mode	Meaning
LOCAL	The update task is carried out by the dialog work process that is processing the transaction.
ASYNCHRONOUS	The update is performed by the update work process asynchronously.
SYNCHRONOUS	The update is performed by the update work process synchronously.

Table 19.3 Update Mode Options

Also provide the following settings in the CREATE RECORDING box ([Figure 19.8](#)):

- ▶ Select the DEFAULT SIZE checkbox to avoid screen resolution issues and use a default size for recording.
- ▶ The CONT. AFTER COMMIT checkbox sets a flag to continue processing when a commit work is performed while using the CALL TRANSACTION method.

- ▶ The NOT A BATCH INPUT SESSION checkbox sets the flag to use the CALL TRANS-ACTION method.
- ▶ Click the START RECORDING button to run the transaction and capture the steps.

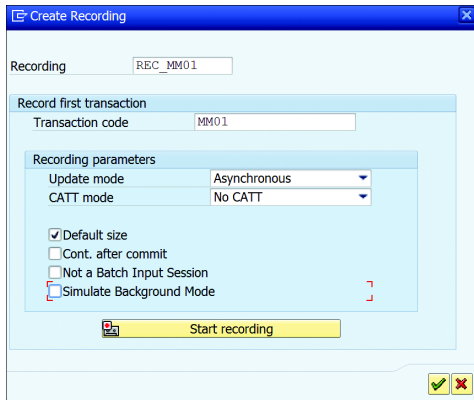


Figure 19.8 Create Recording Screen

3. While recording the transaction, try to avoid any errors, because the error steps will also be recorded. Once the recording is complete, you can see all the screen-related information being captured, as shown in [Figure 19.9](#).

Transaction Recorder: Change Recording REC_MM01

Line	Program	Screen	Start ID	Field name	Field value
1			T	MM01	
2	SAPLMGMM	0060	X		
3				BDC_CURSOR	RMMG1-MATNR
4				BDC_OKCODE	=AUSW
5				RMMG1-MATNR	888997uu
6				RMMG1-MBRSH	P
7				RMMG1-MTART	ROH
8	SAPLMGMM	0070	X		
9				BDC_CURSOR	MSICHTAUSW-DYTXT(01)
10				BDC_OKCODE	=ENTR
11				MSICHTAUSW-KZSEL(01)	X
12	SAPLMGMM	4004	X		
13				BDC_OKCODE	=BU
14				BDC_SUBSCR	SAPLMGMM 2004TABFRA1
15				BDC_SUBSCR	SAPLMGD1 1002SUB1
16				MART-MARTX	Test material
17				BDC_SUBSCR	SAPLMGD1 2001SUB2
18				BDC_CURSOR	MARA-MATKL
19				MARA-MEINS	KG
20				MARA-MATKL	00807
21				BDC_SUBSCR	SAPLMGD1 2561SUB3
22				BDC_SUBSCR	SAPLMGD1 2007SUB4
23				BDC_SUBSCR	SAPLMGD1 2005SUB5
24				BDC_SUBSCR	SAPLMGD1 2011SUB6
25				BDC_SUBSCR	SAPLMGD1 2033SUB7
26				BDC_SUBSCR	SAPLMGD1 0001SUB8
27				BDC_SUBSCR	SAPLMGD1 0001SUB9
28				BDC_SUBSCR	SAPLMGD1 0001SUB10

Figure 19.9 Screen Recorder Data

4. Save the recording by clicking the SAVE button in the toolbar, and click the BACK button to go back to the initial screen of the transaction recorded.
5. On the initial screen of Transaction Recorder, select the recording and click the PROGRAM button, as shown in [Figure 19.10](#).

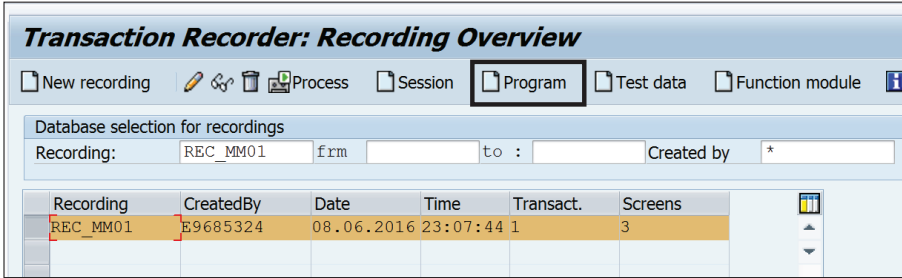


Figure 19.10 Create Program from Recording

6. In the GENERATE PROGRAM window, provide the PROGRAM NAME and select the options shown in [Figure 19.11](#).

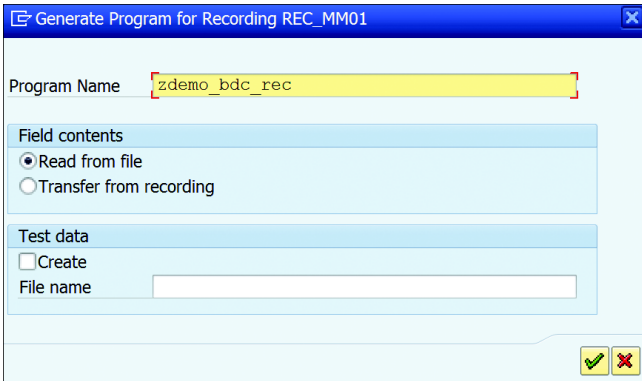


Figure 19.11 Generate Program

The READ FROM FILE option generates additional code to allow the program to upload the data from a sequential file. The sequential file should have the fields in the same order as the structure of the internal table generated in the program (see [Figure 19.12](#)). The TEST DATA section allows you to generate the test data file in the application server directory, which the generated program can read.


```

*** Generated data section with specific formatting - DO NOT CHANGE ***
data: begin of record,
* data element: MATNR
  MATNR_001(018),
* data element: MBRSH
  MBRSH_002(001),
* data element: MTART
  MTART_003(004),
* data element: XFELD
  KZSEL_01_004(001),
* data element: MAKTX
  MAKTX_005(040),
* data element: MEINS
  MEINS_006(003),
* data element: MATKL
  MATKL_007(009),
  end of record.
*** End generated data section ***

```

Figure 19.12 Structure for Sequential File

The selection screen of the generated program, as shown in [Figure 19.13](#), provides you the option to process the program using either the `SESSION` method or the `CALL TRANSACTION` method.

The selection screen is divided into two main sections: 'Generate session' (selected) and 'Call transaction'.

Generate session (selected):

- Session name:
- User:
- Keep session:
- Lock date:
- Nodata indicator:
- Short log:
- DATASET:

Call transaction:

- Processing Mode:
- Update Mode:
- Error sessn:
- User:
- Keep session:
- Lock date:

Figure 19.13 Selection Screen of Generated Program

When processing via the `CALL TRANSACTION` method, the transaction is called immediately and the records are processed immediately. If the `SESSION` method is used, the program generates a session that can be processed later. The records will be updated when the session is processed. You can automate processing the session by scheduling a batch job to run program `RSBDCSUB`.

To process a session manually, proceed with the following steps:

1. Go to Transaction SM35.
2. Select the session from the list and click the PROCESS button (see [Figure 19.14](#)) to process the session. If any errors occur during the processing, you can check the error log to identify the cause of the error.
3. After correcting the error, the session can be processed again to post only the records that returned an error. This ensures data consistency and avoids the same records being updated twice.
4. On the selection screen of the generated program, as shown in [Figure 19.13](#), the KEEP SESSION checkbox ensures the session isn't deleted after successful processing, assuming no errors occur. If the KEEP SESSION checkbox isn't selected, the session will be deleted if all transactions are processed successfully.

Session name	Sta...	C.	Date	Time	C...	Lock Date	A..	Trans.			
SESSION		NI_04.0	01:26:19	ZNI.			NI_5	0	0	15	
SESSION		NI_04.0	01:22:38	ZNI.			NI_5	0	0	15	

Figure 19.14 Processing Session

5. You can also generate a function module from the recording via Transaction Recorder. The function module will act as an interface to transfer the data into the SAP system. Each field of the recording will be added as a parameter to the function module.
6. When processing the batch input session, the system uses the authorizations of the user currently processing the session. If the batch input session is executed using a background job, then you can assign the username of a user whose authorizations should be used to process the session.

19.2 Business Application Programming Interface

BAPIs are standard SAP interfaces that play an important role in the technical integration and exchange of data between different systems, including communication between SAP systems and between SAP and non-SAP systems.

BAPIs are integrated at the business level, which allows them to encapsulate underlying technical details and provide greater stability to integrate various systems.

BAPIs are typically used in the following scenarios:

- ▶ Connecting to third-party systems and software
- ▶ Connecting to legacy systems
- ▶ Implementing distributed environments in which different business components are implemented across various systems
- ▶ Enabling communications between PC programs, such as desktop applications developed using Visual Basic and so on, and SAP systems
- ▶ Connecting SAP systems to the Internet

In this section, we'll look at the different object types and components that can be accessed through BAPIs, the steps and tools used for developing BAPIs, the use of standardized BAPIs for implementing standard functionality, and finally the steps for implementing a BAPI.

19.2.1 Business Object Types and Business Components

BAPIs allow access to the business object types in an SAP system, and business object types allow you to break the SAP system into individual components that are easy to manage and that reduce complexity.

To understand the role of BAPIs, we need to understand what constitutes a business component and a business object type:

▶ **Business component**

A *business component* consists of business objects and provides autonomous functions. For example, the Human Resources business component consists of a business object type employee.

► **Business object type**

A *business object type* represents one business entity. It includes the data and functions of this entity.

► **BAPIs**

BAPIs are added as an API method to the business object type. They allow object-oriented access to the SAP system through the methods of the business object types. BAPIs and business object types together define the interface at the business level. For example, a purchase order business object type can have different BAPIs added as API methods to create a purchase order, edit a purchase order, or display a purchase order.

Business object types allow you to structure data and processing according to business criteria. Because BAPIs are defined as API methods of business object types, many external applications can gain object-oriented access to the SAP system.

19.2.2 BAPI Development via BAPI Explorer

To help you find existing BAPIs in the SAP system and develop new BAPIs, SAP provides a tool called *BAPI Explorer*.

BAPI Explorer (Transaction BAPI) integrates all the tools required to develop and maintain BAPIs. Using BAPI Explorer, you can search for available BAPIs in the system either by application area or by alphabetical order, using the HIERARCHICAL tab or ALPHABETICAL tab, respectively, as shown on the left-hand side of the screen in [Figure 19.15](#).

On the right-hand side of the screen (see [Figure 19.15](#)), BAPI Explorer contains different tabs that provide various information about the BAPI in the following areas:

► **DETAILS**

Provides the business object details, such as the object name, object type, package, component, release version, and so on, as shown in [Figure 19.15](#).

► **DOCUMENTATION**

Displays the BAPI documentation, which helps explain the purpose and usage of the BAPI. Each BAPI should be well-documented during implementation to help external developers understand the BAPI's usage.

- ▶ **TOOLS**
Contains tools to create a web service from a BAPI and change the definition of the business object type.
- ▶ **PROJECT**
Allows you to create a new project and carry out all the activities to implement a BAPI.

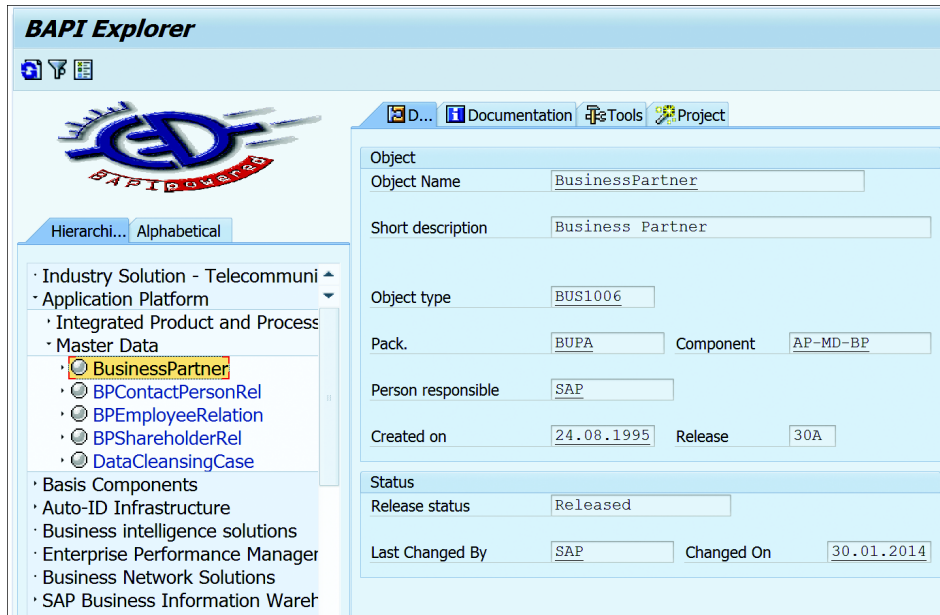


Figure 19.15 BAPI Explorer

The external application developer can call BAPIs as methods of the business object types without needing to understand underlying technical complexities of the business process. Technically, BAPIs are implemented as *remote-enabled function calls* (RFC). RFC is SAP's own standard protocol to allow functions in remote systems to be called.

A BAPI development can be divided into four phases:

1. Analysis

In the analysis phase, you determine which business process should be implemented using the BAPI. In this phase, you identify the business scenarios and

determine which steps of the business process should be exposed to external components. You also identify the business object types and their BAPIs. For example, you may want to have a `CreateForm` BAPI to create an order and another BAPI to edit the order.

2. Design

In the design phase, you define the interface for each BAPI involved in the scenario. In this phase, you only design the structure conceptually. The idea is to design the interfaces of the BAPI intuitively so that they're easy to use.

3. Implementation

In the implementation phase, you define the structures, data elements, and domains on which the parameter interface of the BAPI is based. Once the parameter interface is defined, you write the program logic to perform the required task within a function module. After the function module is developed, the function module is added as an API method to the business object using the *Business Object Repository* (BOR). Adding the function module as an API method allows for object-oriented access to the business object.

4. Testing

Once the BAPI is implemented, you test the BAPI in the testing phase. All the required documentation for the BAPI is maintained during this phase. After successful testing, the BAPI is released.

BAPIs are designed to be called from external applications. The external developer calling the BAPI shouldn't be expected to have any knowledge of SAP. The application knowledge and the BAPI documentation should suffice for external developers to work with the BAPI.

For this reason, the parameter interface and the field names should use generally understandable names. For example, a material field should be named `MATERIAL` instead of `MATNR`, and a language field could be named `LANGU` instead of `SPRAS`.

Before developing a BAPI, you also need to identify if the BAPI is added as a class method or an instance method to the business object type:

► Class method

A *class method* doesn't refer to any specific instance but rather is applicable to all instances—for example, getting details about all the flights or orders for a given search criterion.

► **Instance method**

An *instance method* refers to a specific instance of a business object type—for example, getting details about a particular flight or an order.

Another important decision to make when implementing a BAPI is to check if the BAPI can be implemented as a standardized BAPI. Some BAPIs provide basic functions that can be used for most business object types. These BAPIs that provide similar functions are called *standardized BAPIs* and must be implemented similarly across business object types. For example, the `GetList()` BAPI is used to select multiple instances of objects in a business object type, such as materials, employees, and so on. Similarly, a `Create()` or `CreateFormData()` BAPI is used to create an instance of the business object type, such as creating an employee record.

If the BAPI you're planning to develop can be implemented as a standardized BAPI (i.e., it performs standardized functions), then you should follow the naming conventions defined for standardized BAPIs. This helps avoid confusion when accessing BAPIs across multiple business object types. BAPIs that do not provide standard functions can be developed as individual BAPIs with company-specific naming conventions.

Various BAPIs can have the same parameters because they contain the same or the equivalent data in all BAPIs. Such parameters are known as *standardized parameters* and should be implemented the same way in all BAPIs. For example, a standard `BAPIADDR2` reference structure is provided for addresses of individuals. All BAPIs containing addresses of individuals must copy this structure to use in the BAPI.

19.2.3 Standardized BAPIs

If the BAPI provides standard functions, it's recommended to implement a standardized BAPI instead of an individual BAPI.

Some of the standardized BAPIs provided are as follows:

► **GetList()**

The `GetList()` BAPI provides a list of instances for a business object type. This BAPI is implemented as a class method and returns the list of objects within the business object type—for example, a list of employees or materials.

- ▶ **GetDetail()**
The `GetDetail()` BAPI returns the details of an instance of the business object type—for example, details about a specific employee or specific material. This BAPI is added as an instance method.
- ▶ **Create or CreateFormData**
This BAPI creates an instance of the business object type—for example, creating an employee or creating a material.
- ▶ **Change()**
This BAPI changes the existing instance of a business object type—for example, changing the details about an employee maintained in the business object type.
- ▶ **Delete()**
This BAPI deletes an instance of the business object type from the database. Normally, we set a delete flag for the record instead of actually deleting the record. The `Delete()` BAPI is an instance method.
- ▶ **Undelete()**
This BAPI removes the delete flag for the instance of the business object type. The `Undelete()` BAPI is an instance method.
- ▶ **Cancel()**
This BAPI cancels the instance of a business object type. Unlike `Delete()`, the instance is not deleted but is cancelled by creating a new instance—for example, cancelling an invoice receipt.
- ▶ **Add<subobject>()**
This BAPI adds a subobject to the business object instance. For example, to add a subobject `item` to the business object type `purchase order`, you create the `AddItem()` BAPI.
- ▶ **Remove<subobject>()**
This BAPI removes a subobject from an object—for example, `RemoveItem()`.

19.2.4 Standardized Parameters

BAPIs that contain the same or equivalent data should be implemented with same parameters. The standardized parameters are as follows:

- ▶ **Address parameters**
Specific standard SAP structures are defined for address parameters for BAPIs.

You should copy these structures to use in your BAPIs. [Table 19.4](#) lists the standard address reference structures that should be used in BAPIs.

Structure	Use
BAPIADDR1	Addresses of companies and organizations
BAPIADDR2	Addresses of individuals
BAPIADDR3	Addresses of contact persons within an organization
BAPIADDR1X	Change structure for addresses of companies and organizations
BAPIADDR2X	Change structure for addresses of individuals
BAPIADDR3X	Change structure for addresses of contact persons within an organization

Table 19.4 Address Reference Structures

► **Change parameters**

In BAPIs that cause database changes (e.g., `Change()` or `CreateFormData()`), the parameters that contain modified values must be differentiated from parameters that contain original values. This distinction can be made by using the standardized parameters. For example, in [Table 19.4](#), the parameter that contains addresses for individuals should use the reference structure `BAPIADDR2` if the parameter contains the original value and the reference structure `BAPIADDR2X` if it contains the modified value.

► **Extension parameters**

To provide a mechanism for customers to enhance a BAPI without modification, you should include the `ExtensionIn` (for data import) and `ExtensionOut` (for data export) parameters. These extension parameters serve as containers to pass customer enhancements to the BAPI and must always be based on the `BAPIPAREX` structure.

► **Return parameters**

Each BAPI must have an export parameter to return messages to the calling application. This return parameter should be named `Return` and implemented in a standardized way to provide a consistent error handling process.

► **Test run parameters**

BAPIs that cause database changes (`Change()` or `CreateFormData()`) should use the `TestRun` parameter to simulate the creation of object instance without actually updating the data.

19.2.5 Implementing BAPIs

BAPIs are usually implemented as remote-enabled function modules. Once the function module is implemented, the BAPI is added as an API method to the business object type in the BOR. The BOR (Transaction SWO1) is used to describe and create business object types and their BAPIs. This allows for object-oriented access to the SAP system from external applications. [Figure 19.16](#) shows an object in the BOR containing attributes, methods, and events.

Object type	SBOOKAUX	Flight booking
Attributes		
FlightBookingAux.ObjectType		Object type
FlightBookingAux.Flight		Flight
FlightBookingAux.Customer		Customer
FlightBookingAux.TravelAgency		Travel agency of booking
FlightBookingAux.ConnectionID		Flight connection code
FlightBookingAux.FlightDate		Departure date
FlightBookingAux.CustomerNumber		Customer number
FlightBookingAux.FlightClass		Flight Class
FlightBookingAux.PostingDate		Posting Date
FlightBookingAux.CounterNumber		Number of sales office
FlightBookingAux.AgencyNumber		Travel Agency Number
FlightBookingAux.CancelledFlag		Cancellation indicator
Methods		
FlightBookingAux.Find		Find object
FlightBookingAux.CreateFromData	✓	Create flight booking
FlightBookingAux.Confirm	✓	Confirm flight booking
FlightBookingAux.Cancel	✓	Cancel flight booking
FlightBookingAux.GetList	✓	Determine list of flight bookings
FlightBookingAux.CreateAndRespond		Create Flight Booking and Send Confirmation
FlightBookingAux.SendResponse		Send Confirmation of Flight Booking (Outbound)
FlightBookingAux.Create		Create Object
FlightBookingAux.Edit		Change Object
FlightBookingAux.ExistenceCheck		Check Existence of Object
FlightBookingAux.Display		Display Object
FlightBookingAux.DARGEVerify		Verify for Archiving/Deletion
FlightBookingAux.BookFlightFromQuot		Book Flight
FlightBookingAux.CheckFlightAvailable		Flights Available?
FlightBookingAux.GetFirstQuotation		Choose First Offer
Events		
FlightBookingAux.Created		Booking created
FlightBookingAux.Cancelled		Booking has been cancelled (not deleted)

Figure 19.16 Object in BOR

We use a few different tools when implementing BAPIs:

► Function Builder

The BAPI is implemented as an RFC-enabled function module in Function

Builder. The BAPI interface and the business logic is maintained within the function module.

▶ **ABAP Data Dictionary**

The BAPI interface should consist of parameters that refer to ABAP Data Dictionary objects. We use ABAP Data Dictionary to define the required structures, data elements, and so on for defining the parameter interface of the BAPI.

▶ **Business Object Repository**

The BOR (Transaction SWO1) is used to define the business object type. Once the business object type is defined, the BAPI is added as an API method to the business object type. One business object type contains multiple BAPIs (methods).

BAPI development is similar to developing a function module in Function Builder. However, when developing a BAPI, certain conventions and rules should be followed, as we'll discuss next.

Conventions

When implementing BAPIs, certain conventions apply. These conventions should always be adhered to for both BAPI methods and BAPI parameters.

The following conventions should be followed for *methods*:

- ▶ If the BAPI that is being implemented is a standardized BAPI, then you should use generic names for the method—for example, `GetList`, `GetDetail`, and so on.
- ▶ You should always use English names for the methods, and the method name can be a maximum of thirty characters in length.
- ▶ You shouldn't use underscores (`_`) in method names. You should use camel case for BAPI names; that is, the first letter of each word should be in uppercase, as in `GetList`.
- ▶ Each BAPI should have a return parameter (export parameter). The return parameter should be either an export structure or a table of type `BAPIRET2`.
- ▶ To support enhancement of BAPIs, the BAPI interface should include `ExtensionIn` and `ExtensionOut` structures. These extension structures allow you to pass additional data to the BAPI.

For example, if you performed a screen enhancement for Transaction ME21N to capture additional data while creating a purchase order, then the corresponding `Create()` BAPI for a business object type purchase order can be enhanced to process this additional data using the extension structures. The extension structures are based on the type `BAPIPAREX`.

The following conventions should be followed for *parameters*:

- ▶ Wherever possible, optional parameters should be used to avoid incompatible changes in the future and provide stability to the interface.
- ▶ Related parameters should be grouped into a structure instead of using too many scalar parameters.
- ▶ The parameter names should be as meaningful as possible. Avoid using technical names, such as *table names*, or abbreviations, such as *flag*.
- ▶ The parameter names should be in English and a maximum of thirty characters.
- ▶ For ISO-relevant fields such as country, language, unit of measure, and so on, additional fields for ISO codes should be provided. For example, if you have a language field `LANGU`, then an additional field for the ISO code should be provided as `LANGU_ISO`.
- ▶ A quantity field must accompany a unit of measure field. Similarly, a currency field must accompany a currency identifier.

Defining the Data Structure in the ABAP Dictionary

Once the parameter interface is identified, the required data structures should be defined in ABAP Data Dictionary. This includes defining the required structure, data elements, domains, and so on. We discussed defining these data structures in [Chapter 10](#).

When defining the data structures in ABAP Data Dictionary for the BAPI interface, certain conventions should be followed:

- ▶ The name of the structure should start with `(namespace)BAPI_`. For example, `ZBAPI_`. If the parameter of a BAPI is a structure, it can be referred directly to this structure in ABAP Data Dictionary. However, if the parameter is an individual field, then it should be referred to the component of the structure.
- ▶ You should create separate data structures for BAPIs and not use the existing structures used for the applications, because when the BAPI is released, the

underlying structures will be frozen and certain restrictions will apply when you want to change these structures later.

- ▶ You should keep the structure name as meaningful as possible.
- ▶ The data structure should not have any `APPEND` or `INCLUDE` structures, because such structures can cause incompatibilities if the BAPI data structure changes.
- ▶ Wherever possible, always use existing data elements and domains for the components of the BAPI structure. This ensures consistency.
- ▶ The components of the BAPI structure should have meaningful English names. For example, use `MATERIAL` as a field name instead of `MATNR`. The field name can't exceed thirty characters. You can use report `BAPIFELD` to create a proposal for English field names for a given ABAP Data Dictionary structure.
- ▶ You should define single values or a value table for the domain of the field so that `F4` help is available to identify the possible values. This helps external programmers know the possible values that can be passed to the field. Whenever applicable, you should define a foreign key for the fields of the BAPI structure. If a value table is maintained, then it should be defined as a foreign key table for the component.
- ▶ The internal data format should be used for all parameter fields. For example, the date field should always use the format `YYYYMMDD`.

Creating a Function Module in Function Builder

Once the data structures for the BAPI parameters are defined, the underlying function module for the BAPI is created in Function Builder. We discussed creating function modules in [Chapter 7](#).

When creating the function module, the following rules apply:

- ▶ The function module name should follow the naming convention `<Name space>BAPI_<business object>_<method>`. For example, for method `Create()` of business object type `Employee`, you can name the function module `ZBAPI_EMPLOYEE_CREATE`. The name of the function module can't exceed thirty characters. If required, you can use abbreviations if you ensure that the business object type assignment is recognizable.
- ▶ All the BAPIs belonging to the same business object type should be created in one function group.

- ▶ You shouldn't put BAPIs belonging to different business object types in the same function group.
- ▶ The function module should be RFC-enabled.
- ▶ The function module should have a return parameter of type `BAPIRET2`.
- ▶ Even though underscores aren't allowed for parameter names in BOR, you can separate each word in the parameter name using the underscore in function module. For example, you can name the parameter `FirstName` in BOR as `FIRST_NAME` in the function module.
- ▶ The extension structures `ExtensionIn` and `ExtensionOut` should be of type `BAPIPAREX`. These parameters can be named `EXTENSION_IN` and `EXTENSION_OUT` in the function module.
- ▶ The function module shouldn't raise any exceptions; instead, the error message should be captured in the `RETURN` structure. The error message should be easily understandable.
- ▶ The function module shouldn't generate any screen output or display any dialog boxes.
- ▶ The function module shouldn't terminate the program. Any errors should be sent to the caller with the appropriate message type in the `RETURN` structure.
- ▶ No `COMMIT WORK` statement should be executed in the BAPI. This allows for calling multiple BAPIs in one LUW.
- ▶ Wherever possible, every BAPI should execute its own authorization checks.
- ▶ Individual parameters of the function module interface should be referred to the components of the BAPI structure.

Defining the Business Object Type in BOR

Once the function module is developed, the BAPI is defined as a method of the business object type. SAP provides a wizard that can add the function module as an API method to the business object type. Using BOR, you can define your own business object types or create a subtype for an existing business object type.

To add the BAPI as a method to the business object type, proceed with the following steps:

1. Open an existing business object type or create a new business object type as per your requirements via Transaction SWO1.

- Follow the menu path UTILITIES • API METHODS • ADD METHOD, as shown in [Figure 19.17](#). This will start a wizard, which will take you through a sequence of steps to add the function module as a method. The wizard will validate if you've followed all the rules in developing the BAPI function module.

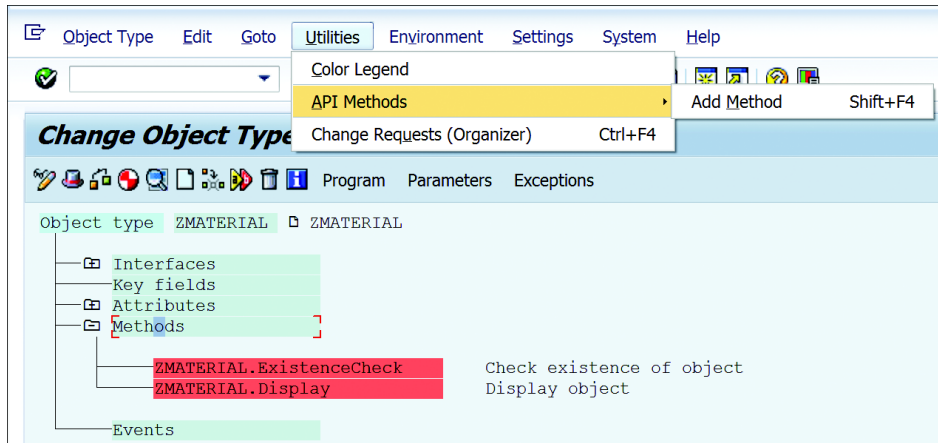


Figure 19.17 Adding BAPI as Method

- Once you start the wizard by selecting the menu path UTILITIES • API METHODS • ADD METHOD, enter the function module name that should be added as an API method, as shown in [Figure 19.18](#), and click the green checkmark.

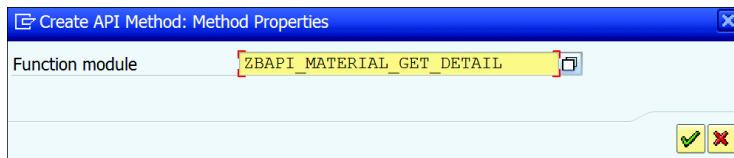


Figure 19.18 Add Function Module

- On the next screen (see [Figure 19.19](#)), verify suggested properties for the method and make any necessary changes. The DIALOG checkbox should be selected if the function module generates any dialogs that require user actions. The SYNCHRONOUS checkbox should be selected if the method is a synchronous method; if the checkbox isn't selected, the method will be an asynchronous method.

The INSTANCE-INDEPENDENT flag is an indicator that the attribute is valid for all instances of the object type. Instance independent methods deliver an object reference such as the Create or Find method. Once the necessary properties are edited, click the NEXT STEP button.

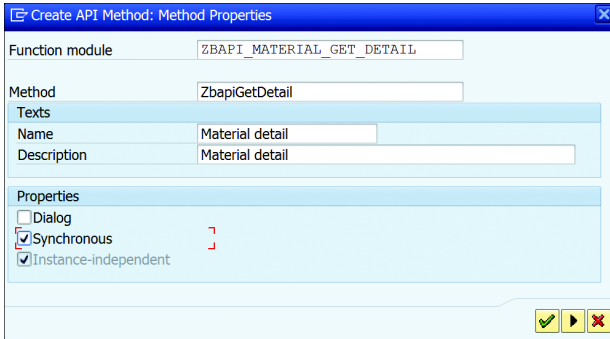


Figure 19.19 Method Properties

5. In the next step, as shown in [Figure 19.20](#), verify the method parameters and, if required, edit the parameter names for the method. Click NEXT STEP to implement the method.

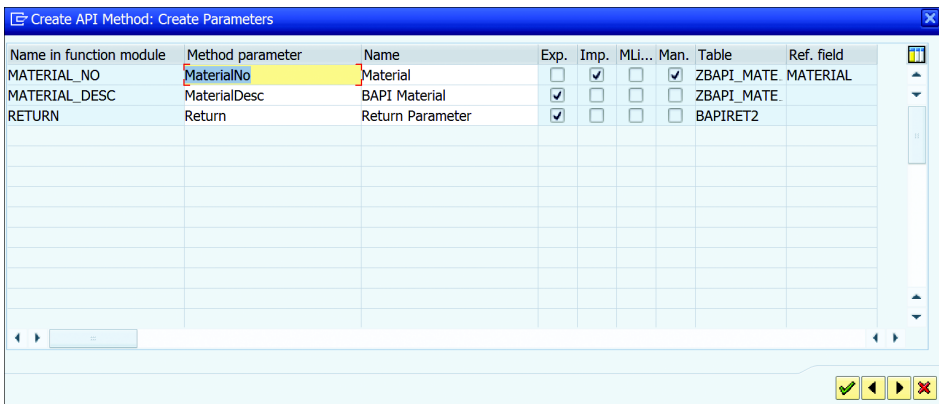


Figure 19.20 Method Parameters

6. The wizard will prompt you to generate the required template to implement the method, as shown in [Figure 19.21](#). Click YES to generate the template and implement the method automatically. At this point, the method will be added to the object type, as shown in [Figure 19.22](#).

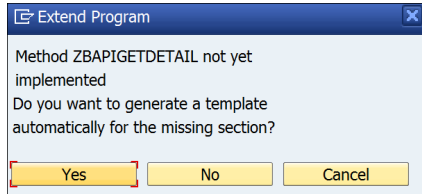


Figure 19.21 Generate Template

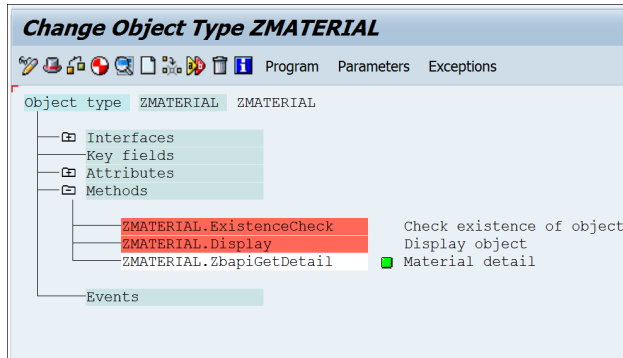


Figure 19.22 Method Added to Object Type

7. You can repeat the steps to add more methods if required. After adding the required methods, generate the object by clicking the GENERATE button in the toolbar. After the object is successfully generated, change the release status of each method using the menu path EDIT • CHANGE RELEASE STATUS • OBJECT TYPE (see Figure 19.23). The method should be set to RELEASED status to be used by external applications.

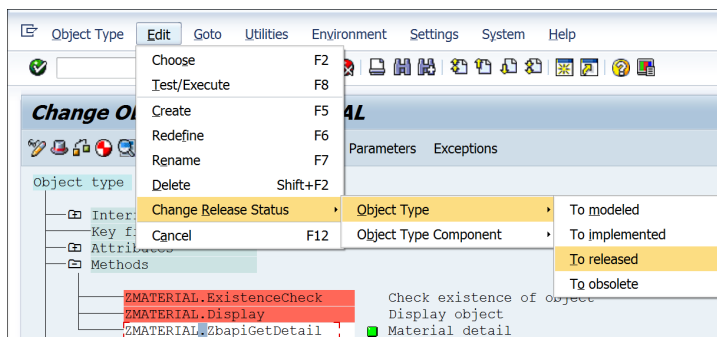


Figure 19.23 Change Release Status

8. After changing the release status of the methods, the object itself should be released so that external applications can use the business object and access its components. To change the release status of the business object, from the initial screen of Transaction SWO1, follow the menu path OBJECT TYPE • CHANGE RELEASE STATUS TO • RELEASED, as shown in [Figure 19.24](#). The business object should be saved in a transportable package for the object to be set to the RELEASED status. Once released, the object can be accessed from BAPI Explorer and used by external applications.

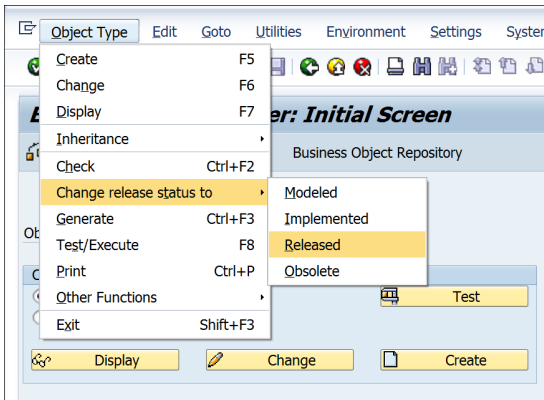


Figure 19.24 Release Object

Note

We strongly recommend not creating new BAPIs for SAP business object types, because doing so involves modification.

Documentation

Each BAPI should be thoroughly documented. The documentation can be maintained for each BAPI from within BAPI Explorer. The BAPI documentation should be maintained in such a way that an external developer with sufficient business knowledge but no knowledge of the SAP system should be able to work with the BAPI.

The documentation should clearly explain the purpose of the methods—what actions they perform and their limitations and dependencies. If the BAPI performs any COMMIT WORK process, it should be explicitly mentioned in the documentation.

All the parameters of the method should be documented to reflect their purpose and expected values. If the parameters have any default values, they should be clearly documented.

Releasing

Once the BAPI is developed and the documentation maintained, all the development objects associated with the BAPI can be released for customers:

- ▶ The function module should be released in Function Builder, using the menu path `FUNCTION MODULE • RELEASE`.
- ▶ The business object type should be released in BOR, using the menu path `OBJECT TYPE • CHANGE RELEASE STATUS TO`.
- ▶ The BAPI method should be released in BOR, using the menu path `EDIT • CHANGE RELEASE STATUS`.

In this section, we discussed the use of BAPIs in the integration and exchange of data between different systems, including communication between SAP systems and between SAP and non-SAP systems. We looked at the different conventions, methods, and steps for implementing BAPIs.

In the next section, we'll explore using IDocs for EDI and ALE technologies.

19.3 EDI/ALE/IDocs

Intermediate Documents (IDocs) are a common data exchange format. IDoc interfaces allow you to exchange data with external systems. IDocs are commonly used with Electronic Data Interchange (EDI) and Application Link Enabling (ALE) technologies to exchange data with external systems.

Working with IDocs in EDI and ALE scenarios is a complex process and requires numerous configurations, such as RFC destinations, port definitions, partner profiles, and so on to be maintained in the system. This involves the help of system administrators and functional consultants to carry out different tasks. As an ABAP programmer, your role is limited to developing IDocs and posting programs.

19.3.1 Electronic Data Interchange

EDI is a popular way to exchange business documents between companies. For example, say that company ABC typically procures raw materials from different vendors. These vendors are known as *business partners*. When company ABC wants to procure different raw materials from its business partners, it needs to send a purchase order.

In a typical scenario, a purchase order is created in company ABC's system and sent to the business partner via mail or fax. Someone at the business partner enters the purchase order details in the partner's own internal system to generate an invoice. The business partner then sends the invoice to company ABC. The operator at company ABC enters the invoice details in the system for processing.

[Figure 19.25](#) illustrates this manual process. Manually exchanging business documents is a time-consuming, expensive, and error-prone process. For example, what if the operator inputs the purchase order details incorrectly while generating the invoice due to an illegible fax?

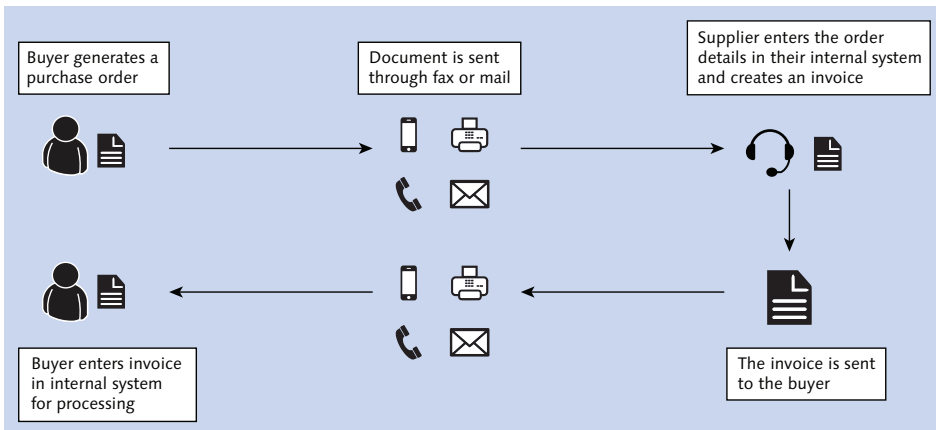


Figure 19.25 Manual Exchange of Business Documents

To make it easy for different organizations to exchange business documents, a more automated process was required. However, different organizations use different systems, and the same data can have different formats, one unique to each system. Company ABC may create a document in which the data is maintained in a specific format—for example, representing the date as YYYYMMDD.

When this document is sent to the business partner, the business partner's system may use a different format for the same data. For the purposes of our example, say the business partner's system uses an MMDDYY format for the date. When different systems use different formats to represent the same data, it becomes difficult to automate data exchange.

EDI addresses such challenges by defining a standard format for exchanging data. EDI is a computer-to-computer electronic exchange of business documents in a standard format. You may argue that sending a document via email is also an electronic way to exchange business documents—but when a business document is sent via email, you still need an operator to manually post the document in the system. EDI completely automates the process of exchanging business documents without any human intervention.

Because EDI doesn't involve exchanging paper-based documents, it provides major benefits, such as reduced costs, almost real-time exchange of documents, reduced errors, and so on. EDI completely removes human interaction in the whole process of exchanging business documents.

Figure 19.26 further illustrates the EDI process.

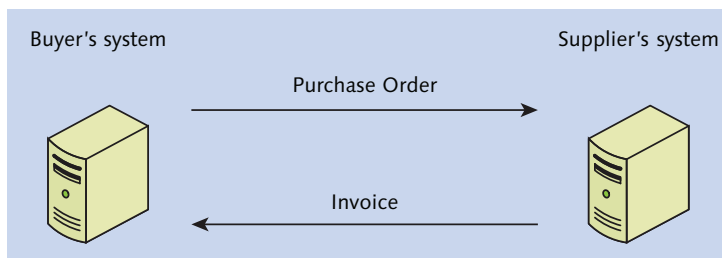


Figure 19.26 EDI Process

EDI is used to exchange many business documents, such as purchase orders, invoices, custom documents, shipping documents, and more. Because EDI improves the speed of processing documents, it provides a business advantage over competitors that still use a paper-based approach.

Now that you have a baseline understanding of the advantages of EDI, in the following subsections we'll look at how IDocs are used with EDI and the processes used to exchange business documents.

Using IDocs with EDI

Because EDI involves the exchange of documents directly between computers without any human intervention, a common standard needs to be defined, describing what data is exchanged for a business document and in what format the data is presented. For this reason, there are several EDI standards in use today, such as ANSI, EDIFACT, TRADACOMS, and so on. Each standard also has multiple versions, such as ANSI5010, EDIFACT version D12, and so on. When two business agree to exchange documents using EDI, they agree to use a specific EDI standard and version.

Businesses typically use a translator program to convert the company's internal format to an EDI-specific format before exchanging documents. For example, in an SAP system, when an application document such as a purchase order is created, the data is stored in the database in an SAP-specific format (e.g., the date is stored in YYYYMMDD format). If this application document needs to be sent to a business partner, it has to be converted first to the agreed-upon EDI standard.

Converting the document to EDI standard produces just another way of representing the same data. However, EDI standards keep evolving, and if SAP selects to take on the task of converting application documents to EDI standards, then SAP needs to keep updating its system each time an EDI standard changes. To avoid such a problem, SAP has outsourced the translation to third-party software vendors. The translator software is called the EDI subsystem. Using a third-party subsystem allows SAP not to worry about ever-changing EDI standards; it's the job of the subsystem vendors to update their translation program to keep in sync with the changing standards.

However, there's still one challenge. SAP needs the flexibility to change its own internal format and application documents. If the subsystem depends on the SAP application data directly, then each time SAP changes its application format, the subsystem needs to be updated to keep in sync with the changes.

To overcome such challenges, SAP uses the IDoc format. When the application document is created in the system, an IDoc is generated based on the application document. IDoc is another way of representing the same data. The subsystem uses the IDoc for conversion to the EDI document. SAP guarantees that no matter how an application changes internally, the subsystem always receives data in a fixed IDoc format. This allows SAP the flexibility to change its internal applications without worrying about the subsystems being affected.

The role of the subsystem is to convert the IDoc to an EDI document and transmit it to the business partner. Similarly, when the EDI transmission is received from a business partner, the EDI subsystem translates the EDI document into an IDoc. Exchange of business documents involves an outbound process, in which a document is sent from your system to a business partner's system, and an inbound process, in which a document is received from a business partner and posted in your system.

We'll discuss the inbound and outbound processes further in the following subsections.

Outbound and Inbound Processing

Figure 19.27 shows the steps involved in the outbound and inbound processes.

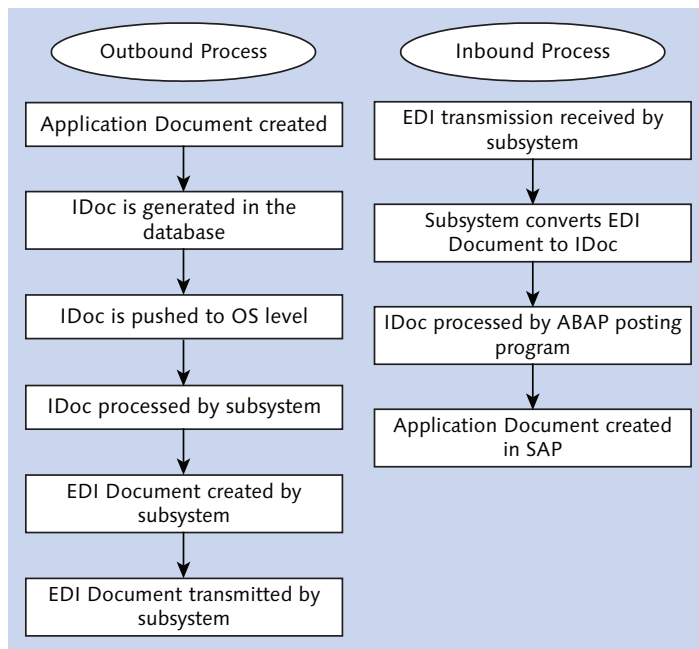


Figure 19.27 Outbound and Inbound Processes

In the outbound process, the application document is first created in the SAP system. If the document needs to be transmitted through EDI (e.g., via message control), then it triggers the EDI process.

In the EDI process, an ABAP program selects the application data from the database and generates an IDoc. This IDoc is then pushed to the OS level, to a specific directory from which the subsystem will pick up the IDoc for processing. The subsystem will first perform a syntax check to ensure the IDoc is in the correct format and all mandatory fields are filled.

The IDoc consists of a control record, multiple data records, and multiple status records:

► **Control record**

The *control record* consists of the general information about the IDoc, such as the type of document (called the message type), the business partner details, and so on. You'll learn more about the control record when we discuss IDoc development in the upcoming subsections.

► **Data records**

Data records consist of the actual data of the document.

► **Status records**

Status records are updated with the status of the IDoc at each processing step.

The subsystem converts the IDoc to EDI format. It uses the control record of the IDoc to identify the partner details and the message type being exchanged. Based on the message type, the required syntax checks are performed to ensure data consistency. The subsystem takes care of stripping the control record before transmitting the document to the business partner.

Once the IDoc is converted to an EDI document, the subsystem establishes a connection with the business partner's system (either through direct connection to the partner's network or through VAN) and transmits the document. Once the acknowledgement is received from the business partner that the document has been received successfully, the subsystem updates the IDoc with the status message. The IDoc is updated with the status message at each step of the process to help identify any issues.

In the inbound process, the EDI transmission is received by the subsystem and takes care of converting the EDI document to an IDoc. SAP provides program `STARTRFC` to start any RFC-enabled function module from the operating system level. The subsystem uses program `STARTRFC` to call the `EDI_DATA_INCOMING` function module to trigger the inbound processing. The control record of the IDoc is used to identify the partner profile, using which the ABAP posting program is

executed. The posting program processes the IDoc and posts the application document in the SAP system.

The subsystem, even though not part of SAP, plays a crucial role. It takes care of processing IDocs, performs a syntax check on the document before transmission, establishes a connection with the business partner's network to transmit the file, and acknowledges the successful transmission.

19.3.2 Application Link Enabling

ALE is a proprietary SAP technology used to create and run distributed applications. ALE allows you to integrate business processes between SAP systems and between SAP and non-SAP systems. When communicating with non-SAP systems, special SAP-certified third-party translator programs are used. ALE allows you to exchange data between application systems in a distributed environment in a controlled and consistent manner.

A distributed environment refers to a landscape in which various business functions of an organization are distributed across multiple systems. Due to increased globalization of markets, a distributed environment is required to decentralize such systems. For example, the accounts system is maintained in a central location, with manufacturing systems maintained locally at each plant.

Distributed environments also improve system performance, because different business functions run in different systems as opposed to all business functions running in the same system. ALE is middleware that allows you to integrate different components of a business process. It allows you to integrate processes from vendors and customers; business processes aren't restricted to one organization only, but instead involve vendors and customers as business partners.

The applications of ALE integrated systems are loosely coupled, and the data is exchanged asynchronously using IDocs. Synchronous connections are used to read the data only. Asynchronous exchange of data ensures that the data arrives in the receiver system even if the system is offline when the data is sent.

ALE guarantees the distribution and synchronization of master data, customization data, and transaction data. There are many configurations that need to be set up for ALE integration. Transaction SALE is used to maintain all the required settings in the system.

ALE has three layers:

- ▶ Application layer
- ▶ Distribution layer
- ▶ Communication layer

Figure 19.28 shows the different ALE layers.

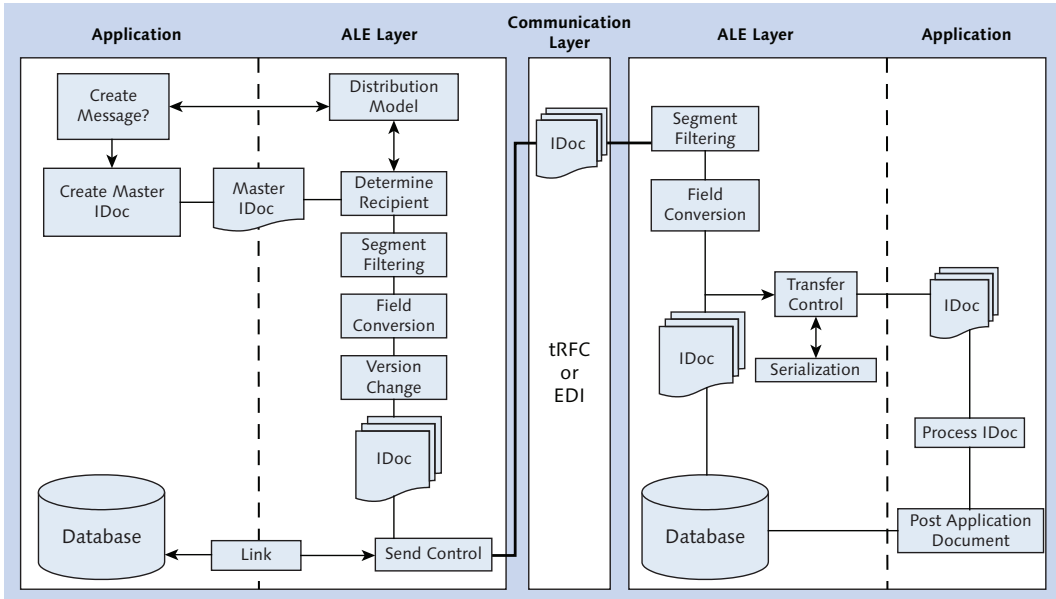


Figure 19.28 ALE Services

The outbound ALE process involves the following steps:

1. Receiver determination

The *receiver* is determined either by the application or by using the distribution model. The *distribution model* defines which applications in the distributed system communicate with each other and what data is exchanged between them.

A master IDoc is created with all the data that should be exchanged with external systems, and for each receiver system identified from the distribution model, the data specified by the filter objects in the distribution model is selected from the master IDoc.

2. Data selection

The data that needs to be distributed is selected by the application program.

3. Segment filtering

Individual segments can be removed from the IDoc before it's dispatched. For each receiving system, segment filtering is configured to set the filter objects.

4. Field conversion

If the receiver system requires any field conversions, the conversion rules can be maintained in customizing. For example, a plant field can be converted to a four-character field from a two-character field when it's being transferred from an SAP R/2 system to the latest SAP system.

5. Version change

SAP guarantees that ALE will work to exchange the documents between different SAP releases. Changing the IDoc format allows you to convert message types from different releases.

6. Dispatch control

Dispatch of IDocs can be controlled based on time and quantity. IDocs either can be sent immediately or can be scheduled to process in the background by scheduling a background job. The background job can be scheduled to run at periodic intervals. An IDoc also can be assigned a packet size so that it can be dispatched in packets. These settings are maintained in the partner's profile.

Inbound processing involves the following steps:

1. Segment filtering

Similar to outbound processing, individual segments can be removed from the IDoc on the receiver's system.

2. Field conversion

This step is similar to the outbound processing step.

3. Transfer control

Once the IDoc is written to the database, it can be posted by the application either immediately or in the background by scheduling a background job. A workflow can be used to notify the user if any error occurs while posting the document.

4. Serialization

Serialization is used to post documents in the correct sequence.

Now that you have a better understanding of EDI and ALE technology, we'll dive into the use of IDocs in the next subsection.

19.3.3 Intermediate Documents

IDocs are used to exchange data via EDI or ALE. An IDoc is just a data container that holds data in a specific data format. For example, a date field may be stored in an IDoc with an MMDDYY format.

As previously mentioned, an IDoc consists of one control record, multiple data records, and multiple status records, as shown in [Figure 19.29](#). The control record holds control information such as direction (inbound or outbound), sender and receiver information, message type, IDoc status, and so on. In [Figure 19.29](#), you'll see the control record information displayed under the SHORT TECHNICAL INFORMATION section.

IDoc Display: 000000003219745																				
Segments with Errors																				
IDoc display	Additional information	Short Technical Information																		
<ul style="list-style-type: none"> [-] IDoc 000000003219745 <ul style="list-style-type: none"> [+] Control Rec. [+] Data records <ul style="list-style-type: none"> [+] E1EDK01 [+] E1EDK14 014 [+] E1EDK14 009 [+] E1EDK14 013 [+] E1EDK14 011 [+] E1EDK03 012 [+] E1EDK03 011 [+] E1EDKA1 AG [+] E1EDKA1 LF [+] E1EDKA1 WE [+] E1EDK02 001 [+] E1EDK17 001 [+] E1EDK17 002 [+] E1EDP01 [+] E1EDS01 002 [+] Status records <ul style="list-style-type: none"> [+] 03 [+] 30 [+] 01 	Total number: 000020 Segment 000001 Segment 000002 Segment 000003 Segment 000004 Segment 000005 Segment 000006 Segment 000007 Segment 000008 Segment 000009 Segment 000010 Segment 000011 Segment 000012 Segment 000013 Segment 000014 Segment 000020	Direction: 1 Outbox Current Status: 03 Basic type: ORDERS05 Extension: Message Type: ORDERS Partner No.: 9010 Partn.Type: LI Port: XI_00_800 <hr/> Content of Selected Segment <table border="1"> <thead> <tr> <th>Fld Name</th> <th>Fld Cont.</th> </tr> </thead> <tbody> <tr><td>KZABS</td><td>X</td></tr> <tr><td>CURCY</td><td>USD</td></tr> <tr><td>HWAER</td><td>USD</td></tr> <tr><td>WKURS</td><td>1.00000</td></tr> <tr><td>ZTERM</td><td>0001</td></tr> <tr><td>BSART</td><td>NB</td></tr> <tr><td>BELNR</td><td>4500019075</td></tr> <tr><td>RECIPT_NO</td><td>0000009010</td></tr> </tbody> </table>	Fld Name	Fld Cont.	KZABS	X	CURCY	USD	HWAER	USD	WKURS	1.00000	ZTERM	0001	BSART	NB	BELNR	4500019075	RECIPT_NO	0000009010
Fld Name	Fld Cont.																			
KZABS	X																			
CURCY	USD																			
HWAER	USD																			
WKURS	1.00000																			
ZTERM	0001																			
BSART	NB																			
BELNR	4500019075																			
RECIPT_NO	0000009010																			
	Data passed to port OK IDoc ready for dispatch (ALE : IDoc generated																			

Figure 19.29 IDoc Example

The data records hold the application data. When developing an IDoc, we define segments (structures) that hold the application data. When the IDoc is generated,

the segments translate to data records. The fields of the IDoc segment and the data they contain can be seen under the CONTENT OF SELECTED SEGMENT section in [Figure 19.29](#). Each IDoc contains multiple data records.

Status records contain the IDoc status. A status record is added to the IDoc at each milestone. For example, the status code 01 is added when the IDoc is generated, status code 03 is added when it's transferred to the port, and so on. The status records show the status of the IDoc processing at any given stage.

In [Figure 19.29](#), you'll see status records 03, 30, and 01 attached to the IDoc under the STATUS RECORDS folder and displayed in a tree structure. Transaction WE47 contains all the available status codes and their descriptions, as shown in [Figure 19.30](#). These status codes should be used when updating the status record of the IDoc.

Status	Short text
00	00: used, only R/2
01	IDoc generated
02	Error passing data to port
03	Data passed to port OK
04	Error within control information of EDI subsystem
05	Error During Translation
06	Translation OK
07	Error during syntax check
08	Syntax check OK
09	Error during interchange handling
10	Interchange handling OK
11	Error during dispatch
12	Dispatch OK
13	Retransmission OK
14	Interchange Acknowledgement positive
15	Interchange Acknowledgement negative
16	Functional Acknowledgement positive
17	Functional Acknowledgement negative
18	Triggering EDI subsystem OK
19	Data transfer for test OK
20	Error triggering EDI subsystem
21	Error passing data for test
22	Dispatch OK, acknowledgement still due
23	Error during retransmission
24	Control information of EDI subsystem OK
25	Processing despite syntax error (outbound)
26	Error during syntax check of IDoc (outbound)
27	Error in dispatch level (ALE service)
28	IDoc sent to ALE distribution unit retroactively
29	Error in ALE service

Figure 19.30 IDoc Statuses

IDoc Structure

IDocs are stored internally in the database in the following database tables:

► EDIDC

The control record is stored in database table `EDIDC` with a unique IDoc number (see [Figure 19.31](#)). Each IDoc has one record in this table and is uniquely identified by the IDoc number. Other details about the IDoc, such as IDoc status, direction, receiver port, partner details, and so on, also are stored in this table.

Dictionary: Display Table

Transparent Table: `EDIDC` Active
 Short Description: Control record (IDoc)

Attributes: Delivery and Maintenance | Fields: Entry help/check | Currency/Quantity Fields

Field	Key	Ini...	Data element	Data Type	Len...	Deci...	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3		Client
DOCNUM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	EDI_DOCNUM	NUMC	16		IDoc number
DOCREL	<input type="checkbox"/>	<input type="checkbox"/>	EDI_DOCREL	CHAR	4		SAP Release for IDoc
STATUS	<input type="checkbox"/>	<input type="checkbox"/>	EDI_STATUS	CHAR	2		Status of IDoc
DOCTYP	<input type="checkbox"/>	<input type="checkbox"/>	EDI_DOCTYP	CHAR	8		IDoc Type
DIRECT	<input type="checkbox"/>	<input type="checkbox"/>	EDI_DIRECT	CHAR	1		Direction for IDoc
RCVPOR	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVPOR	CHAR	10		Receiver port (SAP System, EDI su
RCVPERT	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVPERT	CHAR	2		Partner Type of Receiver
RCVPRN	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVPRN	CHAR	10		Partner Number of Receiver
RCVSAD	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVSAD	CHAR	10		Recipient address (SADR)
RCVSMN	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVSMN	CLNT	3		SADR client (recipient)
RCVSNA	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVSNA	CHAR	1		SADR flag for international recipier
RCVSCA	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVSCA	CHAR	3		Communication type (SADR) of rec
RCVSDF	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVSDF	CHAR	1		SADR default flag for recipient add
RCVSLF	<input type="checkbox"/>	<input type="checkbox"/>	EDI_RCVSLF	CHAR	3		Sequential Number of Recipient Ac

Figure 19.31 Table EDIDC

► EDID4

The data records of the IDoc are stored in this table, linking the control record with the document number. There can be one to many records for each IDoc. This table stores the segment information and the application data for the IDoc. The `SDATA` field in table `EDID4` contains the application data. This field is parsed in the program at runtime to a structure of the type of the IDoc segment.

When developing an IDoc, a segment is defined that contains the fields to store the application data. *Segments* are stored in ABAP Data Dictionary as structure types.

For example, say you want to generate an IDoc for a billing document with data such as document number, payer, and net value. To store this data about the billing document in an IDoc, define a segment with fields such as VBELN, KUNRG, and NETWR to store the document number, payer, and net value, respectively. This segment translates to a data record when the IDoc is generated and stored in table EDID4 with the segment name and corresponding data of the segment fields. The segment name is stored in the SEGNAM field, and the data of the segment fields is stored in the SDATA field of table EDID4.

Dictionary: Display Table

Transparent Table: EDID4 Active
 Short Description: IDoc Data Records from 4.0 onwards

Attributes: Delivery and Maintenance, Fields, Entry help/check, Currency/Quantity Fields

Field	Key	Ini...	Data element	Data Type	Length	De...	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3	0	Client
DOCNUM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	EDI_DOCNUM	NUMC	16	0	IDoc number
COUNTER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	EDI_CLUSTC	NUMC	3	0	Counter in cluster table
SEGNUM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	IDOCDSGNUM	NUMC	6	0	Number of SAP segment
SEGNAM	<input type="checkbox"/>	<input type="checkbox"/>	EDI_SEGNAM	CHAR	30	0	Name of SAP segment
PSGNUM	<input type="checkbox"/>	<input type="checkbox"/>	EDI_PSGNUM	NUMC	6	0	Number of the hierarchically higher S
HLEVEL	<input type="checkbox"/>	<input type="checkbox"/>	EDI_HLEVEL	NUMC	2	0	Hierarchy level
DATAAGING	<input type="checkbox"/>	<input type="checkbox"/>	DATA_TEMPERATURE	DATS	8	0	Data Filter Value for Data Aging
DTINT2	<input type="checkbox"/>	<input type="checkbox"/>	EDI_DTINT2	INT2	5	0	Length field for VARC field
SDATA	<input type="checkbox"/>	<input type="checkbox"/>	EDI_SDATA	LCHR	1000	0	Application data

Figure 19.32 Table EDID4

► EDIDS

The status records of the IDoc are stored in this table and linked to the control and data records using the IDoc number. Table EDIDS contains different information about the status record, such as the IDoc status, status code, time and date of status record, and so on, as shown in [Figure 19.33](#).

Dictionary: Display Table

Transparent Table: EDIDS Active
 Short Description: Status Record (IDoc)

Attribu... Delivery and Maintenance Fiel... Entry help/check Currency/Quantity Fields

Srch Help Predefined Type 1 / 26

Field	Key	Ini...	Data element	Data Type	Length	Deci...	Short Description
MANDT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	MANDT	CLNT	3		0 Client
DOCNUM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	EDI DOCNUM	NUMC	16		0 IDoc number
LOGDAT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	EDI LOGDAT	DATS	8		0 Date of status information
LOGTIM	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	EDI LOGTIM	TIMS	6		0 Time of status information
COUNTR	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	EDI COUNTR	NUMC	16		0 IDoc status counter
CREDAT	<input type="checkbox"/>	<input type="checkbox"/>	EDI CREDAT	DATS	8		0 Date status record was created
CRETIM	<input type="checkbox"/>	<input type="checkbox"/>	EDI CRETIM	TIMS	6		0 Time status record was created
STATUS	<input type="checkbox"/>	<input type="checkbox"/>	EDI STATUS	CHAR	2		0 Status of IDoc
UNAME	<input type="checkbox"/>	<input type="checkbox"/>	EDI UNAME	CHAR	12		0 User name
REPID	<input type="checkbox"/>	<input type="checkbox"/>	EDI REPID	CHAR	30		0 Program Name
ROUTID	<input type="checkbox"/>	<input type="checkbox"/>	EDI ROUTID	CHAR	30		0 Name of subroutine (routine, function mo
STACOD	<input type="checkbox"/>	<input type="checkbox"/>	EDI STACOD	CHAR	8		0 Status code
STATXT	<input type="checkbox"/>	<input type="checkbox"/>	EDI STATX	CHAR	70		0 Text for status code
SEGNUM	<input type="checkbox"/>	<input type="checkbox"/>	IDOCSSGNUM	NUMC	6		0 Number of SAP segment
SEGFLD	<input type="checkbox"/>	<input type="checkbox"/>	EDI SEGFLD	CHAR	30		0 Field Name in SAP Segment

Figure 19.33 Table EDIDS

IDoc Development and Tools

An IDoc is an instance of the IDoc basic type. The IDoc is the data container, and the IDoc basic type defines the IDoc structure. The basic type consists of one or more segments, which exist as structures in ABAP Data Dictionary. The IDoc segments provide the structure to parse the application data from the `SDATA` field of the data record.

When creating an IDoc basic type, we perform two basic steps: First, define the segments containing various fields for the application data that should be saved in an IDoc, and then add these segments to the IDoc basic type. IDoc basic types are created using Transaction WE30, whereas segments are defined using Segment Editor (Transaction WE31).

After creating the basic type with the required segments, the basic type should be linked to a *logical message type*, which describes the business process, and each basic type should be linked to a business process via the logical message type. For example, the `ORDERS` logical message type usually is used for purchase orders in

outbound processing, and if you're creating a basic type to pass purchase order data to the external system, then the basic type should be linked to the `ORDERS` logical message type.

You can link the basic type to an existing logical message type or create a new logical message type if none of the existing message types describe your business process adequately. One basic type can be assigned to several logical message types and vice versa, so you should create a new message type only if none of the existing message types adequately describe your business process.

The logical message types are defined using Transaction WE81, whereas the basic type is linked to the message type using Transaction WE82.

Now that you have a better idea of the structure, types, and tools of IDocs, in the following subsections we'll walk you through IDoc development and the required configuration steps.

Creating a Segment

Use Transaction WE31 to create a segment. The segment is added to the basic type to define the structure of the IDoc.

Proceed as follows to create a segment:

1. On the initial screen of Transaction WE31 (see [Figure 19.34](#)), provide the name of the segment and click the **CREATE** button in the application toolbar. SAP segment names start with `E1`, and customer segment names should start with `Z1`.

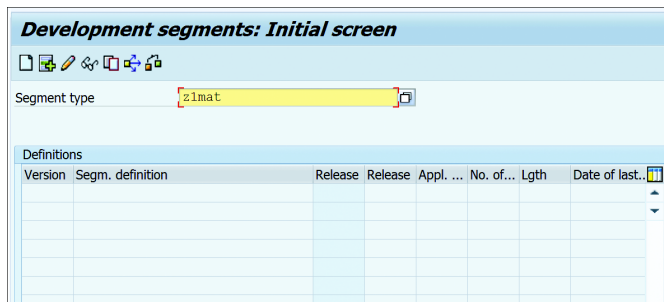


Figure 19.34 Create Development Segment Initial Screen

2. On the **CREATE SEGMENT DEFINITION** screen, as shown in [Figure 19.35](#), maintain the fields for the segment just like you would maintain the components of a

structure. For example, enter the field name in the FIELD NAME column and the data element name in the DATA ELEMENT column. Once the fields of the segment are maintained, click the SAVE icon in the standard toolbar to save the segment. The fields of the segment translate to data records in the IDoc.

3. If the segment has a different meaning in different scenarios, then you can select the QUALIFIED SEGMENT checkbox (see [Figure 19.35](#)). When this checkbox is set, the meaning of the segment is derived from the first field of the segment.

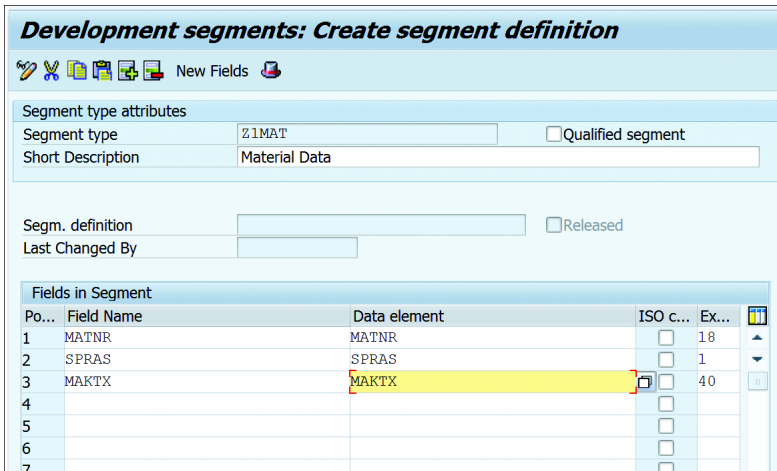


Figure 19.35 Create Segment Definition

4. Once the segment is saved, release the segment by following the menu path EDIT • SET RELEASE from the initial screen, as shown in [Figure 19.36](#).

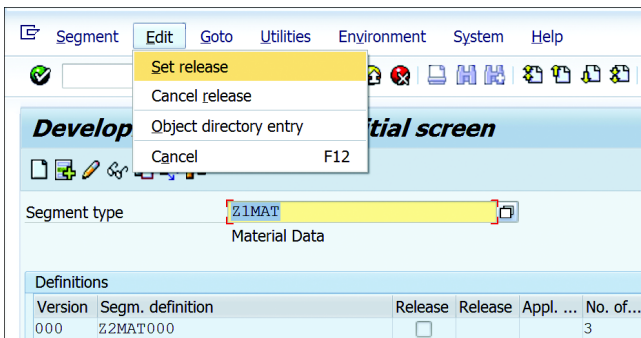


Figure 19.36 Releasing Segment

Similarly, you can create additional segments as per your requirements.

Creating a Basic Type

Once the segment is defined using Segment Editor, we can define the basic type using Transaction WE30.

Proceed with the following steps to create the basic type:

1. On the initial screen of Transaction WE30 (see [Figure 19.37](#)), you can create a new basic type or create an extension for the existing basic type. To create a new basic type, select the BASIC TYPE radio button. If you want to create an extension type, enter the name of the basic type for which you want to create an extension and select the EXTENSION radio button. For this example, create a new basic type.

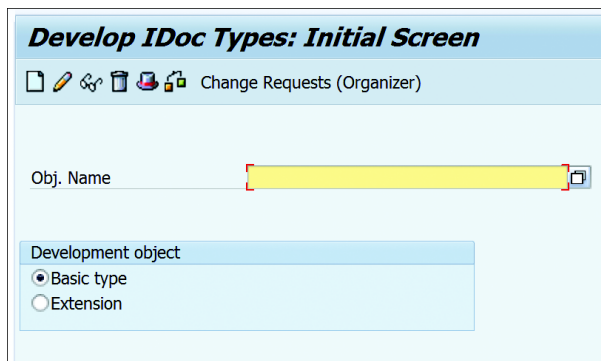


Figure 19.37 Develop IDoc Types: Initial Screen

2. Once you provide the name of the basic type and click the CREATE button, you'll see a dialog box in which you can create a basic type from scratch, copy from an existing basic type, or create a successor to an existing basic type, as shown in [Figure 19.38](#). Select the CREATE NEW radio button to create a new basic type.
3. Now, you'll see a screen in which you can add segments to the basic type. Click the CREATE button, as shown in [Figure 19.39](#).

Figure 19.38 Create Basic Type

Figure 19.39 Add Segment

- In the MAINTAIN ATTRIBUTES dialog box that appears (see [Figure 19.40](#)), enter the name of the segment type that you created previously. You can also set the segment as mandatory. If the MANDATORY SEG. checkbox is selected, then the IDoc will be checked at runtime to validate if all the mandatory segments are filled.

The MAXIMUM NUMBER and MINIMUM NUMBER fields allow you to set how many times the segment can be repeated in the IDoc. For example, if the segment contains header fields, then you can set it to repeat only once. Similarly, if the segment contains item records, you can set it to repeat multiple times.

The SEGMENT EDITOR button takes you to the segment editor, in which you can check the segment definition.

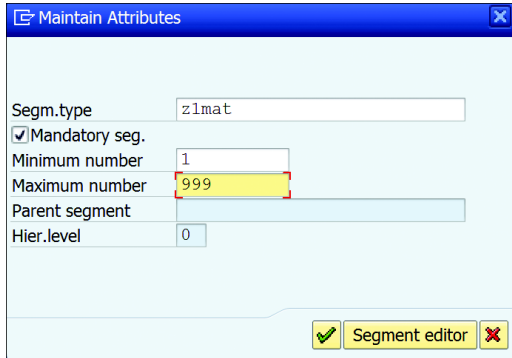


Figure 19.40 Maintain Attributes

5. Click OK to add the segment to the basic type. Similarly, you can add additional segments to the basic type. You can add new segments as child segments by keeping the cursor on the existing segment and clicking the CREATE icon. Doing so will open a dialog box asking you whether you want to add the new segment type at the same level or as a child segment, as shown in [Figure 19.41](#).

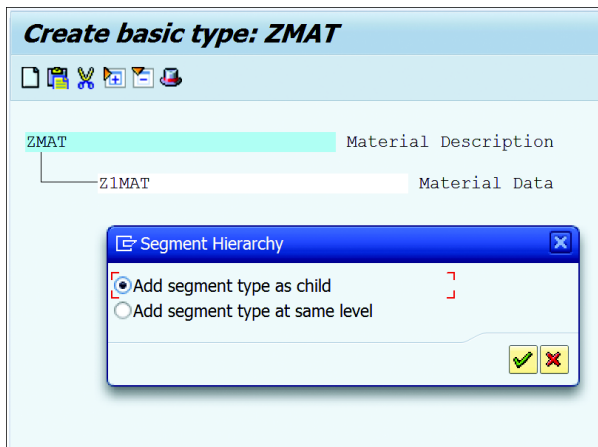


Figure 19.41 Adding Additional Segments

6. Once all the required segments are added to the basic type, save it. The basic type should be released to freeze any further changes. To release the basic type, follow the menu path EDIT • SET RELEASE from the initial screen of Transaction WE30.

7. You can check the consistency of the basic type by following the menu path DEVELOPMENT OBJECT • CHECK from the initial screen. At this point, when the basic type is checked for consistency, the system will show a warning message indicating that the basic type isn't linked to any logical type, as shown in [Figure 19.42](#).

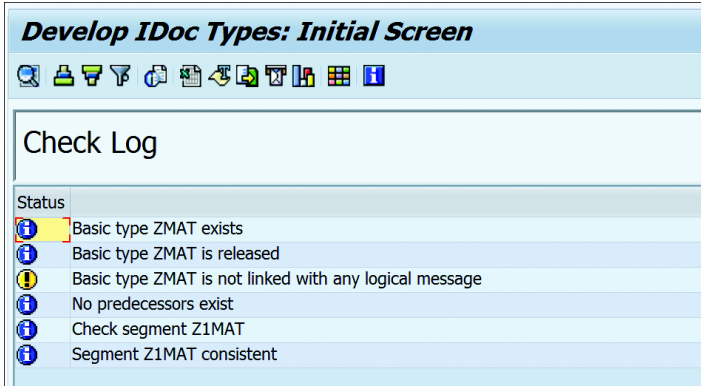


Figure 19.42 Consistency Check

The logical message type identifies the type of document being exchanged. Each basic type is linked to a logical message type. For example, the `ORDERS` message type is used for purchase orders, `ORDRSP` is for order confirmation, and so on. You can create a new logical type and link your basic type to it. If we want to extend an existing logical type, create a new extension type and add the extension type to the existing logical type. In the next subsection, we'll explain the steps for creating a logical type.

Creating a Logical Type

You can create a new logical type using Transaction WE81. Proceed with the following steps to create a logical type:

1. On the initial screen of Transaction WE81, enter change mode and click the NEW ENTRIES button to maintain the new logical type and its description, as shown in [Figure 19.43](#).
2. Once the message type and the short description is maintained, click the SAVE button in the toolbar to save the new logical message type. The basic type can now be linked to the new logical message type. The logical message type plays

a central role in linking different objects of IDoc processing, such as partner profiles and inbound and outbound process codes.

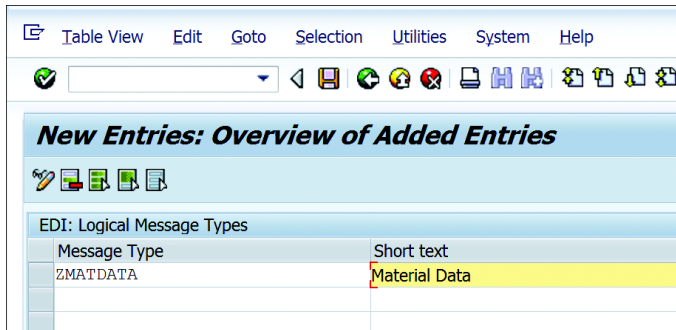


Figure 19.43 Maintain Message Type

Assigning Basic Types

After the logical message type is maintained, the basic type is linked to the message type using Transaction WE82.

The following steps walk you through how to assign and link the basic type you've created:

1. On the initial screen of Transaction WE82, enter change mode and click the NEW ENTRIES button. Maintain the message type and basic type details, as shown in Figure 19.44.

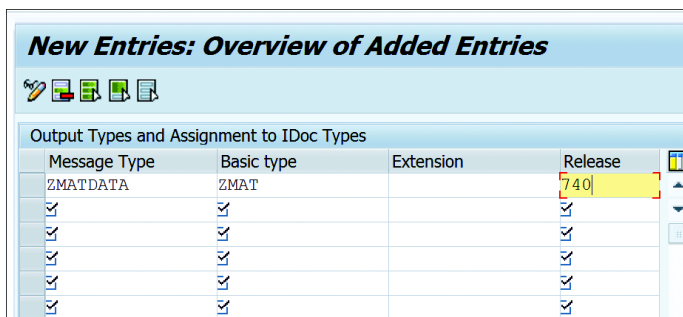


Figure 19.44 Assigning Basic Type to Message Type

2. The RELEASE column should be filled with the SAP release version to which this message type and basic type are linked. This helps the system use the correct

basic type based on the system release. One message type can have multiple basic types (a one to many [1:n] relation).

3. Use the menu path **SYSTEM • STATUS** to find the release version of your system under the **SAP SYSTEM DATA** section, as shown in [Figure 19.45](#).

The screenshot shows two windows. The left window, titled 'SAP System data', has a 'Product Version' field with a '- See Details -' button and a magnifying glass icon. An arrow points from this icon to the right window. The right window, titled 'Installed Software Component Versi...', displays a table of installed product versions.

Component	Release	SP-Level	Support Package	Type	Short
SAP_BASIS	740	0009	SAPKB74009	SAP	
SAP_ABA	740	0009	SAPKA74009	Cros	
SAP_GWEND	740	0009	SAPK-740009INSAPGWEND	SAP	

Figure 19.45 SAP Product Version

Before IDocs can be generated, there are many configurations that need to be maintained in the system. Most of these configurations are maintained by system administrators. In the following subsection, we'll walk through the different system configurations.

19.3.4 System Configurations

There are configurations that need to be maintained to work with IDocs. There are certain configurations that are specific to EDI and certain configurations that are specific to ALE. The following sections describe some common configurations.

The different configuration steps involved are as follows:

1. Define an RFC destination using Transaction SM59.
2. Define a port definition using Transaction WE21.
3. Define a partner profile using Transaction WE20.
4. Define an outbound process code using Transaction WE41.
5. Define an inbound process code using Transaction WE42.
6. Define IDoc attributes using Transaction BD51.

On the SAP Easy Access screen, type "WEDI" in the command field to open the **IDOC AND EDI BASIS** menu, as shown in [Figure 19.46](#); from this menu, all the tools discussed in the following sections can be accessed.

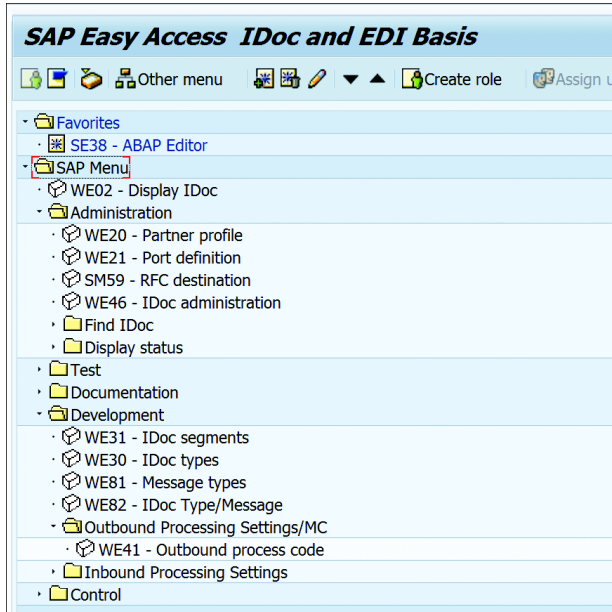


Figure 19.46 IDoc and EDI Basis Menu

RFC Destination (Transaction SM59)

The RFC destination is maintained by the system administrator using Transaction SM59, from which different types of connections can be maintained, as shown in Figure 19.47.

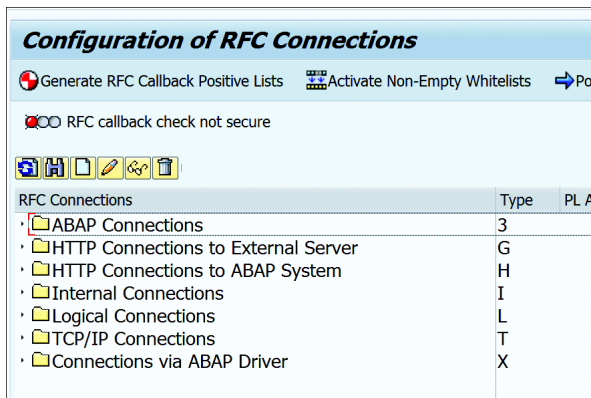


Figure 19.47 RFC Destinations

For ALE, an ABAP connection (TYPE 3) is defined, whereas for EDI, a TCP/IP connection (TYPE T) is defined. Within the RFC destination (see [Figure 19.48](#)), the remote server details and the logon credentials for the remote server are maintained. The system uses this information to connect to the remote server.

The remote server details are maintained under the TECHNICAL SETTINGS, tab and the logon credentials for the remote system are maintained under the LOGON & SECURITY tab.

The UNICODE tab allows you to maintain details about the communication type, such as Unicode or non-Unicode. Under the SPECIAL OPTIONS tab, you can maintain the settings to activate an RFC trace and the time-out duration if the remote system doesn't respond. All these settings are maintained by the SAP Basis administrator; a discussion of these individual settings is beyond the scope of this book.

RFC Destination 0M1RLN0

Remote Logon Connection Test Unicode Test

RFC Destination: 0M1RLN0

Connection Type: 3 ABAP Connection Description

Description

Description 1: RFC destination for logical system 0M1RLN0

Description 2:

Description 3:

Administrat... **Technical Settin...** Logon & Secur... Unicode Special Options

Target System Settings

Load Balancing Status

Load Balancing: Yes No

Target Host: ciec3.wdf.sap.corp Instance No.: 00

Save to Database as

Save as: Host IP Address ciec3.wdf.sap.corp

Figure 19.48 RFC Destination for ABAP Connection

In addition to maintaining RFC destinations statically via Transaction SM59, with the release of SAP NetWeaver 7.5, SAP added support for dynamic RFC destinations. A new `CL_DYNAMIC_DESTINATION` class is available to manage dynamic RFC destinations. The `CREATE_RFC_DESTINATION` method of the `CL_DYNAMIC_DESTINATION` class makes it possible to create a dynamic destination, which can be used in current sessions for RFCs.

Port Definition (Transaction WE21)

The port definition is maintained using Transaction WE21. The *port definition* is an important configuration that tells the system the physical/logical directory in which the IDoc file will be stored when using EDI or the RFC destination for a transactional RFC (tRFC) when using ALE.

When using a tRFC, the called function module is executed exactly once in the called system. This ensures that the function module isn't restarted automatically if an error occurs.

For EDI, we maintain a file port, and for ALE, we maintain a tRFC port, as shown in [Figure 19.49](#). If the IDoc is transmitted as an XML file, then an XML file port is maintained.

Defining ports is the responsibility of the SAP Basis administrator; our concern as ABAP developers is limited to knowing the port definition that should be used while generating the IDoc.

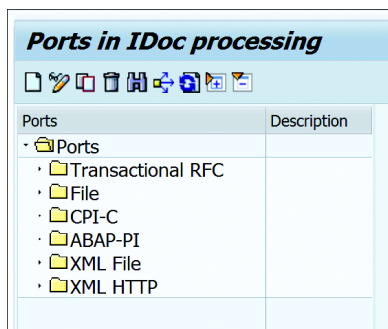


Figure 19.49 Port Definition

Partner Profile (Transaction WE20)

A *partner profile* is one of the important configurations that needs to be maintained in the system before any data can be exchanged with a business partner.

The partner type is a key field for maintaining the partner profile. [Figure 19.50](#) shows different types of partners under which business partner profile is maintained. In a distributed environment, the receiving system details are maintained under PARTNER TYPE-LOGICAL SYSTEM (LS).

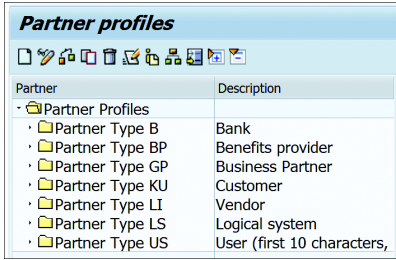


Figure 19.50 Partner Profile

For each business partner, the outbound and inbound parameters are maintained. The outbound parameters contain information such as the message type, basic type, port definition, outbound process code, and so on. The inbound parameters contain information such as the inbound process code, which identifies the inbound posting program that should be triggered when an inbound transmission is received from the partner.

You can also maintain agent details for who can be informed in case of any errors in processing. The system will trigger a workflow in which the agent can take actions to correct the errors.

Figure 19.51 shows the outbound and inbound parameters maintained for a partner profile.

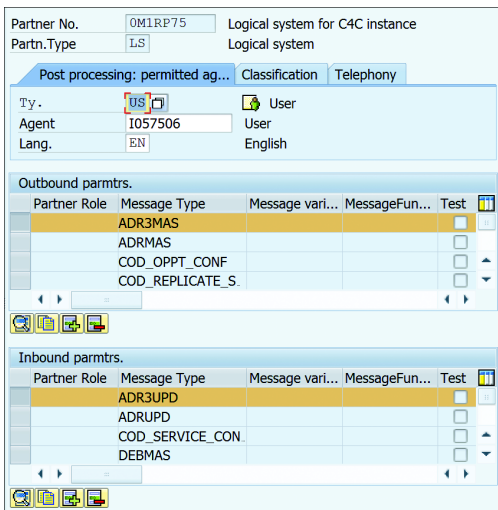


Figure 19.51 Outbound and Inbound Parameters for Partner Profile

Outbound Process Code (Transaction WE41)

The *outbound process code* links the extraction program that reads the application data and generates the IDoc with the message type. The extraction and posting programs are maintained as function modules.

In the outbound process code, we define a process code to which the function module that extracts the application data and generates the IDoc is linked (see [Figure 19.52](#)). This process code is then assigned to the outbound parameter of the partner profile under the MESSAGE CONTROL settings, as shown in [Figure 19.53](#). Message control is a mechanism to automate the output or follow up with processing of partner-dependent messages.

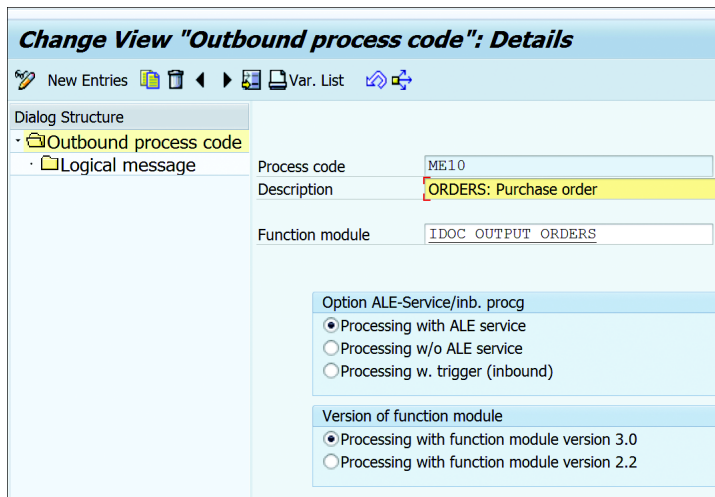


Figure 19.52 Outbound Process Code

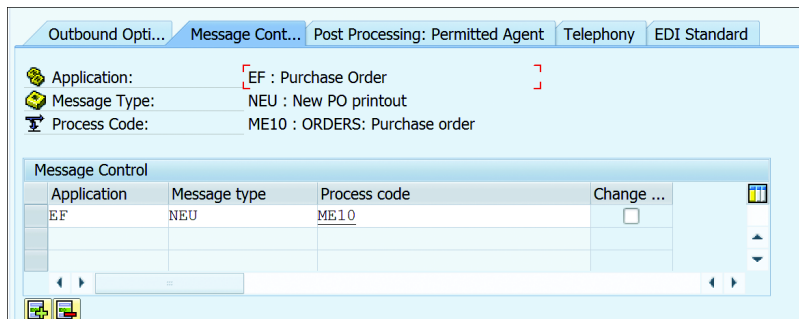


Figure 19.53 Process Code in Partner Profile

When message control is used to initiate EDI/ALE process, the partner profile is read to identify the outbound process code from which the extraction program is triggered.

Inbound Process Code (Transaction WE42)

Similar to the outbound process code, the *inbound process code* links the message type with the posting program. The inbound process code is maintained in the partner profile. When an inbound IDoc is received, the process code is read from the partner profile to identify the posting program. The posting program then processes the IDoc to post the application document.

Figure 19.54 shows the inbound process code linking to the function module. Figure 19.55 shows the process code maintained in the inbound parameters of the partner profile.

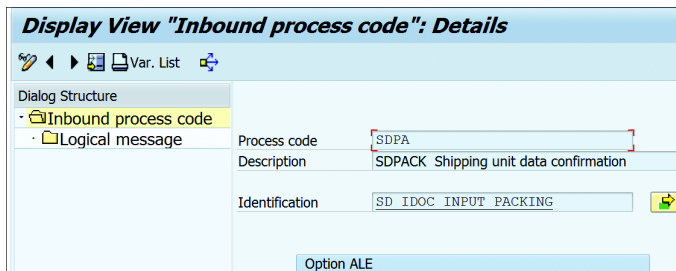


Figure 19.54 Inbound Process Code

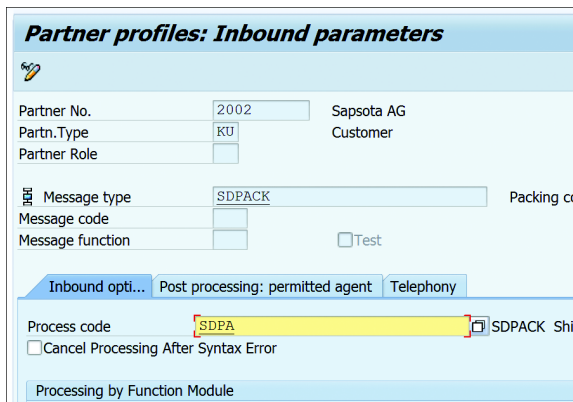
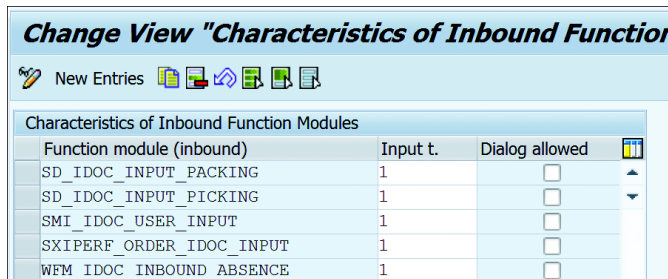


Figure 19.55 Process Code Maintained in Partner Profile

Define IDoc Attributes (Transaction BD51)

IDoc attributes define how the inbound IDoc functions are processed. In this transaction, you maintain the inbound function module name and the processing attribute with one of the following options (see [Figure 19.56](#)):

- ▶ 0
Mass processing for functions that use direct input to the database to post the application document (direct input method).
- ▶ 1
Individual input for functions that use the `CALL TRANSACTION` method to post the application document.
- ▶ 2
Individual input for functions that use the `CALL TRANSACTION` method and lock the IDoc.
- ▶ DIALOG ALLOWED
Flag that allows screen display when IDocs are processed in the foreground and the function uses the `CALL TRANSACTION` method.



Function module (inbound)	Input t.	Dialog allowed
SD_IDOC_INPUT_PACKING	1	<input type="checkbox"/>
SD_IDOC_INPUT_PICKING	1	<input type="checkbox"/>
SMT_IDOC_USER_INPUT	1	<input type="checkbox"/>
SXIPERF_ORDER_IDOC_INPUT	1	<input type="checkbox"/>
WFM_IDOC_INBOUND_ABSENCE	1	<input type="checkbox"/>

Figure 19.56 IDoc Characteristics

19.3.5 Inbound/Outbound Programs

Once the basic type is developed and all the required configurations are maintained, we need ABAP programs to extract the application data and create an IDoc in the database for outbound IDocs and to post the application data from an inbound IDoc. We'll look at inbound and outbound programs in the following subsections.

Inbound Program

In an *inbound program*, the function module should use the standard interface using the EDIDC and EDIDD structures. The IDoc control record will be filled in the EDIDC structure, which can be used to validate if the inbound program is called for the correct message type. The EDIDD structure is an internal table that contains the data records. The SDATA field of the EDIDD structure contains the unformatted row, which can be parsed to a structure of the correct segment type. This row is then updated in the database.

The easy way to define an interface is to copy the interface from an existing inbound function module. [Listing 19.2](#) shows the inbound function code in its bare minimum form. This inbound function module is configured in Transaction WE42.

```

FUNCTION ZDEMO_IDOC_INPUT.
*"-----
**"Local Interface:
*" TABLES
*"     IDOC_CONTRL STRUCTURE EDIDC
*"     IDOC_DATA STRUCTURE EDIDD
*"     IDOC_STATUS STRUCTURE BDIDOCSTAT
*"-----
DATA : wa_makt TYPE zmakt,
       st_seg TYPE zseg1,
       it_makt TYPE STANDARD TABLE OF zmakt.

LOOP AT IDOC_DATA WHERE DOCNUM EQ IDOC_CONTRL-DOCNUM.
  st_seg = IDOC_DATA-SDATA.
  MOVE-CORRESPONDING st_seg TO wa_makt.
  APPEND wa_makt TO it_makt.
ENDLOOP.
MODIFY zmakt FROM TABLE it_makt.
IF sy-subrc IS INITIAL.
  IDOC_STATUS-DOCNUM = IDOC_CONTRL-DOCNUM.
  IDOC_STATUS-STATUS = '53'.
  IDOC_STATUS-MSGTY = 'I'.
  IDOC_STATUS-MSGID = 'YM'.
  IDOC_STATUS-MSGNO = '004'.
  APPEND IDOC_STATUS.
ELSE.
  IDOC_STATUS-DOCNUM = IDOC_CONTRL-DOCNUM.
  IDOC_STATUS-STATUS = '51'.
  IDOC_STATUS-MSGTY = 'E'.
  IDOC_STATUS-MSGID = 'YM'.

```



```

IDOC_STATUS-MSGNO = '004'.
APPEND IDOC_STATUS.
ENDIF.
ENDFUNCTION.

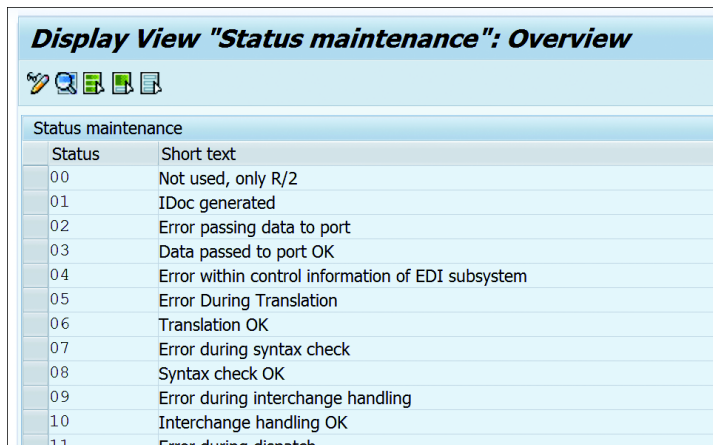
```

Listing 19.2 Inbound Function

In [Listing 19.2](#), we defined a function module with three tables parameters. The `IDOC_CONTRL` parameter is of type `EDIDC`, the `IDOC_DATA` parameter is of type `EDIDD`, and the `IDOC_STATUS` parameter is of type `BDIDOCSTAT`. When the function module is called, the IDoc control record is transferred to the `IDOC_CONTRL` parameter and the data records are transferred to the `IDOC_DATA` parameter. We also defined a local structure, `st_seg`, which is of the segment type used in the basic type. The IDoc's data records are based on this segment type.

We first move the data in the `SDATA` field into the `st_seg` structure. This data is then moved to the internal table from which the corresponding database table is updated. Note that we kept the code very basic in this example to highlight how the data is processed from the IDoc and updated in the application table.

We updated the status in the `IDOC_STATUS` parameter, which is updated in the IDoc as a status record. The status codes are maintained in the system, and only these codes should be used to update the IDoc. You can find all the status codes via Transaction `WE47`, as shown in [Figure 19.57](#).



Display View "Status maintenance": Overview

Status	Short text
00	Not used, only R/2
01	IDoc generated
02	Error passing data to port
03	Data passed to port OK
04	Error within control information of EDI subsystem
05	Error During Translation
06	Translation OK
07	Error during syntax check
08	Syntax check OK
09	Error during interchange handling
10	Interchange handling OK
11	Error during dispatch

Figure 19.57 IDoc Status Codes

Outbound Program

In an *outbound program*, we extract the application data from the database and fill the EDIDD structure. The MASTER_IDOC_DISTRIBUTE function module is used to generate the IDoc. This outbound function module is configured in Transaction WE41.

Listing 19.3 shows sample code for an outbound function module. Again, we have kept the code to the bare minimum to make it easy for you to follow the logic.

```

FUNCTION ZDEMO_IDOC_OUTPUT.
*"-----
**"Local Interface:
*"  IMPORTING
*"    VALUE(OBJECT) LIKE  NAST STRUCTURE  NAST
*"    VALUE(CONTROL_RECORD_IN) LIKE  EDIDC STRUCTURE  EDIDC
*"  EXPORTING
*"    VALUE(CONTROL_RECORD_OUT) LIKE  EDIDC STRUCTURE  EDIDC
*"  TABLES
*"    IDOC_DATA STRUCTURE  EDIDD
*"    IDOC_STATUS STRUCTURE  BDIDOCSTAT
*"-----
"Segment Type
DATA : S_ZMAKTID LIKE zseg1.
"Application table
DATA : T_makt TYPE STANDARD TABLE OF zmakt.
DATA : w_makt TYPE zmakt.
DATA : v_matnr TYPE matnr.
"Generated Communication IDoc
DATA : T_COMM_IDOC TYPE STANDARD TABLE OF EDIDC.
CONSTANTS :
C_ZMAKTID LIKE EDIDD-SEGNAM VALUE 'ZSEG1', "Segment name
C_IDOCTP LIKE EDIDC-IDOCTP VALUE 'ZDMIDOC'. "Basic Type

*Fill control record
MOVE control_record_in TO control_record_out.
control_record_out-direct = '1'.
control_record_out-serial = sy-datum.
control_record_out-serial+8 = sy-uzeit.

*Fetch application data
v_matnr = object-objky.
SELECT matnr spras maktx FROM makt INTO TABLE t_makt WHERE matnr EQ v_
matnr.

*Fill data records
LOOP AT t_makt INTO w_makt.

```

```

S_ZMAKTID-matnr = w_makt-matnr.
S_ZMAKTID-spras = w_makt-spras.
S_ZMAKTID-maktx = w_makt-maktx.
IDOC_DATA -SEGNAM = C_ZMAKTID.
IDOC_DATA -SDATA = S_ZMAKTID.
APPEND IDOC_DATA.
CLEAR IDOC_DATA.
ENDLOOP.

*Send IDoc
CALL FUNCTION 'MASTER_IDOC_DISTRIBUTE'
  EXPORTING
    MASTER_IDOC_CONTROL          = control_record_out
  TABLES
    COMMUNICATION_IDOC_CONTROL  = T_COMM_IDOC
    MASTER_IDOC_DATA            = IDOC_DATA
  EXCEPTIONS
    ERROR_IN_IDOC_CONTROL       = 1
    ERROR_WRITING_IDOC_STATUS   = 2
    ERROR_IN_IDOC_DATA         = 3
    SENDING_LOGICAL_SYSTEM_UNKNOWN = 4
    OTHERS                      = 5.
IF SY-SUBRC <> 0.
*Update IDoc status
ENDIF.

```

Listing 19.3 Outbound Program

In [Listing 19.3](#), we selected the data from the application table and filled the IDoc data structure. This data is then sent to the `MASTER_IDOC_DISTRIBUTE` function module along with the control record to generate the IDoc.

In this section, we discussed the steps for creating and generating IDocs to be used with EDI and ALE technologies. In the next section, we'll look at a tool used to migrate data from legacy systems.

19.4 Legacy System Migration Workbench

Legacy System Migration Workbench (LSWM) is a tool provided by SAP to quickly migrate data from legacy systems. This tool targets functional consultants and end users and doesn't require any ABAP knowledge. The tool provides different options to migrate data, such as the direct input method, batch input method, BAPIs, or IDocs.

Using the tool simply involves following a sequence of steps. The major step is to map the input file fields with the data transfer structure generated from the data transfer technique used. LSMW is generally employed for one-time data transfers from non-SAP to SAP systems.

19.4.1 Getting Started

To get started, proceed with the following steps:

1. Open Transaction LSMW.
2. On the initial screen, as shown in [Figure 19.58](#), enter names for the following fields:
 - ▶ **PROJECT:** The project name corresponds to an organizational unit that is used to combine mapping and field assignments. A project can have any number of subprojects.
 - ▶ **SUBPROJECT:** A *subproject* is an organizational unit that is subordinated to a project. For example, all the data that should be transferred will be a project, and if the data needs specific changes for each plant, then each plant can be a subproject. A subproject can have any number of objects.
 - ▶ **OBJECT:** An object constitutes a business object—for example, customer master, material master, and so on.

Once you've finished providing names for the fields, click **CREATE**.

Figure 19.58 LSMW Initial Screen

3. The system will open the **CREATE OBJECT** dialog box, in which you can maintain descriptions for the **PROJECT**, **SUBPROJECT**, and **OBJECT**, as shown in [Figure 19.59](#). Once the project is created, click the **EXECUTE** button to start maintaining the migration tasks.

Project	DEMO_PROJECT
Subproject	DEMO_SUBPROJ
Object	MATMAS
Name	Material Master

Figure 19.59 Create Object

19.4.2 Migration Process Steps

Working with LSMW involves following a sequence of steps. In each step, we maintain certain configurations.

To perform a step, you need to select the step and click the EXECUTE button, as shown in [Figure 19.60](#). The list of steps varies based on the object attributes maintained in the first step.

Process Step	La
Maintain Object Attributes	
Maintain Source Structures	
Maintain Source Fields	
Maintain Structure Relations	
Maintain Field Mapping and Conversion Rules	
Maintain Fixed Values, Translations, User-Defined Routines	
Specify Files	
Assign Files	
Read Data	
Display Read Data	
Convert Data	
Display Converted Data	
Create Batch Input Session	
Run Batch Input Session	
Start Direct Input Program	
Start IDoc Generation	
Start IDoc Processing	
Create IDoc Overview	
Start IDoc Follow-Up	
Frame Program for Periodic Data Transfer	

Figure 19.60 LSMW Selection Steps

You can personalize the steps by clicking **USER MENU** in the application toolbar or by following the menu path **EXTRAS • USER MENU**.

The following list walks through the steps shown in [Figure 19.60](#):

1. Select the first step, **MAINTAIN OBJECT ATTRIBUTES**, and click the **EXECUTE** button. On the **LSM WORKBENCH: CHANGE OBJECT ATTRIBUTES** screen, click the **DISPLAY/CHANGE** button to maintain the attributes for the object.

Here, you can maintain the import method. If a **STANDARD BATCH/DIRECT INPUT** program is available for the object, you can use the direct input program in **LSMW** to migrate the data. You can also use any standard **BAPI** or **IDoc** available for the object. If none of these are available, you can perform a **BATCH INPUT RECORDING** to call the relevant transaction for the object and record the sequence of steps to post the data using a session.

For example, if you want to post materials, you can perform a recording using Transaction **MM01**, similar to using **Transaction Recorder**. Batch input recordings are specific to an object.

LSM Workbench: Change Object Attributes

Display <-> Change Documentation Display Interfaces

Attributes

Object: MATMAS [Material Master]

Owner: E9685324 Student

Data Transfer: Once-Only Periodic

File Names: System-Dependent

Object Type and Import Method

Standard Batch/Direct Input

Object: []

Method: []

Program Name: []

Program Type: []

Batch Input Recording

Recording: []

Business Object Method (BAPI)

Business Object: []

Method: []

Message Type: []

Basic Type: []

IDoc (Intermediate Document)

Message Type: []

Basic Type: []

Enhancement: []

Allow Structure Assignment for EDIDC40

Figure 19.61 Maintain Object Attributes

When a BAPI is used, it generates an IDoc automatically. We need to maintain the relevant message type and basic type if BUSINESS OBJECT METHOD (BAPI) or IDOC (INTERMEDIATE DOCUMENT) is selected as the input method. If you want to use other input options instead of recording, select the corresponding radio button and use the **F4** help for the field to select the available method for the object.

For this example, use BATCH INPUT RECORDING. Select the BATCH INPUT RECORDING radio button and click the RECORDINGS OVERVIEW icon on the right-hand side of the field.

This will take you to the RECORDINGS OVERVIEW screen, as shown in [Figure 19.62](#), where all the recordings for the object are displayed. Because you're creating a new project, this screen will be empty. Click the CREATE icon to perform a new recording.

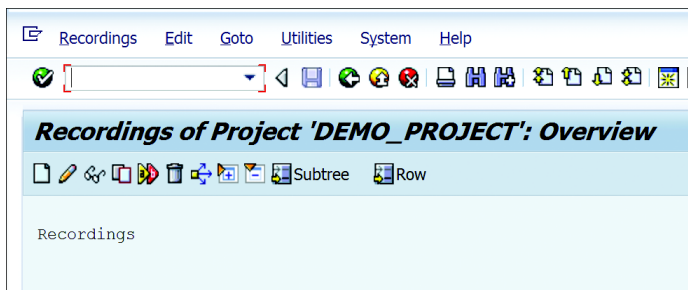


Figure 19.62 Recordings Overview

The system will present the CREATE RECORDING dialog box, in which you can input the name for the RECORDING and a short DESCRIPTION, as shown in [Figure 19.63](#). The recording name will be used for mapping the input file fields.

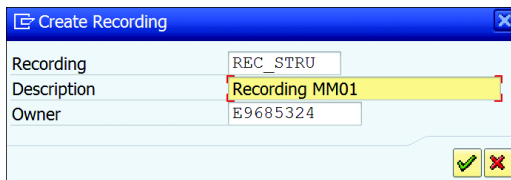


Figure 19.63 Create Recording

In the next window displayed (TRANSACTION CODE; see [Figure 19.64](#)), enter the TRANSACTION CODE and complete the recording.

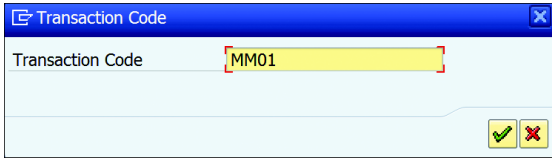


Figure 19.64 Transaction Code

Once the recording is completed, the system will capture all the information about the transaction screens, such as program name, screen number, field name, and field values, similar to recordings using Transaction SHDB. For this example, we recorded Transaction MM01 to create a material. The recording is displayed as shown in [Figure 19.65](#).

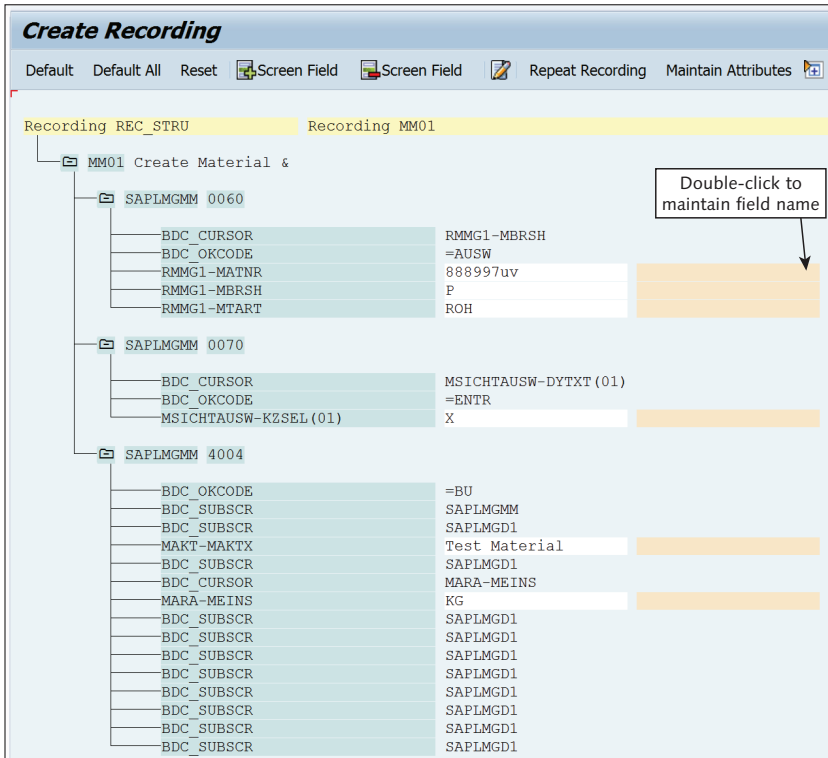
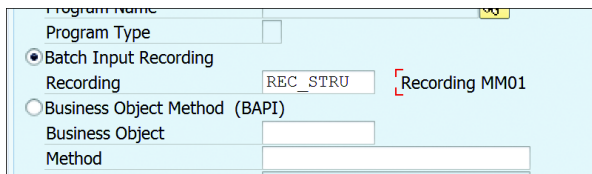


Figure 19.65 Recording of Transaction MM01

For all the fields where a value was entered during the recording, a colored field is displayed. Double-click such a field to maintain a field name to which the field of the input file will be mapped. The values entered during the recording will be maintained as the default values. If, during processing, no data is passed for the field from the input file, then the default value will be used. If you wish, you can delete the default value while assigning the field name.

You can add or delete fields using the respective button in the toolbar. Once the field names are maintained, SAVE the recording and click the BACK button twice to go back to the LSM WORKBENCH: CHANGE OBJECT ATTRIBUTES screen.

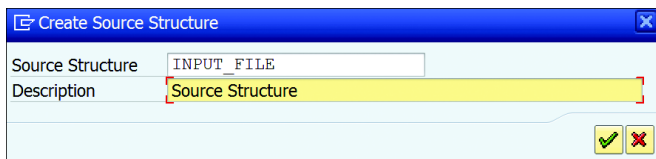
2. Maintain the recording name for the BATCH INPUT RECORDING field, as shown in [Figure 19.66](#). Save the attributes and click the BACK button to go back to the list of steps.



The screenshot shows a dialog box titled 'Maintain Recording'. It has several fields and radio buttons. The 'Batch Input Recording' radio button is selected. The 'Recording' field contains the text 'REC_STRU'. The 'Recording MM01' field contains the text 'Recording MM01'. Other fields include 'Program Name', 'Program Type', 'Business Object Method (BAPI)', 'Business Object', and 'Method', but they are empty or partially obscured.

Figure 19.66 Maintain Recording

3. Select the next step, MAINTAIN SOURCE STRUCTURES, and click the EXECUTE button.
4. Click the DISPLAY/CHANGE icon and select the CREATE icon in the toolbar. This will open a dialog box in which you can enter the input SOURCE STRUCTURE name, as shown in [Figure 19.67](#). SAVE and click the BACK button.



The screenshot shows a dialog box titled 'Create Source Structure'. It has two main fields: 'Source Structure' and 'Description'. The 'Source Structure' field contains the text 'INPUT_FILE'. The 'Description' field contains the text 'Source Structure'. There are also 'OK' and 'Cancel' buttons at the bottom right.

Figure 19.67 Maintain Source Structure

5. Select the MAINTAIN SOURCE FIELDS step and click the EXECUTE button. In this step, maintain the source fields that will be mapped to the recording fields created in the earlier step. The source fields will be set as per the sequential file that will be uploaded.

- Keep the cursor on the source structure name and click the TABLE MAINTENANCE icon to maintain the source fields. This will open the table maintenance screen in which you can maintain the source fields, as shown in [Figure 19.68](#). Click SAVE and then BACK to return to the list of steps.

Field Name	Type	L...	Field description
MATNR	C	18	Material
MBRSH	C	1	Industry sector
MTART	C	3	Material Type
MAKTX	C	18	Material Description
MEINS	C	3	Base Unit of Measure

Figure 19.68 Maintain Source Fields

- The next step is MAINTAIN STRUCTURE RELATIONS. In this step, notice that the system has automatically mapped the source structure with the recording structure (see [Figure 19.69](#)), because you've maintained only one source structure. If maintain multiple source structures, then you can map the relevant source structure with the recording structure in this step.

LSM Workbench: Display Structure Relationships

DEMO_PROJECT - DEMO_SUBPROJ - MATMAS Material Master

Structure Relations

```

  |
  |--- REC_STRU Recording MM01 <<<< INPUT_FILE Source Structure
  |      Select Target Structure REC_STRU .
  
```

Figure 19.69 Maintain Structure Relations

- In the next step, MAINTAIN FIELD MAPPING AND CONVERSION RULES, you can map the field of the source structure with the recording structure. If you selected other input methods in step 1, MAINTAIN OBJECT ATTRIBUTES, then the fields will be shown as per the method selected. For example, if you selected a

BAPI, then the fields will be displayed as per the BAPI interface, to which you can map the source fields.

Click the DISPLAY/CHANGE icon to maintain the field relationships. Keep the cursor on the field and click the SOURCE FIELD icon, as shown in [Figure 19.70](#).

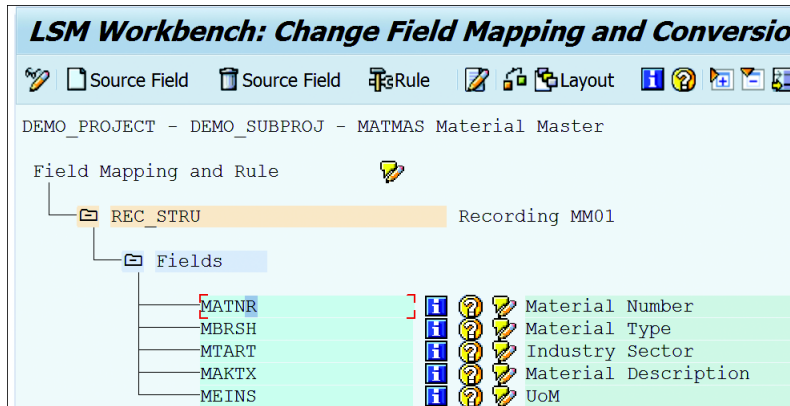


Figure 19.70 Maintain Field Mapping

Now, you'll see a list of fields from the source file. Double-click the source file field to map it to the recording structure. This will map the fields, as shown in [Figure 19.71](#). You can double-click the displayed rule to edit the mapping. For example, you can write an IF condition to check that the source field is not initial.

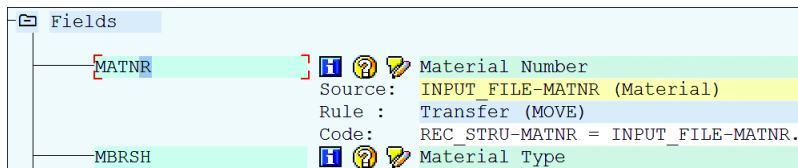


Figure 19.71 Field Rule

You can click the RULE icon in the toolbar to insert any rule. [Figure 19.72](#) shows the different rules that can be defined. For example, you can make the field initial, assign a constant, assign a fixed value, write ABAP code, and so on. You can learn more about each rule by visiting http://help.sap.com/saphelp_nw74/help-data/en/4d/adab9cbd316d57e1000000a42189e/frameset.htm.

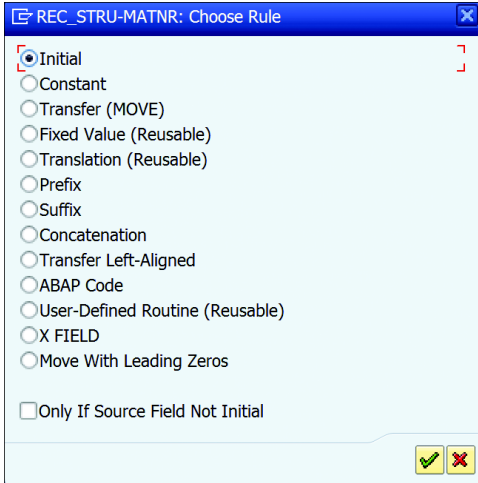


Figure 19.72 Rules

Once all the field mappings are maintained, SAVE and click BACK to return to the list of steps.

9. In the next step, MAINTAIN FIXED VALUES, TRANSLATIONS, USER-DEFINED ROUTINES, you can maintain reusable rules such as fixed values for the fields, translations, or user routines (subroutines; see [Figure 19.73](#)). This step allows you to maintain any conversion rules to convert the data to SAP format. Click the CREATE button in the toolbar to create the reusable rules that can be used during field mapping.

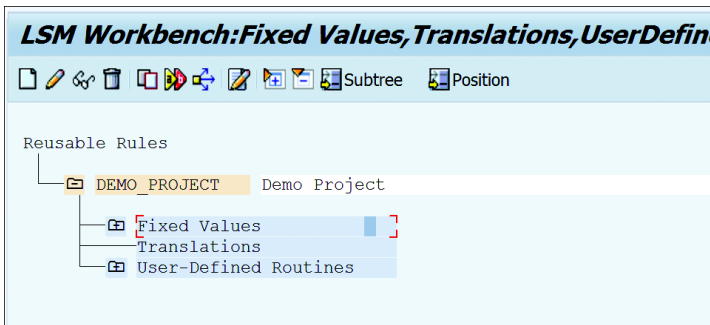


Figure 19.73 Reusable Rules

10. In the next step, SPECIFY FILES, maintain the path of the input field from which the input field will be read. The file can be imported either from a PC frontend or from a specific directory in the application server. The system also shows the name for the converted file. You can edit the name if you wish. To specify the file path, keep the cursor on the respective option (either PC FRONTEND or APPLICATION SERVER) and click the CREATE button on the toolbar (see [Figure 19.74](#)).

In the MAINTAIN PROPERTIES window, as shown in [Figure 19.75](#), you can maintain the input file properties. Here, you can maintain file properties such as delimiter, file type, code page, and so on, as shown in [Figure 19.75](#).

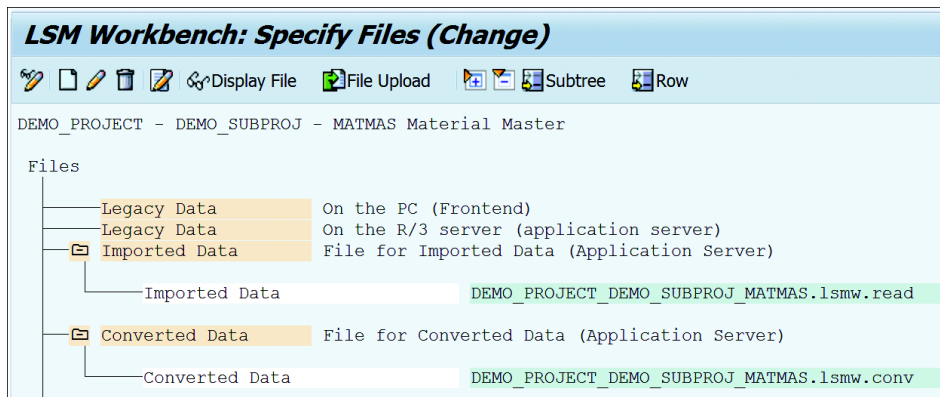


Figure 19.74 Specify Files

DELIMITER specifies how the columns in the input file are separated (e.g., by commas or tab spaces). FIELD NAME AT THE START OF THE FILE specifies that the input file contains the column headings in the first row. Selecting this checkbox will ignore the first row of the input file. The FILE ORDER MATCHES SOURCE STRUCTURE DEFINITION checkbox specifies that the fields in the input file match the source structure maintained in LSMW. The FILE TYPE options specify how the system identifies the end of a line. In the CODE PAGE section, you can select the applicable code page.

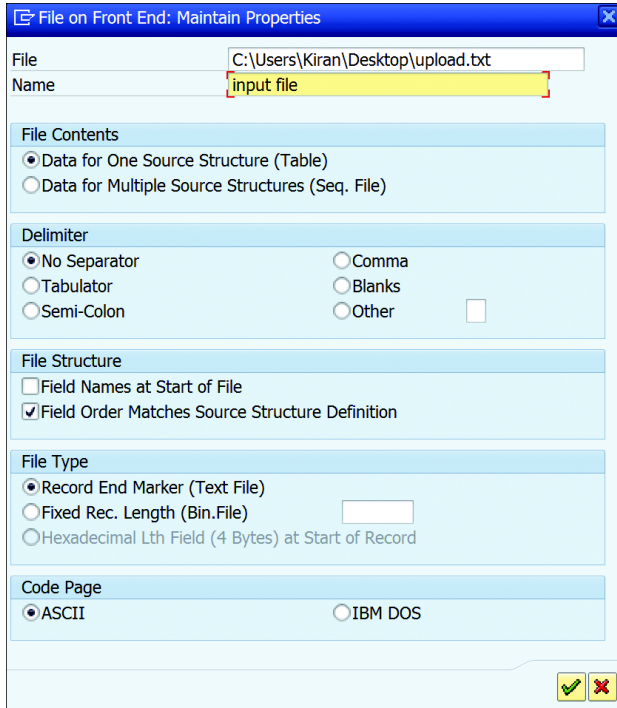


Figure 19.75 Import File

11. In the next step, ASSIGN FILES, you'll that the source structure is assigned to the input file path. If you've created multiple source fields, you can assign the files in this step. Otherwise, you can ignore this step.
12. Execute the next step, READ DATA, to read the input file. The system will upload the data from the input file.
13. In the next step, DISPLAY READ DATA, you can verify if the uploaded data is mapped correctly to the source structure.
14. The next step, CONVERT DATA, converts the uploaded data as per the field mappings and conversion rules maintained in the earlier step.
15. In the next step, DISPLAY CONVERTED DATA, you can verify that the data is converted correctly and mapped correctly to the recording structure, as shown in [Figure 19.76](#).

LSM Workbench: Converted Data

Structure REC_STRU

Fld Name	Fld Text	Field Length	FldValue
TABNAME	Table Name	30	REC_STRU
TCODE	Transaction Code	20	MM01
MATNR	Material Number	18	77uuuii
MBRSH	Material Type	1	P
MTART	Industry Sector	4	ROH
MAKTX	Material Description	40	Test
MEINS	UoM	3	KG

Figure 19.76 Converted Data

16. The next and final step is to create a batch input session (see [Figure 19.77](#)). Once the final step is executed, the batch input session is created, which can be processed using Transaction SM35. If the LSMW object attribute is maintained as direct input or BAPI or IDoc in the first step, then you'll see the respective final processing step instead of the batch input steps.

LSM Workbench: DEMO_PROJECT, DEMO_SUBPROJ

User Menu Numbering Off Double Click=Display Object Overview Action

Process Step
Maintain Object Attributes
Maintain Source Structures
Maintain Source Fields
Maintain Structure Relations
Maintain Field Mapping and Conversion Rules
Maintain Fixed Values, Translations, User-Defined Routines
Specify Files
Assign Files
Read Data
Display Read Data
Convert Data
Display Converted Data
Create Batch Input Session
Run Batch Input Session

Figure 19.77 Final Steps

As you can see, LSMW allows you to migrate data from an external system without having to write any code. The prerequisite for using LSMW is to extract the data from the external system into a sequential file.

If you're using a BAPI or IDoc as an input method, then you need to maintain the IDoc inbound processing settings (see [Figure 19.78](#)) via the menu path **SETTINGS • IDoc INBOUND PROCESSING** on the initial LSMW screen. Here, you maintain details such as **PARTNER TYPE**, **PARTNER NUMBER**, **PORT**, and so on.

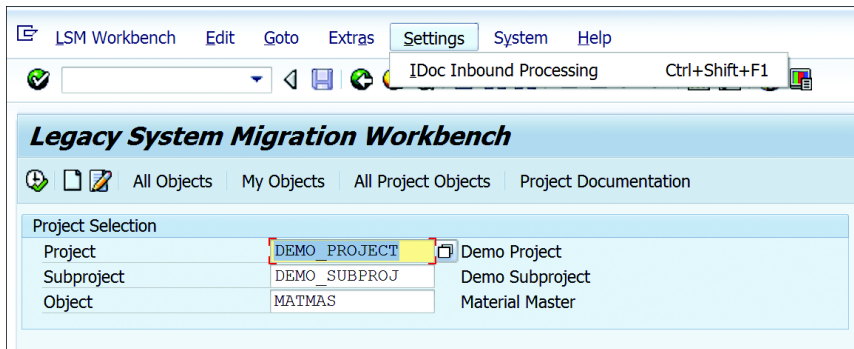


Figure 19.78 Maintain IDoc Inbound Processing

19.5 Web Services

A *web service* is a service offered by one system to another system over the Internet. With a web service, web technology such as HTTP is utilized for communication between systems. Using web services, data is exchanged in machine-readable file formats, such as XML and JSON. Web services provide object-oriented web interfaces for SAP systems; for example, a mobile application can access SAP data using a web service. A web service uses Internet standards such as the following:

- ▶ **Web Services Description Language (WSDL)**

Used to specify the interface of a web service. WSDL describes the interface of the actual web service using this XML-based language, consisting of the names of operations and the input and output messages. [Figure 19.79](#) shows a sample web service definition that defines the web service interface in an XML-based format. In [Figure 19.79](#), note that the interface definition consists of the fields of table SPFLI.


```

▼ <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://schemas.xmlsoap.org/wsdl/http/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:tns:targetNamespace="urn:sap-com:document:sap:soap:functions:mc-style">
  ▼ <wsdl:documentation>
    <sidl:sidl xmlns:sidl="http://www.sap.com/2007/03/sidl"/>
  </wsdl:documentation>
  ▼ <wsdl:types>
    ▼ <xsd:schema attributeFormDefault="qualified" targetNamespace="urn:sap-com:document:sap:rfc:functions">
      ▼ <xsd:simpleType name="char1">
        ▼ <xsd:restriction base="xsd:string">
          <xsd:maxLength value="1"/>
        </xsd:restriction>
      </xsd:simpleType>
      ▼ <xsd:simpleType name="char20">
        ▼ <xsd:restriction base="xsd:string">
          <xsd:maxLength value="20"/>
        </xsd:restriction>
      </xsd:simpleType>
      > <xsd:simpleType name="char3">...</xsd:simpleType>
      > <xsd:simpleType name="cInt3">...</xsd:simpleType>
      > <xsd:simpleType name="numeric4">...</xsd:simpleType>
      > <xsd:simpleType name="quantum9.4">...</xsd:simpleType>
      > <xsd:simpleType name="time">...</xsd:simpleType>
      > <xsd:simpleType name="unit3">...</xsd:simpleType>
    </xsd:schema>
    ▼ <xsd:schema xmlns:n0="urn:sap-com:document:sap:rfc:functions" attributeFormDefault="qualified" targetNamespace="urn:sap-com:document:sap:rfc:functions">
      <xsd:import namespace="urn:sap-com:document:sap:rfc:functions"/>
      ▼ <xsd:complexType name="Spfli">
        ▼ <xsd:sequence>
          <xsd:element name="Mandt" type="n0:cInt3"/>
          <xsd:element name="Carrid" type="n0:char3"/>
          <xsd:element name="Connid" type="n0:numeric4"/>
          <xsd:element name="Countryfr" type="n0:char3"/>
          <xsd:element name="Cityfrom" type="n0:char20"/>
          <xsd:element name="Airpfrom" type="n0:char3"/>
          <xsd:element name="Countryto" type="n0:char3"/>
          <xsd:element name="Cityto" type="n0:char20"/>
          <xsd:element name="Airpto" type="n0:char3"/>
          <xsd:element name="Fltime" type="xsd:int"/>
          <xsd:element name="Deptime" type="n0:time"/>
          <xsd:element name="Arntime" type="n0:time"/>
          <xsd:element name="Distance" type="n0:quantum9.4"/>
          <xsd:element name="Distid" type="n0:unit3"/>
          <xsd:element name="Fltype" type="n0:char1"/>
          <xsd:element name="Period" type="xsd:unsignedByte"/>
        </xsd:sequence>
      </xsd:complexType>
      ▼ <xsd:element name="ZdemoWs">
        ▼ <xsd:complexType>
          ▼ <xsd:sequence>
            <xsd:element name="Carrid" type="n0:char3"/>
            <xsd:element name="Connid" type="n0:numeric4"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      ▼ <xsd:element name="ZdemoWsResponse">
        ▼ <xsd:complexType>
          ▼ <xsd:sequence>

```

Figure 19.79 Sample Web Service Definition

► Simple Object Access Protocol (SOAP)

Used to transmit remote procedural calls. SOAP specifies a general format for XML messages that are application-independent. These messages can be exchanged between partners over different transport protocols.

► **Universal Description, Discovery, and Integration (UDDI)**

UDDI is used to locate web services. UDDI publicly enables web services, which allows them to be discoverable.

Using a web service, we can expose any remote-enabled function module, BAPI, IDoc, or XI Message interface. We can also consume web services in an ABAP program. We'll discuss how to create a web service in [Section 19.5.1](#) and look at consuming web services in an ABAP program in [Section 19.5.2](#).

Any business function that exists independently can be exposed as a web service using the web service framework. For example, if an external application developer wants to access the material data from an SAP system, he can consume a BAPI that has been exposed as a web service. Web services allow you to expose business functions to external applications using open standards.

SOAP requests are processed through the Internet Communication Framework (ICF). The ICF was introduced with the SAP Web Application Server in release 6.10, which allows ABAP programs direct access to the Internet. ICF is implemented in AS ABAP as an object-oriented layer that enables handling of HTTP requests within the ABAP environment. ICF enables you to work with the Internet Communication Manager (ICM) of the application server and is the technological foundation for all SAP technologies that use HTTP(s) for communication, such as Web Dynpro ABAP, web services, and SAPUI5. In web services, the HTTP protocol of the ICF is used for communication between web service requesters and providers. For web services to work, the SOAP runtime should be activated in Transaction SICF.

On the initial screen of Transaction SICF (see [Figure 19.80](#)), enter "service" in the HIERARCHY TYPE field and click EXECUTE to see the complete list of services. Navigate to the /SAP/BC/SRT node, and activate the node and all its subnodes, as shown in [Figure 19.81](#). You also need to activate the /SAP/BC/WEBDYNPRO/SAP/APPL_SOAP_MANAGEMENT node.

Maintain Services

Filter for Calling ICF Hierarchy

Hierarchy Type: SERVICE

Virtual Host:

Service Path:

Service Name:

Reference Service:

Description:

Language: English

Filter for Detail Information

Created By:

Created On: to

Last Changed By:

Changed On: to

Figure 19.80 Transaction SICF Initial Screen

Service/Host Edit Goto Client System Help

- Create Host
- Create Service F7
- Display/Change F8
- Activate Shift+F11
- Deactivate Shift+F12
- Alias References
- Delete Shift+F2
- Wizard: Create Service F9
- Exit Shift+F3

Apply Reset Fine-Tune

External Aliases System Monitor Active

Service Path

Ref.Service:

Virtuelle Hosts / Services	Documentation
soap	DIRECTORY FOR SOAP HTTP HANDLER
spi_gate	SPI: HTTP Interface for Tracking Records
srt	SOAP Runtime
esf	ESF Default Handler
esf_in	Generated for Web Service Administration (

Figure 19.81 Activating SOAP Runtime

19.5.1 Creating a Web Service

The web service framework provides a wizard that makes it easy to create a web service without writing any code. Any remote-enabled function module or BAPI can be easily exposed as a web service.

For our example, let's expose an RFC that provides the flight schedule data from table SPFLI as a web service. This function module has CARRID and CONNID as import parameters and the SPFLI structure as the export parameter.

Proceed with the following steps:

1. From the Function Builder menu, select UTILITIES • MORE UTILITIES • CREATE WEB SERVICE • FROM THE FUNCTION MODULE. You can also open the function module in Object Navigator and select CREATE • ENTERPRISE SERVICE from the context menu. This will start the CREATE WEB SERVICE wizard.
2. On the wizard screen, provide the web service name (SERVICE DEFINITION) and DESCRIPTION, then click the CONT. button, as shown in [Figure 19.82](#).

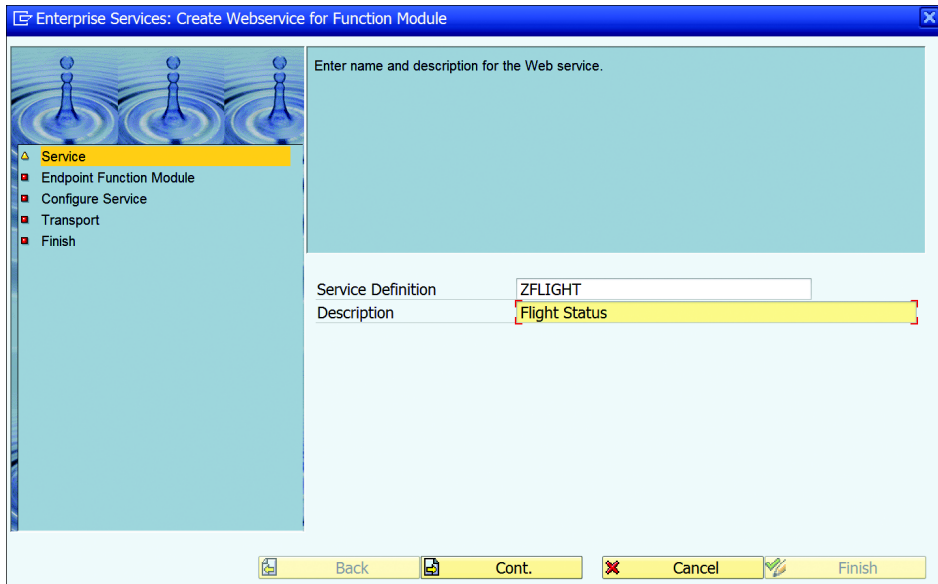


Figure 19.82 Web Service Wizard

- In the next step, enter the name of the function module that should be offered as a web service. Because you initiated the wizard from the function module, the function module name should be prefilled. Select the `MAP NAME` checkbox (see [Figure 19.83](#)) to remove underscores in the function name and capitalize the first letter in each word (i.e., set it in camel case).

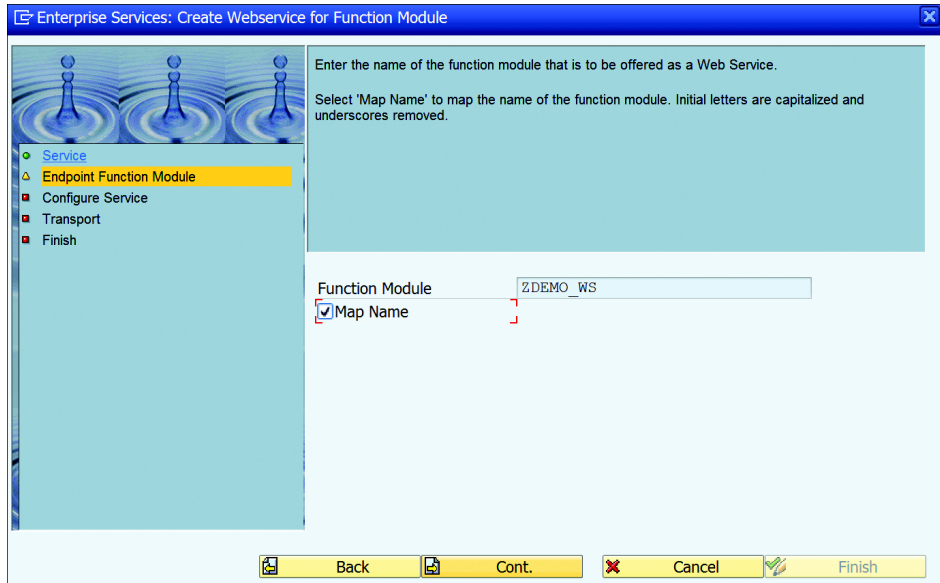


Figure 19.83 Endpoint Function Module

- In the next step, select the security profile for the web service. This specifies how the authentication should be performed for the web service.
- In the next step, select the package for the web service.
- In the final step, click `COMPLETE` to create the web service. The system will take you to the web service definition, as shown in [Figure 19.84](#). Activate the web service.
- The web service doesn't have a runtime release at this point, so it can't be used. We can create the runtime release via `Transaction SOAMANAGER`.
- Open `Transaction SOAMANAGER` to launch the `SOA MANAGEMENT` application in your web browser. Under the `SERVICE ADMINISTRATION` tab, click the `SIMPLIFIED WEB SERVICE CONFIGURATION` link, as shown in [Figure 19.85](#).

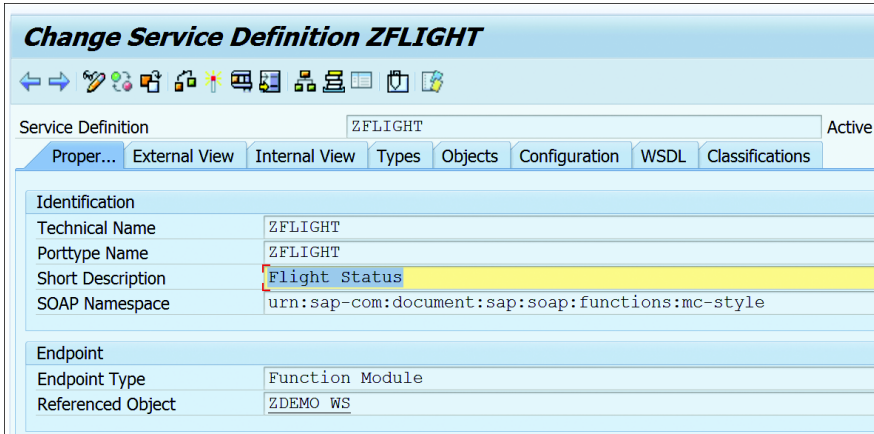


Figure 19.84 Web Service Definition

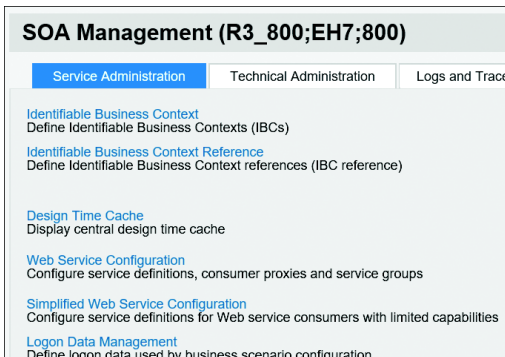


Figure 19.85 SOA Management

9. Search for the web service, select the USER NAME/PASSWORD (BASIC) checkbox, and click SAVE, as shown in [Figure 19.86](#).
10. Select the web service as shown in [Figure 19.86](#), and click the SHOW DETAILS button. This will display the WSDL URLs ([Figure 19.87](#)). Click on the URL to see the web service definition. Copy this URL; you'll use it to consume the web service in the next section. The URL listed after WITH WS POLICY contains the web service policy (such as security policy) advertised for the consumer.

Simplified Service Configuration (R3_800;EH7;800)

Configuration API Settings

Search Service Definitions

Search Pattern:

Configure Service Definitions

Save Show Details Show Changes Information: Standard Set Selected Unset Selected

Internal Name	User Name/Password (Basic)
ZFLIGHT	<input checked="" type="checkbox"/>

Figure 19.86 Simplified Service Configuration

Details of Provider Configuration

Details of the Service Definition

Internal Name: ZFLIGHT
 External Name: ZFLIGHT
 External Namespace: urn:sap-com:document:sap:soap:functions:mc-style
 Description: Flight Status

WSDL URLs

With WS Policy: [http://\[redacted\]/sap/bc/srt/wsd/fliv_10002A1111D1/bndg_url/sap/bc/srt/scs/sap/zflight?sap-client=800](http://[redacted]/sap/bc/srt/wsd/fliv_10002A1111D1/bndg_url/sap/bc/srt/scs/sap/zflight?sap-client=800)
 Without WS Policy: [http://\[redacted\]/sap/bc/srt/wsd/fliv_10002A1011D1/bndg_url/sap/bc/srt/scs/sap/zflight?sap-client=800](http://[redacted]/sap/bc/srt/wsd/fliv_10002A1011D1/bndg_url/sap/bc/srt/scs/sap/zflight?sap-client=800)

Configuration Details

Configuration Name: 94DE80C1CC561EE68C9BD18F7F2D5DF1
 Access URL: [http://\[redacted\]/sap/bc/srt/scs/sap/zflight?sap-client=800](http://[redacted]/sap/bc/srt/scs/sap/zflight?sap-client=800)
 Last Change User: E9685324
 Last Change Date: 13.06.2016
 Last Change Time: 02:15:13

OK

Figure 19.87 WSDL URLs

11. You can execute the web service definition in Object Navigator to test the web service (see [Figure 19.88](#)).

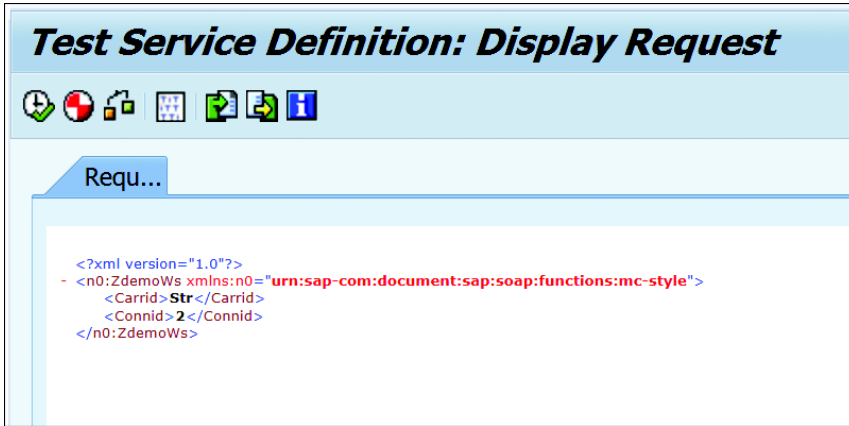


Figure 19.88 Test Web Service Definition

19.5.2 Consuming Web Services

A web service is consumed using the web service client proxy. The *client proxy* takes care of creating valid SOAP messages and evaluates received responses. It also acts as a transfer program and takes care of connecting to the server of the required web service.

In the following subsections, we'll walk through the process of consuming a web service.

Creating a Web Service Consumer

The following steps walk you through creating a web service consumer:

1. In Object Navigator, select CREATE • ENTERPRISE SERVICE from the context menu of the package.
2. This will open the ENTERPRISE SERVICES wizard. Select the SERVICE CONSUMER radio button, as shown in [Figure 19.89](#), and click CONT.
3. Select EXTERNAL WSDL/SCHEMA, as shown in [Figure 19.90](#).
4. Select URL as the WSDL/SCHEMA SOURCE, as shown in [Figure 19.91](#). This allows you to use the URL of the web service as the source.

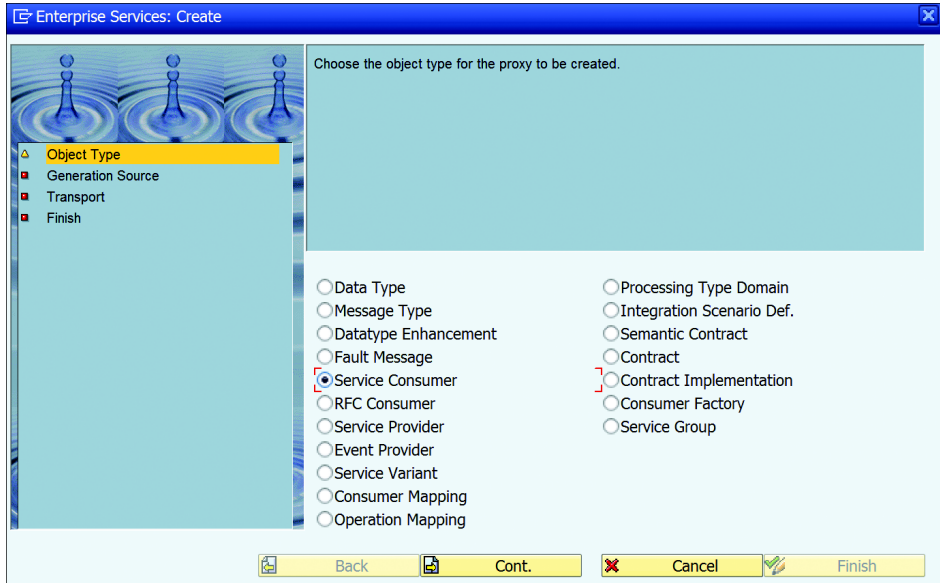


Figure 19.89 Create Consumer

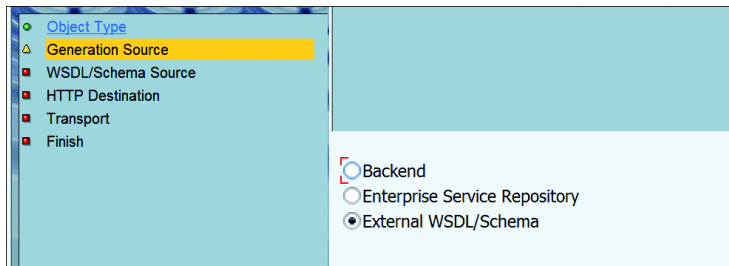


Figure 19.90 Select Generation Source

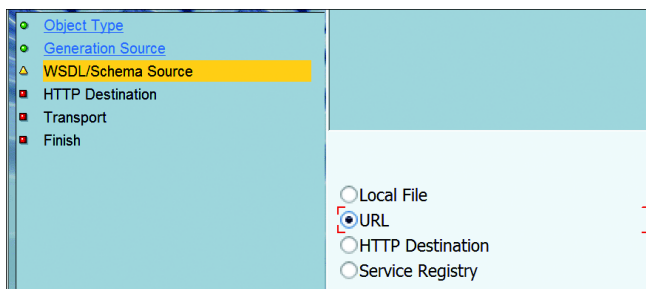


Figure 19.91 Schema Source

5. Provide the URL for the web service. For this example, use the URL that was generated for the service definition in the previous steps.
6. The system will prompt you to enter a USER NAME and PASSWORD to connect to the URL.
7. Enter the PACKAGE details (see [Figure 19.92](#)).

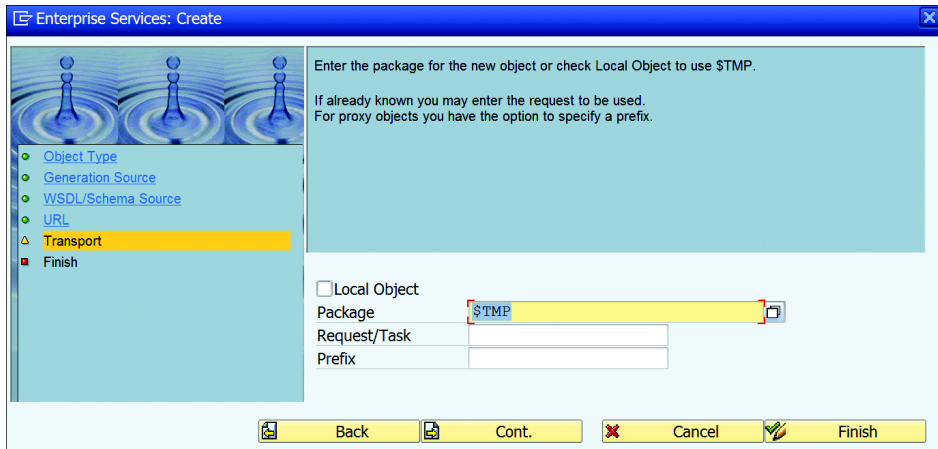


Figure 19.92 Maintain Package

8. In the final step, verify the details and click COMPLETE to complete the wizard. Activate the service proxy by clicking the ACTIVATE icon.

Maintaining Port Information

Once the web service proxy is created, we need to maintain port information in Transaction SOAMANAGER. Proceed with the following steps:

1. In Transaction SOAMANAGER, click on the WEB SERVICE CONFIGURATION link (see [Figure 19.85](#)) and search for the consumer proxy you just created (see [Figure 19.93](#)). If you provided a prefix to be used while creating the proxy, the same will be used as a prefix.
2. Select the consumer proxy you created and click the CREATE • MANUAL CONFIGURATION (see [Figure 19.94](#)) option to create a logical port name (see [Figure 19.95](#)). A *logical port* is used for configuration of the runtime services of the client proxy.

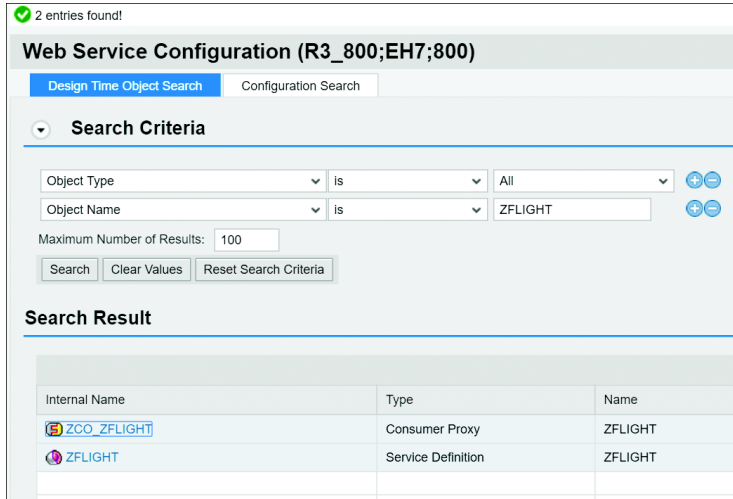


Figure 19.93 Search Consumer Proxies

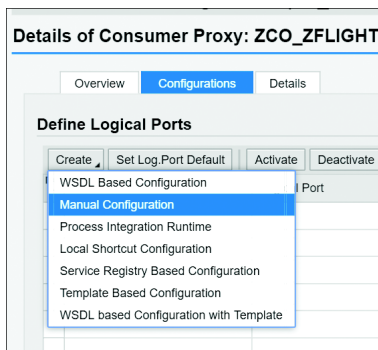


Figure 19.94 Create Manual Configuration

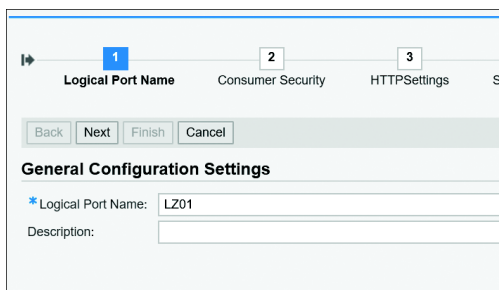


Figure 19.95 Maintain Logical Port

3. Provide the authentication details that should be used by the consumer (see [Figure 19.96](#)).

Figure 19.96 Consumer Security

4. In the next step, maintain HTTP settings: Provide the web service producer URL in the URL ACCESS PATH field (see [Figure 19.97](#)). Leave the other fields set to their default values and click FINISH.
5. Once completed, you can execute the service consumer in Object Navigator for testing.

Figure 19.97 HTTP Settings

Creating an ABAP Program to Consume a Web Service

Finally, we need to create an ABAP program to consume the web service. In Object Navigator, you can drag and drop the web service consumer (see [Figure 19.98](#)) into ABAP Editor to generate the code template.

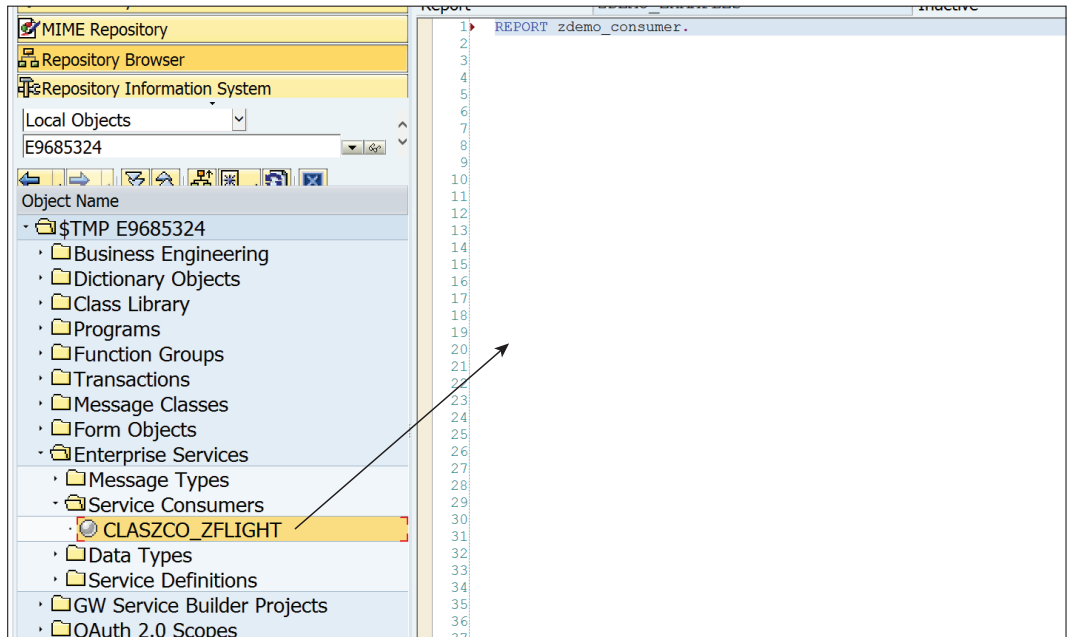


Figure 19.98 Generate Sample Code

[Listing 19.4](#) shows sample code to consume a web service in an ABAP program.

```
REPORT ZDEMO_WS.
DATA: r_ws TYPE REF TO zwsco_zflight,
      output TYPE ZWSZDEMO_WS_RESPONSE,
      input TYPE ZWSZDEMO_WS.
input-carrid = 'AA'.
input-connid = '0017'.

TRY.
  CREATE OBJECT r_ws
  EXPORTING
    logical_port_name = 'LZ01'.

  CALL METHOD r_ws->zdemo_ws
  EXPORTING
```

```

    input = input
IMPORTING
    output = output.

CATCH cx_ai_system_fault .
ENDTRY.

```

Listing 19.4 Consuming Web Service

In [Listing 19.4](#), a `R_WS` reference object is defined, referring to the service consumer. The function module is added as a method with two parameter, `input` and `output`. The `input` parameter holds the importing fields of the function module exposed as a web service. The `output` parameter holds the exporting parameters of the function module. The web service proxy object is instantiated using the logical port name.

[Figure 19.99](#) shows the data in the `output` structure after the method call.

Exp.	Component	Val...	Val.	Ch...	Technical Type
	CONTROLLER	[0x2 (62)]	Standard Table		Standard Table[0x2
	SPFLI	Structure: deep	deep		Structure: deep(184
	CONTROLLER	[0x2 (62)]	Standard Table		Standard Table[0x2
	MANDT	800			C(3)
	CARRID	AA			C(3)
	CONNID	0017			N(4)
	COUNTRYFR	US			C(3)
	CITYFROM	NEW YORK			C(20)
	AIRPFROM	JFK			C(3)
	COUNTRYTO	US			C(3)
	CITYTO	SAN FRANCISCO			C(20)
	AIRPTO	SFO			C(3)
	FLTIME	361			I(4)

Figure 19.99 Data in Output Structure

19.6 OData Services

Open Data Protocol (OData) is an open protocol that allows you to create and consume RESTful APIs in a simple and standard way. *RESTful APIs* follow Representational State Transfer (REST) standards-based architecture for data communication. In REST architecture, the REST server provides access to a resource, and the REST client can access the resource to present it in a UI. OData is released under Microsoft's Open Specification Promise (OSP) and allows for querying and updating data. Using OData, we can expose business data to any application running on any device, such as smartphones or HTML-based clients. OData is an OASIS standard and is endorsed by many leading software vendors.

OData has the potential to become the common protocol for bringing business data to any Internet-ready device. With OData, information can be accessed from a variety of sources, including relational databases, file systems, content management systems, traditional websites, and more. In the future, OData could be the standard for sharing business data across applications and could be as popular as SQL is for relational databases.

OData software development kits (SDKs) are available for many client technologies, making consuming OData easy in any kind of application. Many client technologies also have dedicated developer tools and frameworks that allow for jumpstarting application development by consuming available OData services.

SAP itself has invested in SAP Gateway technology to improve the reach of SAP business applications. *SAP Gateway* is a RESTful interface based on OData and Atom formats for the ABAP technology platform. Atom is an XML language used to syndicate content for web feeds (much like the RSS syndicate for the web). It provides a centralized interface to the SAP world that is open and standards-based. With SAP Gateway and OData, you can easily build applications—for example, mobile apps—that can work with business data in the SAP system. The OData protocol defines conventions for performing CRUDQ (create, read, update, delete, query) operations, which the OData service can support.

Figure 19.100 provides an architectural overview of the SAP Gateway interface.

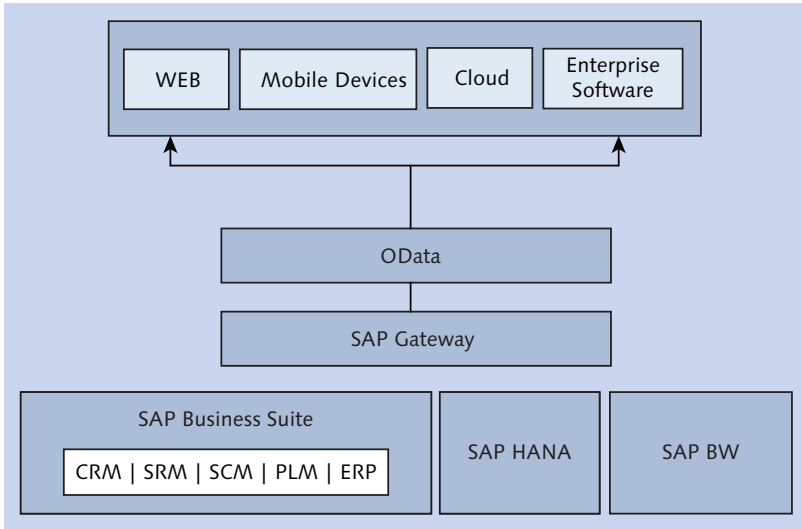


Figure 19.100 SAP Gateway as a Common Interface

Further Information

An in-depth discussion of SAP Gateway and OData commands a book of its own and is beyond the scope of this chapter. For more information on SAP Gateway and OData, please see *SAP Gateway and OData* (second edition, SAP PRESS, 2016).

In this section, we'll describe how an OData service is generated using existing business objects and interfaces. Using SAP Gateway Service Builder, we can generate OData services from existing RFC modules, BAPIs, search helps, and CDS views. For search helps and CDS views, only the `READ` and `QUERY` methods are supported.

Building an OData service involves three steps:

1. Data model definition
2. Service implementation
3. Service maintenance

SAP Gateway Service Builder can be accessed via Transaction `SEGW`. It covers various options, as shown in [Figure 19.101](#).

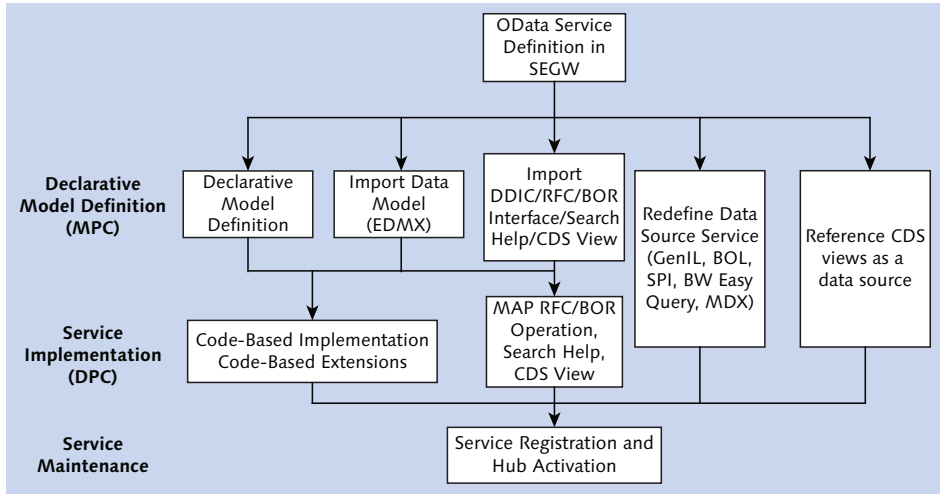


Figure 19.101 Service Options in Transaction SEGW

In the following subsections, we'll walk through the OData service creation process.

19.6.1 Data Model Definition

The following steps illustrate creating a service in Transaction SEGW. For this example, we'll import the `BAPI_EPM_SO_GET_LIST` BAPI to expose sales order data. This BAPI is part of the SAP Enterprise Procurement Model (EPM). SAP EPM is a test application that serves as a proxy. Proceed as follows:

1. From the initial screen of Transaction SEGW, create a new project by clicking the CREATE icon (first icon to the left), as shown in [Figure 19.102](#).



Figure 19.102 Create Project

2. In the CREATE PROJECT dialog box, provide a name for the PROJECT and a short DESCRIPTION, as shown in [Figure 19.103](#).

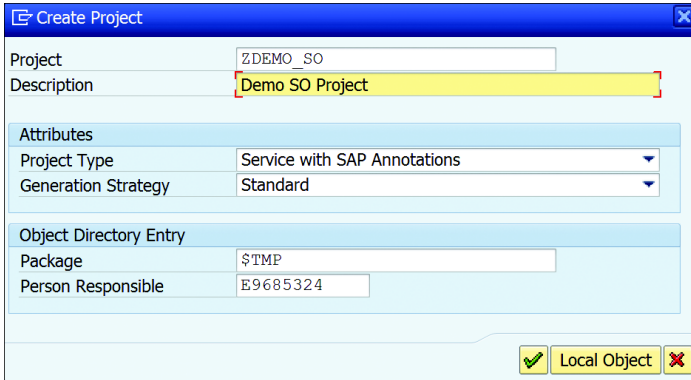


Figure 19.103 Create Project Window

3. Right-click the DATA MODEL folder and select the menu path **IMPORT • RFC/BOR INTERFACE** (see [Figure 19.104](#)) from the context menu.

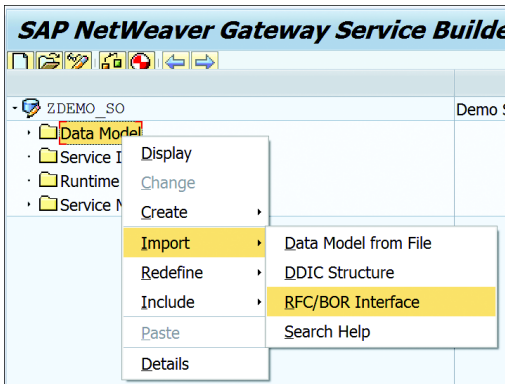


Figure 19.104 Import RFC/BOR Interface

4. In the **CREATE ENTITY** window, provide a name for the entity (**ENTITY TYPE NAME**), as shown in [Figure 19.105](#). You'll be creating two entities, one for the header and one for line items.
5. Under **TARGET SYSTEM**, you can provide a **RFC DESTINATION** to add the RFC from a **REMOTE** system. For this example, however, use the RFC in a **LOCAL** system.
6. Under **DATA SOURCE ATTRIBUTES**, select **REMOTE FUNCTION CALL** and provide the name of the RFC. For this example, enter "BAPI_EPM_SO_GET_LIST". Click **NEXT**.

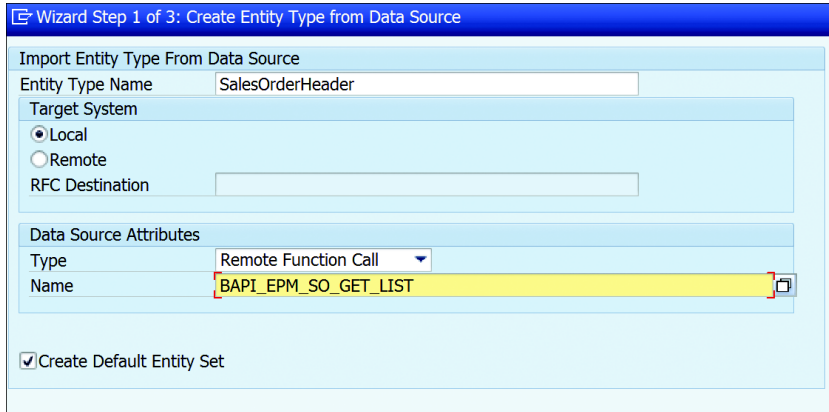


Figure 19.105 Create Entity

7. The wizard will present the interface of the RFC, from which you can select the relevant fields that should be added to the service. Expand the SOHEADER-DATA[] node and select the relevant sales order header fields, as shown in [Figure 19.106](#). For this example, select SO_ID, CREATED_BY, CURRENCY_CODE, GROSS_AMOUNT, and NET_AMOUNT.

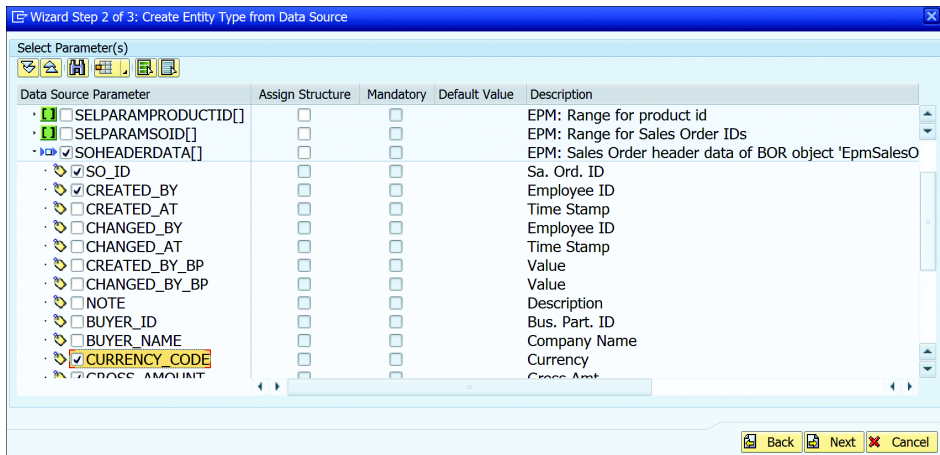


Figure 19.106 Add Fields from Interface

8. In the next step, mark SO_ID as the key field (select the Is KEY checkbox; see [Figure 19.107](#)) and click FINISH.

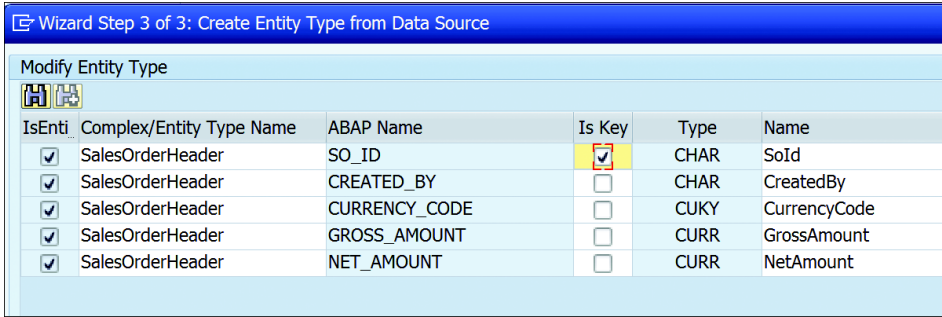


Figure 19.107 Mark Key Fields

- Repeat the steps to add a SALESORDERITEM entity, as shown in Figure 19.108. Name the entity SALESORDERITEM and select fields from the SOITEMDATA[] structure, as shown in Figure 19.109.

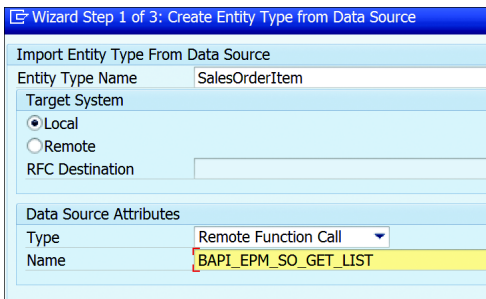


Figure 19.108 Add SalesOrderItem Entity

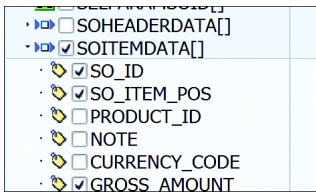


Figure 19.109 Item Fields

- Mark SO_ID and SO_ITEM_POS as key fields.
- Expand the ENTITY SETS folder to see the entity sets you just created. For each entity, you can select the CRUD operations that can be performed, as shown in Figure 19.110

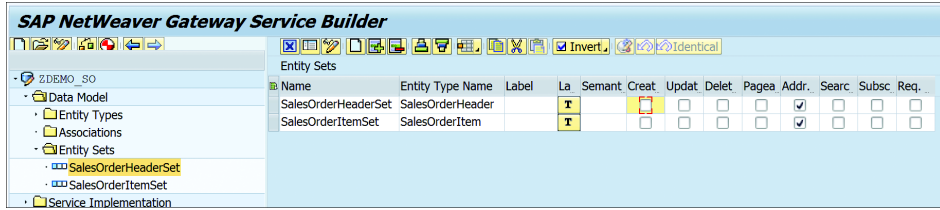


Figure 19.110 Entity Sets Folder Contents

12. The basic definition of the model is completed; now, you need to generate the runtime object. Click the GENERATE RUNTIME icon, as shown in Figure 19.111, to generate the runtime object.

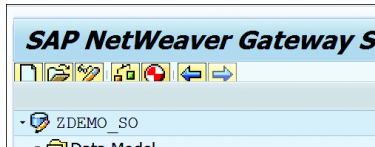


Figure 19.111 Generate Runtime Object

13. In the next window, accept the default values and save the object. At this point, you can verify that the generation was successful, as shown in Figure 19.112.

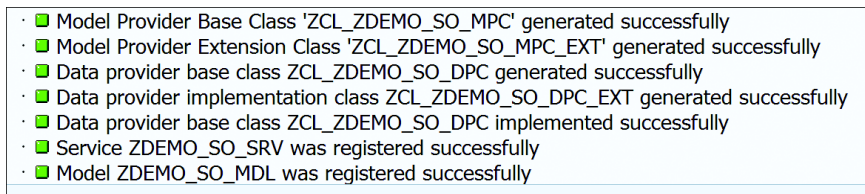


Figure 19.112 Generation Message

19.6.2 Service Maintenance

After defining the service, the service should be activated and registered with the SAP Gateway system. The activation and registration of the service is required for SAP Gateway to consume a service using an OData client. To activate and register the service, use Transaction /IWFND/MAINT_SERVICE. However, this transaction is integrated with Service Builder and thus can be called from within Service Builder.

The following steps walk you through activating and registering the service with SAP Gateway using Service Builder:

1. To register and activate the service, first double-click the SERVICE MAINTENANCE folder and click the REGISTER button, as shown in [Figure 19.113](#).

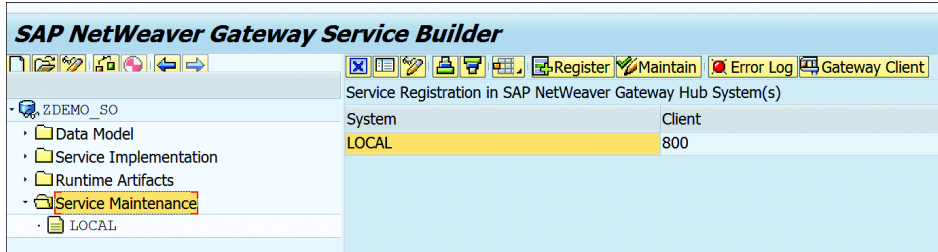


Figure 19.113 Service Registration

2. In the SELECT SYSTEM ALIAS window that opens, you can use **F4** help to select the system that acts as a hub to the backend. You can use the RFC destination that points to the SAP business system. Based on the IMG configuration and system installation, you can use LOCAL as the SYSTEM ALIAS.
3. In the ADD SERVICE window, as shown in [Figure 19.114](#), make a note of the EXTERNAL SERVICE name. This name is used to access the service.

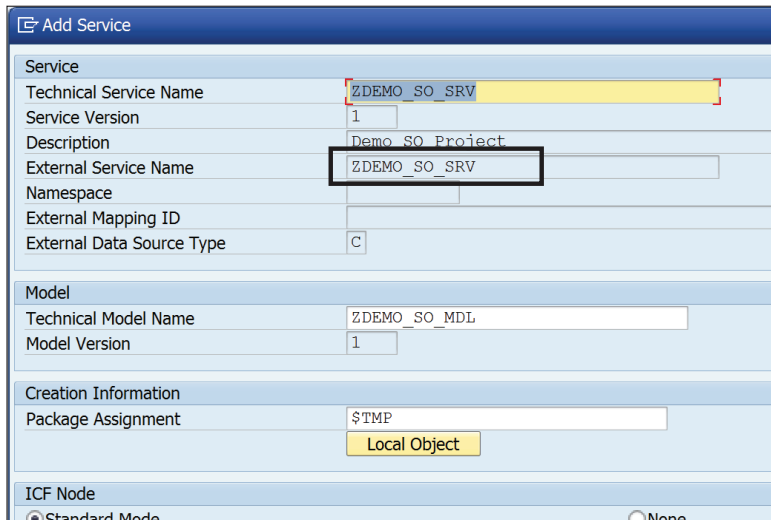


Figure 19.114 Add Service Window

4. At this point, you can verify if the service is registered successfully under the REGISTRATION STATUS column, as shown in [Figure 19.115](#).

Service Registration in SAP NetWeaver Gateway Hub System(s)			
System	Client	RFC Destination	Registration Status
LOCAL	800	NONE	

Figure 19.115 Verify Registration

We've successfully maintained the basic model data. We can access the metadata of the service using Transaction /IWFND/GW_CLIENT. Use the `/sap/opu/odata/sap/ZDEMO_SO_SRV/$metadata` URI to check the metadata.

We've successfully created a project in Service Builder with two entities and registered and activated the OData service. The OData service isn't yet mapped to the data provider, however. The next step is to map the data provider and see the service in action.

19.6.3 Service Implementation

When implementing the OData service, we can map the data provider to perform various CRUD operations. The OData service also supports a query operation to query the data. To map the data provider, proceed as follows:

1. First, expand the SERVICE IMPLEMENTATION folder and then SALESORDERHEADERSET. Right-click GETENTITYSET (QUERY) and select MAP TO DATA SOURCE, as shown in [Figure 19.116](#).

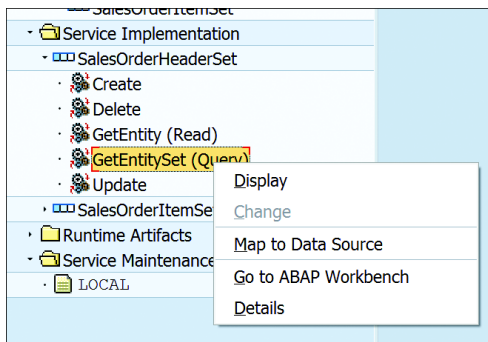


Figure 19.116 Map Data Source

2. Enter the RFC for the DATA SOURCE, as shown in [Figure 19.117](#).

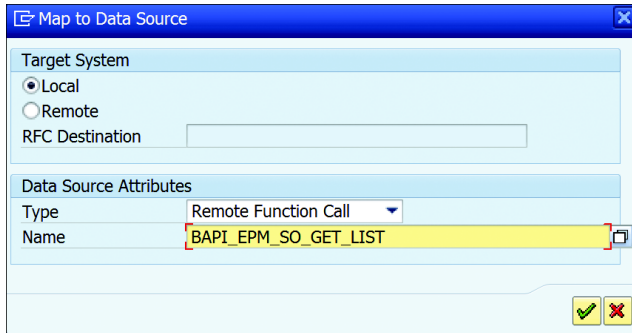


Figure 19.117 Map to Data Source

3. The data source parameters will be displayed on the right, as shown in [Figure 19.118](#). You can click the PROPOSE MAPPING button to have the system automatically propose the mapping, or manually drag and drop the fields from the right window to the DATA SOURCE PARAMETER area on the left.

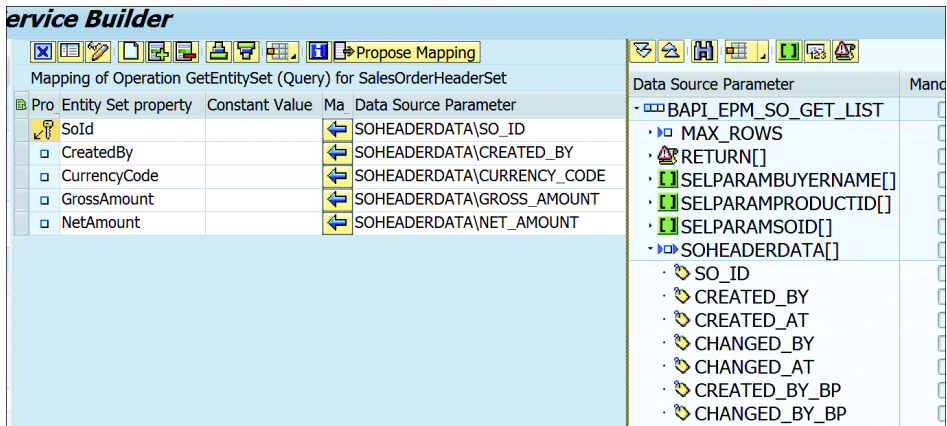


Figure 19.118 Mapping Data Source Parameters

4. Repeat the steps to map the sales order item entity. Once the mapping is complete, click the GENERATE icon to generate the runtime artifacts.

Once the OData service is implemented, we can test it by entering the URI `/sap/opu/odata/sap/ZDEMO_SO_SRV/SalesOrderHeaderSet` in Transaction `/IWFND/GW_CLIENT`. Note that there is no filter value, so the OData service will pick all the doc-

uments from the system, as shown in [Figure 19.119](#). This may impact performance, so we'll map a data source for the READ operation so that we can get a specific sales order's details.

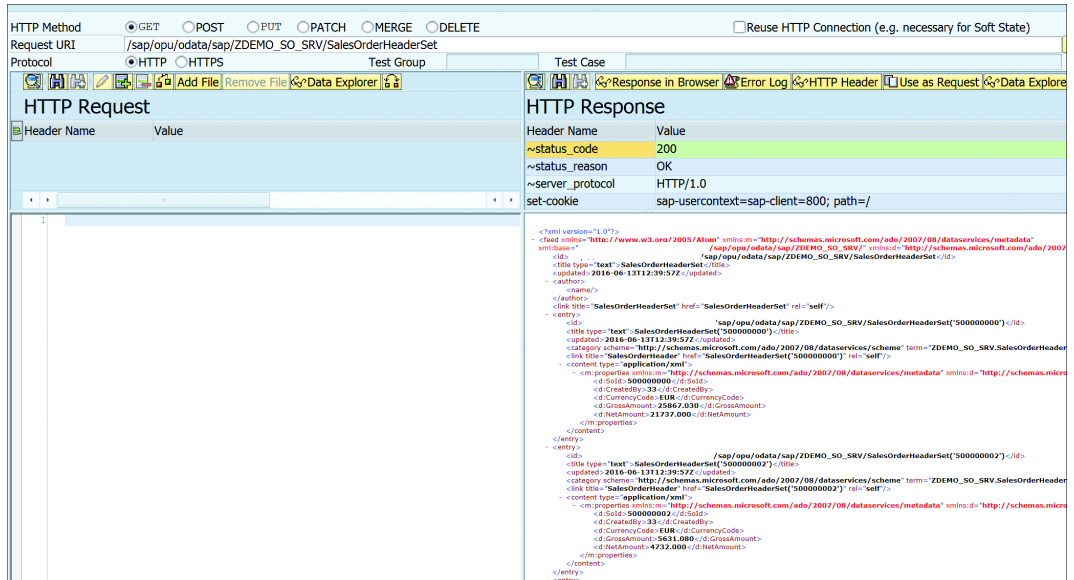


Figure 19.119 Query Operation

19.6.4 READ

The READ method is used to read a specific record from the entity for a given filter. To put it in a SQL perspective, the READ method behaves similarly to using the SELECT SINGLE * Open SQL statement, in which one specific record is read. The QUERY method behaves more like the SELECT INTO TABLE Open SQL statement, in which multiple records are fetched. The CREATE, UPDATE, and DELETE methods behave like the INSERT, UPDATE, and DELETE Open SQL statements.

Proceed as follows to define the READ method:

1. Expand the SERVICE IMPLEMENTATION folder and then SALESORDERHEADERSET. Right-click GETENTITY (READ) and select MAP TO DATA SOURCE.
2. For the READ operation, implement the BAPI_EPM_SO_GET_DETAIL RFC, as shown in [Figure 19.120](#).

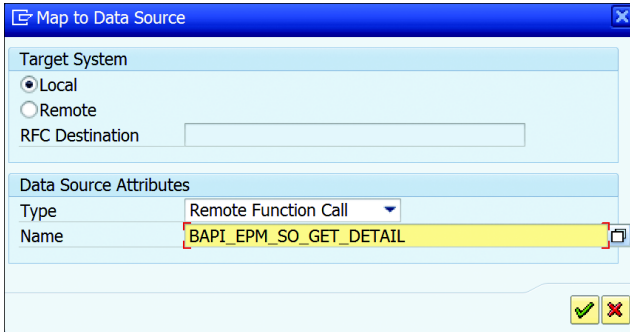


Figure 19.120 Mapping Data Source

- Click the PROPOSE MAPPING button to create a proposal automatically, as shown in Figure 19.121. However, this RFC has an `import` parameter that should be supplied with the document number. To map the input field, next you'll create a new field in the entity manually and then map it to the `import` parameter of the function module.

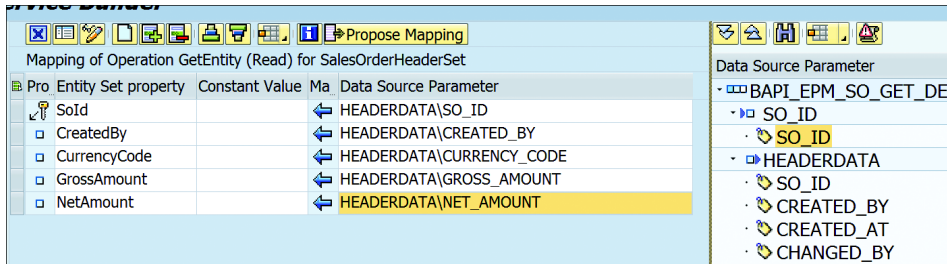


Figure 19.121 Create Proposal

- Click the CREATE icon to add a new row. Use the `[F4]` help in the row to select the field from the list, as shown in Figure 19.122.
- Once the field is added, drag and drop the `SO_ID` key field from the function module to the DATA SOURCE PARAMETER, as shown in Figure 19.123. The direction for this field is automatically set to INPUT; all the other fields are set to OUTPUT.
- In Transaction `/IWFND/GW_CLIENT`, run the `/sap/opu/odata/sap/ZDEMO_SO_SRV/SalesOrderHeaderSet('500000002')` URI by passing a document number. This will call the `READ` method to fetch the details of the specific document, as shown in Figure 19.124.

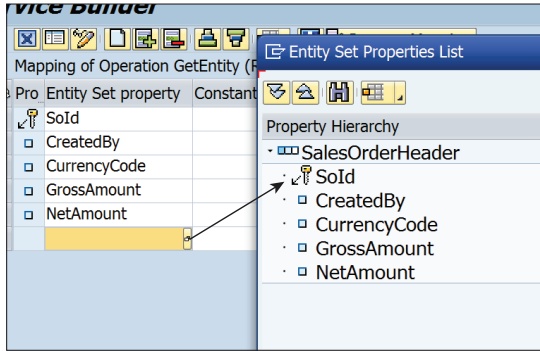


Figure 19.122 Create New Field

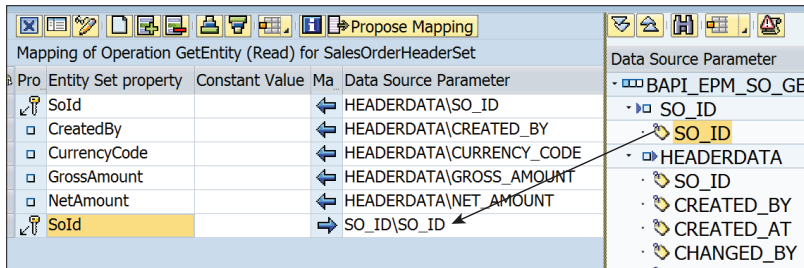


Figure 19.123 Map Import Parameter

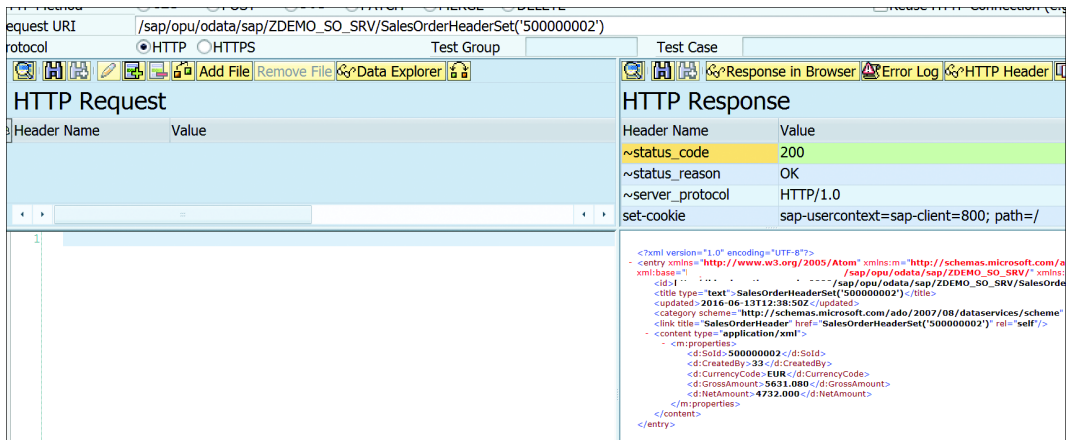


Figure 19.124 Reading Specific Record

We've successfully modeled the OData service and implemented it using the `QUERY` and `READ` methods. You can implement the `CREATE` and `DELETE` methods in a similar way by right-clicking the respective entity and mapping the source (as explained in the preceding steps). For example, for the `CREATE` method, you map the `Create()` BAPI, and for the `DELETE` method you map the `Delete()` BAPI of the business object.

As you can see, SAP Gateway Service Builder makes it easy to expose business services without having to write any code. These services can be consumed by external applications to perform CRUDQ operations. When working with OData services, the most time goes into designing the model and its implementation. Once the service is implemented correctly, consuming the service is fairly easy.

19.7 XSL Transformations

Extensible Stylesheet Language (XSL) is a stylesheet language for XML documents. Extensible Stylesheet Language Transformations (XSLT) allows you to transform XML documents into other formats. XSLT is a declarative programming language used for the structural transformation of XML documents and is standardized by W3C (World Wide Web Consortium).

XML documents consist of tree structures, and XSLT programs contain the rules for transformation in templates. These rules are applied to the source tree for transformation. For navigation and selection of node sets from the source tree, XSLT uses XML Path Language (XPath), which is a query language.

XSLT allows you to deserialize XML documents into ABAP data objects and vice versa. The XSLT processor, called the *SAP XSLT Processor*, is implemented in the kernel of SAP NetWeaver AS ABAP. The SAP XSLT Processor executes the XSLT programs, which exist in the repository just like any other ABAP programs. You can create these programs in Object Navigator via the context menu path `CREATE • OTHER • TRANSFORMATION`.

The `CALL TRANSFORMATION` statement is used to call the transformation. The syntax for using the statement is as follows:

```
CALL TRANSFORMATION trans SOURCE [XML]..
                        RESULT [XML]..
```

In the statement, `trans` is an XSLT or a Simple Transformation (ST) program. If XML is provided for both the source and result, the XML document will be transformed into a different XML document using the specified transformation.

A parameter list, structure, or internal table can be used to serialize or deserialize data into ABAP data objects. Because XSLT expects data in XML format for transformation and can't interpret ABAP data directly, the ABAP data is implicitly transformed into ABAP Serialization XML (asXML) before it's processed by XSLT.

19.7.1 Serialization

During serialization, ABAP data is first converted to asXML using a predefined transformation ID, the result of which is used as the source for XSLT. If ID itself is used for the ID in the transformation, then the asXML itself is output without further transformation.

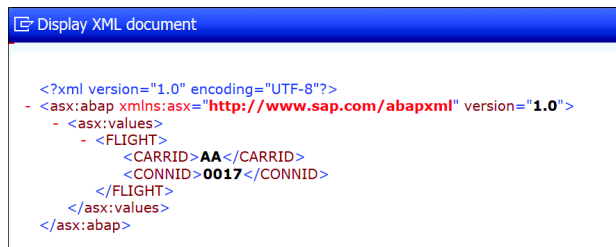
Listing 19.5 shows sample code to transform the data in an ABAP structure to XML. The `DISPLAY_XML_STRING` function module displays the XML output, as shown in Listing 19.5.

```
DATA : BEGIN OF sflight,
        carrid TYPE s_carr_id VALUE 'AA',
        connid TYPE s_conn_id VALUE '0017',
      END OF sflight.
DATA xmlstr TYPE xstring.

CALL TRANSFORMATION ID
SOURCE FLIGHT = sflight
RESULT XML xmlstr.

CALL FUNCTION 'DISPLAY_XML_STRING'
EXPORTING xml_string = xmlstr.
```

Listing 19.5 XML Transformation



```
<?xml version="1.0" encoding="UTF-8"?>
- <asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  - <asx:values>
    - <FLIGHT>
      <CARRID>AA</CARRID>
      <CONNID>0017</CONNID>
    </FLIGHT>
  </asx:values>
</asx:abap>
```

Figure 19.125 XML Output

19.7.2 Deserialization

During deserialization, the result of the XML transformation is deserialized directly into the ABAP data. The following is an example of deserialization:

```
CALL TRANSFORMATION ID
SOURCE XML xmlstr
RESULT FLIGHT = sflight.
```

Similarly, you can use individual fields or internal tables for serialization and deserialization. [Listing 19.6](#) shows sample code for serializing internal table data. The result of the code in [Listing 19.6](#) is shown in [Figure 19.126](#).

```
DATA: it_sflight TYPE STANDARD TABLE OF sflight,
      xmlstr TYPE xstring.
SELECT * FROM sflight INTO TABLE it_sflight WHERE carrid EQ 'AA'.
CALL TRANSFORMATION ID
SOURCE TABLE = it_sflight
RESULT XML xmlstr.
CALL FUNCTION 'DISPLAY_XML_STRING'
EXPORTING xml_string = xmlstr.
```

Listing 19.6 Serializing Internal Table Data



Figure 19.126 Result of Code in Listing 19.6

You can serialize objects as well, if the class implements the `IF_SERIALIZABLE_OBJECT` interface.

19.8 XML and JSON Data Representation

JavaScript Object Notation (JSON) is a lightweight data-interchange format. Similar to XML, JSON is programming language-independent and easy to understand. JSON text can be read and used as a data format in any programming language. JSON is self-describing and is an easy-to-use alternative to XML.

Data is represented differently in JSON compared to XML. For example, [Listing 19.7](#) shows an array with three records represented in JSON.

```
{ "books": [
  { "title": "OData", "isbn": "78-1-59229-907-2" },
  { "title": "ABAP", "isbn": "978-1-59229-859-4" },
  { "title": "Portal", "isbn": "978-1-59229-344-5" }
]}
```

Listing 19.7 Array in JSON

The same data is shown in [Listing 19.8](#) represented in XML. Visit http://help.sap.com/abapdocu_740/en/index.htm?file=abenjson_oview.htm for a short description of JSON.

```
<books>
  <book>
    <title>OData</title> <isbn>78-1-59229-907-2</isbn>
  </book>
  <book>
    <title>ABAP</title> <isbn>978-1-59229-859-4</isbn>
  </book>
  <book>
    <title>Portal</title> <isbn>978-1-59229-344-5</isbn>
  </book>
</books>
```

Listing 19.8 Array in XML

Similar to XML, JSON is self-describing, human-readable, and hierarchical. However, unlike XML, JSON does not use end tags, the statements are shorter, and it's quicker to read and write. JSON also supports arrays.

SAP introduced support for JSON with SAP NetWeaver 7.4 and has downported that support to versions 7.02 and 7.31. JSON is supported in ABAP similarly to

XML. We can use the same tools that are used for XML to process JSON. For example, the `CALL TRANSFORMATION` statement can be used to transform JSON similarly to XML. To enable this, SAP has introduced a new `IF_SXML=>CO_XT_JSON` format of the `sXML` library.

SAP has introduced an SAP-specific JSON-XML format that maps JSON to XML and vice versa. When serializing ABAP data objects to JSON and deserializing from JSON to ABAP data objects, we work with `asJSON`, which represents ABAP data. `asJSON` can be seen as the JSON-XML representation of `asXML`. However, unlike `asXML`, there is no intermediate step internally for `asJSON`.

JSON allows ABAP to work with web services that use JSON format for data interchange.

Listing 19.9 shows serialization with `asJSON`.

```
DATA : BEGIN OF sflight,
    carrid TYPE s_carr_id VALUE 'AA',
    connid TYPE s_conn_id VALUE '0017',
END OF sflight.
DATA writer TYPE REF TO cl_sxml_string_writer.
DATA jsonstr TYPE xstring.
writer = cl_sxml_string_writer=>create( type = if_sxml=>co_xt_json ).
CALL TRANSFORMATION ID
SOURCE flight = sflight
RESULT XML writer.
jsonstr = writer->get_output( ).
```

Listing 19.9 JSON Transformation

In Listing 19.9, we're using the `if_sxml=>co_xt_json` format of the `sXML` library with the string writer. Similarly, we can deserialize from JSON to ABAP, as shown in Listing 19.10.

```
DATA json_str TYPE STRING VALUE '{"TEXT":"This is JSON text!"}',
    v_text TYPE string.
CALL TRANSFORMATION ID SOURCE XML json_str
RESULT text = v_text.
```

Listing 19.10 JSON to ABAP

19.9 WebSockets (ABAP Channels and Messages)

Say you have an ALV report that displays up-to-date sales data. This report fetches the sales data from the database and displays it as a report output. While you're looking at the report output, if any new sales are entered in the system, those new sales won't appear in the report.

Because the data was fetched when the report was executed, you'll only see the current data as of the time of the report was run. What if you want the report to always display up-to-date sales data, even if you spend an hour going through the report output? One way of doing this is to refresh the report frequently or use the `CL_GUI_TIMER` class to auto-refresh the report. However, this approach will put an unnecessary load on the database, because the report will be frequently refreshed (polling) to look for new data even when none is available.

ABAP channels enable event-driven communication and implement a publish/subscribe mechanism. The ABAP messaging channel acts as a broker for the message exchange between ABAP sessions in different application servers. Any ABAP session can subscribe to channels, and any ABAP session can publish messages through channels.

What if there was a mechanism through which the report could be informed to refresh as soon as data is updated in the database? This is where ABAP channels and messages come into the picture. Using ABAP channels, the program that updates the data can act as a producer and raise a message to which the report can subscribe as a subscriber.

ABAP channels come in two flavors:

► **ABAP messaging channels**

Designed to send messages between programs running on different application servers

► **ABAP push channels**

Designed to send messages to and from the Internet using WebSockets

You can create ABAP messaging channels and push channels in Object Navigator via the context menu path `CREATE • CONNECTIVITY`. ABAP push channels are used in web applications such as Web Dynpro ABAP, SAPUI5, and more; a discussion of ABAP push channels is beyond the scope of this book. In this section,

we'll discuss ABAP messaging channels and see how they can be used to send messages across sessions.

19.9.1 Creating an ABAP Messaging Channel

ABAP messaging channels allow communication between ABAP programs using messages. These channels use publish-and-subscribe mechanisms and point-to-point communication that allows messages to be exchanged between any SAP NetWeaver AS ABAP programs running on the same application server or on different application servers. With the publish-and-subscribe method, the sender of the message doesn't know the receiver; point-to-point communication enables a message to be sent to a specific receiver.

Different messages that can be sent and received with ABAP messaging channels include text strings, byte strings, and any content that can be serialized by the sender and deserialized by the receiver in a suitable way, such as XML or JSON content. SAP also provides its own communication format called Push Channel Protocol (PCP) for exchanging messages. As of SAP NetWeaver 7.5, the length of the messages that can be exchanged is currently restricted to thirty thousand bytes.

ABAP messaging channels are implemented as repository objects in the SAP system and can be accessed in sender and receiver programs using an OOP interface. The classes of the ABAP messaging channel programming interface use the naming convention `CL_AMC_`, whereas interfaces use the naming convention `IF_AMC_`.

For each ABAP messaging channel communication, the ABAP messaging channel defined as a repository object must exist in the system. You can create an ABAP messaging channel using the context menu path `CREATE • CONNECTIVITY • ABAP MESSAGING CHANNEL` in Object Navigator, or use Transaction `SAMC`.

The following steps walk you through creating an ABAP messaging channel:

1. On the initial ABAP messaging channel screen, right-click the `ABAP MESSAGING CHANNELS` folder and select `CREATE`, as shown in [Figure 19.127](#).
2. Provide a name for the channel and click `OK`.

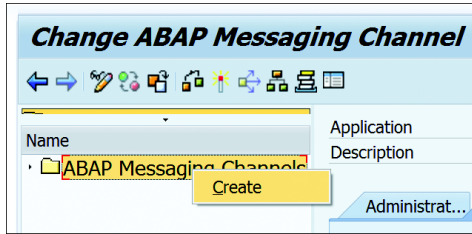


Figure 19.127 Create ABAP Messaging Channel

3. Maintain a description for the channel and other details, as shown in [Figure 19.128](#):

- ▶ **CHANNELS:** Here, you maintain the channel name. This channel name is used to publish or consume messages. The channel name should start with a forward slash (/). For this example, enter "/ping".
- ▶ **MESSAGE TYPE ID:** Maintain the protocol for the channel. You can select TEXT, BINARY, or PCP, based on the type of message. For this example, select TEXT. [Table 19.5](#) lists the interfaces for each message type.

Message Type	Activity	Interface
TEXT (STRING)	Producer/ sender	IF_AMC_MESSAGE_PRODUCER_TEXT
	Consumer/ receiver	IF_AMC_MESSAGE_RECEIVER_TEXT
BINARY (XSTRING)	Producer/ sender	IF_AMC_MESSAGE_PRODUCER_BINARY
	Consumer/ receiver	IF_AMC_MESSAGE_RECEIVER_BINARY
PCP	Producer/ sender	IF_AMC_MESSAGE_PRODUCER_PCP
	Consumer/ receiver	IF_AMC_MESSAGE_RECEIVER_PCP

Table 19.5 Interface for Each Message Type

- ▶ **ACTIVITY SCOPE:** Select SYSTEM, CLIENT, or USER. This defines the scope of the message. For this example, select CLIENT.

- ▶ **AUTHORIZED PROGRAMS:** The message channel can be used only by authorized programs to produce or subscribe to messages. Here, maintain the producer and consumer programs. For this example, we maintained one program to act as a producer and one program to act as a consumer (see [Figure 19.128](#)).

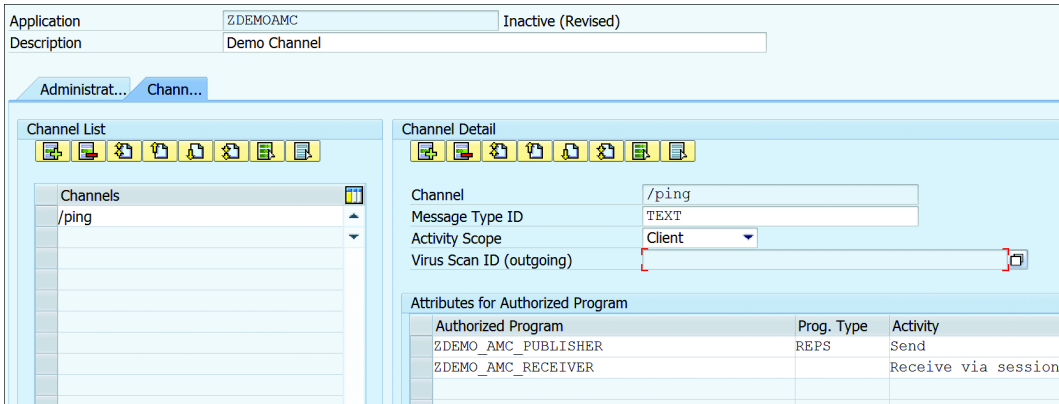


Figure 19.128 Channel Settings

4. Save, then activate the ABAP messaging channel by clicking the **ACTIVATE** icon in the toolbar.

19.9.2 Creating a Producer Program

In the previous step, we registered program `ZDEMO_AMC_PUBLISHER` as the publisher of the message. [Listing 19.11](#) shows sample code to publish a message. This program provides a selection screen on which you can input a message that will then be published as an ABAP message channel message. The subscriber to the ABAP message channel will receive the message.

```
REPORT ZDEMO_AMC_PUBLISHER.
PARAMETERS: message TYPE string.
DATA: lv_application_id TYPE amc_application_id VALUE 'ZDEMOAMC',
      lv_channel_id TYPE amc_channel_id VALUE '/ping',
      lv_extens TYPE amc_channel_extension_id.

DATA: lo_producer_txt TYPE REF TO if_amc_message_producer_text.
DATA: lx_amc_error TYPE REF TO cx_amc_error.

TRY.
```

```

lo_producer_txt ?=
cl_amc_channel_manager=>create_message_producer(
i_application_id = lv_application_id
i_channel_id = lv_channel_id
i_channel_extension_id = lv_extens
).
lo_producer_txt->send( i_message = message ).
CATCH cx_amc_error INTO lx_amc_error.
MESSAGE lx_amc_error->get_text( ) TYPE 'E'.
ENDTRY.

```

Listing 19.11 Code to Publish Message

In [Listing 19.11](#), we defined a reference object of type `if_amc_message_producer_text`. The `create_message_producer` factory method of class `cl_amc_channel_manager` instantiates the producer object. To this method, we pass the application ID and channel ID that we created previously. We're using the `send` instance method to send the message.

19.9.3 Creating a Consumer Program

The consumer program receives the message published by the producer program. We defined `ZDEMO_AMC_RECEIVER` as the consumer program for the message channel.

[Listing 19.12](#) shows sample code for the consumer program `ZDEMO_AMC_RECEIVER`. This program demonstrates receiving any message as soon as the producer program sends it.

```

REPORT ZDEMO_AMC_RECEIVER.
SELECTION-SCREEN BEGIN OF BLOCK part1 WITH FRAME TITLE text-001.
PARAMETERS: time RADIOBUTTON GROUP grp,
             waiting TYPE i DEFAULT 10,
             use_list RADIOBUTTON GROUP grp.
SELECTION-SCREEN END OF BLOCK part1.
DATA: lv_application_id TYPE amc_application_id VALUE 'ZDEMOAMC',
      lv_channel_id     TYPE amc_channel_id     VALUE '/ping',
      lv_extension_id   TYPE amc_channel_extension_id.

DATA: lo_consumer TYPE REF TO if_amc_message_consumer.
DATA: gv_message TYPE string.
DATA: gv_message_received TYPE abap_bool.
DATA: lx_amc_error TYPE REF TO cx_amc_error.

DATA: gt_message_list TYPE TABLE OF string.
DATA: lv_text TYPE string.

```

```

* Implemenation of receiver class for text message type
CLASS lcl_amc_test_text DEFINITION
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.
    CLASS-DATA cv_message TYPE string .

* Interface for AMC messages of type TEXT
  INTERFACES if_amc_message_receiver_text .
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS lcl_amc_test_text IMPLEMENTATION.
  METHOD if_amc_message_receiver_text~receive.
*
*Message handling take place here
*Remark: in this method hold the message be received and the
*main actions should be processed in the main program and
*usually after WAIT statement.
*Any kind of communication, e.g., RFCs, HTTP, and message handling,
*e.g. error message is not supported and will lead to ABAP dump.
    gv_message      = i_message.
* For list processing queue the messages
    APPEND i_message TO gt_message_list.
    gv_message_received = abap_true.
  ENDMETHOD.
ENDCLASS.
START-OF-SELECTION.
  DATA: lo_receiver_text TYPE REF TO lcl_amc_test_text.
  TRY.
    lo_consumer =
      cl_amc_channel_manager=>create_message_consumer(
        i_application_id = lv_application_id
        i_channel_id = lv_channel_id
        i_channel_extension_id = lv_extension_id
      ).
    CREATE OBJECT lo_receiver_text.
    lo_consumer->start_message_delivery(
      i_receiver = lo_receiver_text
    ).
  CATCH cx_amc_error INTO lx_amc_error.
    MESSAGE lx_amc_error->get_text( ) TYPE 'E'.
  ENDTRY.

```

```

IF use_list IS INITIAL.
* Wait until the boolean variable gv_message_received is set to
* true but not longer than waiting time in seconds
  WAIT FOR MESSAGING CHANNELS UNTIL gv_message_received =
    abap_true UP TO waiting SECONDS.
  IF sy-subrc = 8 OR gv_message_received = abap_false.
    gv_message = ')-: Time out ocured and no message
      received'.
  ENDIF.
* Print out the AMC message as popup screen
  CALL FUNCTION 'POPUP_TO_CONFIRM'
    EXPORTING
      titlebar      = 'AMC message'
      text_question = gv_message.
ELSE.
  NEW-PAGE LINE-SIZE 120 NO-TITLE.
  lv_text =
|>>>                                     <<<<| ##NO_TEXT.
  WRITE: /15 lv_text COLOR 1.
  lv_text =
|>>> Double click here to check for arrived messages ! <<<<| ##NO_TEXT.
  WRITE: /15 lv_text COLOR 7 INTENSIFIED.
  lv_text =
|>>>                                     <<<<| ##NO_TEXT.
  WRITE: /15 lv_text COLOR 1.
ENDIF.

AT LINE-SELECTION.
IF gv_message_received = abap_true.
  LOOP AT gt_message_list INTO gv_message.
    CALL FUNCTION 'POPUP_TO_CONFIRM'
      EXPORTING
        titlebar      = 'AMC message'
        text_question = gv_message ##NO_TEXT.
        gv_message_received = abap_false.
    ENDLOOP.
  CLEAR: gt_message_list, gv_message.
ENDIF.

```

Listing 19.12 Consumer Program

In [Listing 19.12](#), we're instantiating the receiver object using the `create_message_consumer` factory method of class `cl_amc_channel_manager`. We defined a `lcl_amc_test_text` local class that implements the `if_amc_message_receiver_text` interface for text message type. The message sent by the producer is received within the `if_amc_message_receiver_text~receive` method.

The code extracts in [Listing 19.11](#) and [Listing 19.12](#) are part of the demo programs `RS_AMC_SEND_MESSAGE_TEXT` and `RS_AMC_RECEIVE_MESSAGE_TEXT`, respectively. You can execute these demo programs in separate sessions to send and receive messages.

19.10 Summary

In this chapter, we discussed all the various interfaces used in SAP. Interfaces are an important aspect of an SAP system and are complicated in nature. Each implementation brings its own challenges and involves consultants across multiple domains.

The topics discussed in this chapter should help you understand the options available to integrate different systems. This in turn should allow you to make an informed decision based on your business requirements.

As technologies evolve, you can expect SAP to add support for more new technologies in the future in order to achieve wider goals. Working on interfaces requires some experience, because business requirements are complicated and no one strategy fits all scenarios. The good news is that SAP provides customization options that allow us to customize programs per our business requirements.

In the next chapter, we'll discuss all the available options to customize standard SAP programs to meet our business requirements.

In this chapter, you'll learn about the different tools and features that can be used to modify and/or enhance repository objects.

20 Modifications and Enhancements

SAP provides standard software that is designed to meet industry-specific requirements. When we implement an SAP system, it comes with a lot of pre-packaged applications that can be used right out of the box. It's possible for a company to implement SAP and start using applications to carry out their day-to-day activities after data migration from legacy systems with little configuration.

However, almost all companies within any industry sector evolve their own business processes and need to customize standard SAP software to meet their business-specific requirements. The process of tailoring standard SAP software to meet company-specific requirements is known as *customization*.

In this chapter, we'll discuss the ways in which a standard SAP system can be customized to meet company-specific requirements. We begin with an overview of the different types of customizations in [Section 20.1](#). We'll then discuss the different types of modifications possible in [Section 20.2](#) through [Section 20.4](#) using the Modification Assistant and the Modification Browser.

We conclude the chapters by looking at enhancements in [Section 20.5](#) through [Section 20.10](#). In these sections, we explore all the different enhancement techniques available in both procedural and object-oriented ABAP programming approaches.

20.1 Customization Overview

As discussed in the introductory chapters of this book, customization can be achieved in three ways:

► **Modifications**

In this type of customization, standard SAP programs are changed by directly editing the code. The major disadvantage of modifications is that the modified code needs to be reimplemented if the program is updated by SAP in a new release or support package.

► **Enhancements**

In this type of customization, standard SAP programs aren't edited directly; instead, enhancements take advantage of certain openings in the code that SAP provides where you can add custom logic to *enhance* applications. Enhancement techniques have evolved over a period of time, with each new iteration trying to make enhancements easy to find and implement.

With enhancements, you don't need to worry about losing changes when the program is upgraded, because enhancement code resides in the customer namespace and you're not editing the standard program code directly. We'll discuss all the available enhancement techniques in this chapter. Enhancements are the preferred way to extend standard SAP programs.

► **Custom developments**

Even though SAP provides many applications out of the box to meet industry requirements, it's not always possible to provide everything a company needs. If there are no standard applications that meet a company's specific requirements or no feasible openings in the standard SAP code to enhance the application, you can develop your own applications in the SAP system. Customer applications should be in the customer namespace; that is, the program name should start with Y or Z.

When customers report bugs in standard SAP programs, SAP reviews the bugs and releases corrections as SAP Notes for customers. Customers can apply SAP Notes in the system using Transaction SNOTE. Once the SAP Note is applied, the standard SAP program that contained the bug will be corrected.

If many customers request similar changes or if standard SAP programs need to be enhanced to meet industry-specific or region-specific requirements, then SAP consolidates all the notes and releases them as a support pack. When a support pack is imported, it will implement all the corrections (SAP Notes) released for that version up to the time of release of the support pack.

Originally, one way of customizing standard SAP programs was to copy the program to a Z program and make the necessary changes in the copied program. For

example, if we feel standard SAP program `RMVKON00` needs certain modifications to meet our requirements, then instead of directly modifying standard program `RMVKON00`, we copy the program to `ZRMVKON00` and make changes in program `ZRMVKON00`. Users then use `ZRMVKON00` instead of `RMVKON00`.

SAP recommended this approach, and because of that, many customers started copying standard programs to customize the software. However, when new updates and bug fixes were released for standard programs, it proved to be a challenge to update all the copied programs to reflect those changes. Clearly, there was a need for better approach to customizing SAP.

Now that you have a better understanding of customization in general, we'll begin looking at the process of implementing modifications.

20.2 Modification Overview

Even though modifications, in which standard SAP programs are edited directly, aren't recommended, many companies still use them. If too many standard programs are modified, it becomes a very time consuming process to upgrade the system or import support packages.

SAP provides two tools for performing and reviewing modifications:

- ▶ **Modification Assistant**

The Modification Assistant tool is integrated into ABAP Workbench. When modifications are carried out using Modification Assistant, the changes are recorded. We'll discuss how this tool can be used in [Section 20.3](#).

- ▶ **Modification Browser**

The Modification Browser tool provides an overview of enhancements and modifications carried out in the system. It allows you to search for modifications integrated using SAP Notes or Modification Assistant. We'll provide more information about the Modification Browser in [Section 20.4](#).

When upgrading the system, the customer can use Transactions SPAU and SPDD to check if the upgrade replaces any modified programs. Transaction SPAU is used to check repository objects such as programs, screens, functions, and the like, whereas Transaction SPDD checks modifications against dictionary objects

such as tables. After the upgrade, all the modifications still need to be reapplied, which can be a time-consuming process.

Modifications can be made in a number of places, including programs, ABAP Class Builder, Screen Painter, Menu Painter, function modules, ABAP Data Dictionary, and documentation.

20.3 Using Modification Assistant

Modification Assistant is integrated into the following ABAP Workbench tools: ABAP Editor, Class Builder, Function Builder, Screen Painter, Menu Painter, ABAP Data Dictionary, text element maintenance, and documentation. When objects are edited using Modification Assistant, the changes are logged and the assistant provides a detailed overview of all the changes in a clear, easy-to-read manner. This simplifies the upgrade process to a great extent.

Modification Assistant is available when you open standard SAP objects in change mode for editing. To open a standard SAP object in change mode, the object should be registered via SAP Support Portal using the SAP Software Change Registration (SSCR) application.

You can access the SSCR application at <https://support.sap.com/sscr>. You'll need the required authorizations to use the SSCR application. Once the object is registered with SAP, you'll be able to generate an access key from within the SSCR application to modify the object.

When you open any standard SAP object that wasn't previously modified by the same user, the system will present a REGISTER OBJECT window (see [Figure 20.1](#)). You can click on the PERFORM REGISTRATION IN THE SAP SUPPORT PORTAL link (see [Figure 20.1](#)) to register the object. The modification registration on the SAP Support Portal is typically carried out by your manager or SAP Basis administrator. Once the modification is registered with SAP, enter the access key to open the standard SAP object for editing.

In the following subsections, we'll look at the different ABAP tools and programs that can be modified via Modification Assistant.

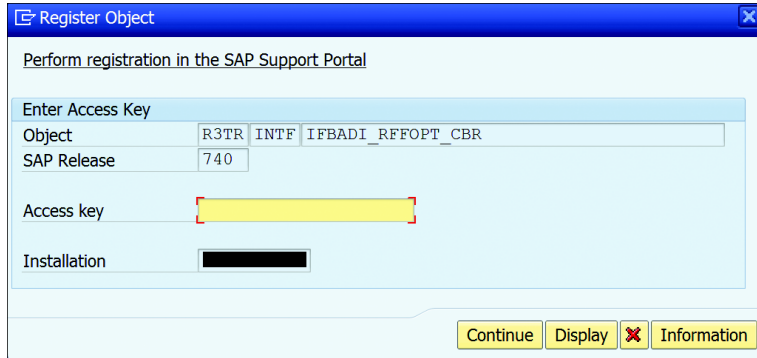


Figure 20.1 Register Object Window

20.3.1 Modifications to Programs

Modification Assistant provides buttons that enable you to edit objects, as shown in [Figure 20.2](#). When editing an object supported by Modification Assistant, you can use these buttons from the application toolbar or follow the menu path **EDIT • MODIFICATION OPERATIONS**.



Figure 20.2 Modification Assistant

Using Modification Assistant, you can insert, replace, or delete code. When you change code using Modification Assistant, the original code is commented, and a window is provided for you to add your code.

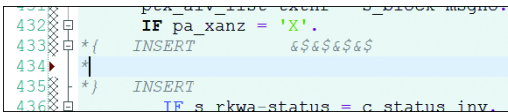
If you wish to undo the changes, keep the cursor on the code that was changed and click the **UNDO** button. This will delete your code and uncomment the original code. When you modify the code in an ABAP program, the changes are logged and assigned to the module (dialog module, procedure, event, etc.) in which the modification is carried out. You can check this log by clicking the **MODIFICATION OVERVIEW** button to display all the modifications that were carried out to the program module-wise.

Modifications to programs happen via buttons in ABAP Editor: **INSERT**, **REPLACE**, and **DELETE**. The following subsections walk through using each of these buttons.

Insert

Using the INSERT button, you can insert additional code into the original program. To do so, select the line where you want to insert new code and click the INSERT button from the toolbar. This will add INSERT comment lines (see [Figure 20.3](#)) to identify the type of modification being carried out.

The line where the cursor was positioned will be input-enabled so that you can add your code, as shown in [Figure 20.3](#). Press to add additional lines of code.



```

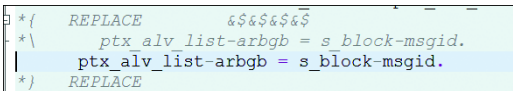
432 |      IF pa_xanz = 'X',
433 |      *{  INSERT          &$$$&$$$
434 |      *|
435 |      *}  INSERT
436 |      IF s_rkwa-status = c_status inv.

```

Figure 20.3 Inserting Code

Replace

If you want to replace existing code, select the line you want to replace and click the REPLACE button. To replace multiple lines of code, highlight the lines and then click the REPLACE button. This will add REPLACE comment lines to identify the type of change and copy the selected line in the editable area while commenting out, as shown in [Figure 20.4](#). You can edit the copied line and press to add more lines.



```

*{  REPLACE          &$$$&$$$
*|  ptx_alv_list-arbgb = s_block-msgid.
|  ptx_alv_list-arbgb = s_block-msgid.
*}  REPLACE

```

Figure 20.4 Replacing Code

Delete

You can delete an existing line by selecting the line and clicking the DELETE button. This will comment out the line in the program. To delete multiple lines, select all the lines you want to delete before clicking the DELETE button. The DELETE comment line is inserted to identify the type of change.

20.3.2 Modifications to Class Builder

Modification Assistant also allows you to modify classes and interfaces within Class Builder. You can modify the class or interface by adding attributes, methods, events, and internal data types. You can also modify the existing components of the class or interface.

Table 20.1 describes different components that can be modified.

Component	Modifications for Classes	Modifications for Interfaces
Attribute	Visibility (public, private, protected), kind (static, instance, constant), type, initial value, and read-only flag	Kind (static, instance-dependent, constant), type, initial value, and read-only flag
Method	Visibility, kind, final flag, and abstract flag	Kind
Event	Visibility and kind	Kind
Type	Internal data type's visibility and type	None

Table 20.1 Modification Options for Class/Interface Components

To modify the components of the global class/interface or to add new components, open the class or interface in change mode using Class Builder. Class Builder will make the relevant fields input-enabled (see Figure 20.5), allowing you to edit the relevant properties for the component.

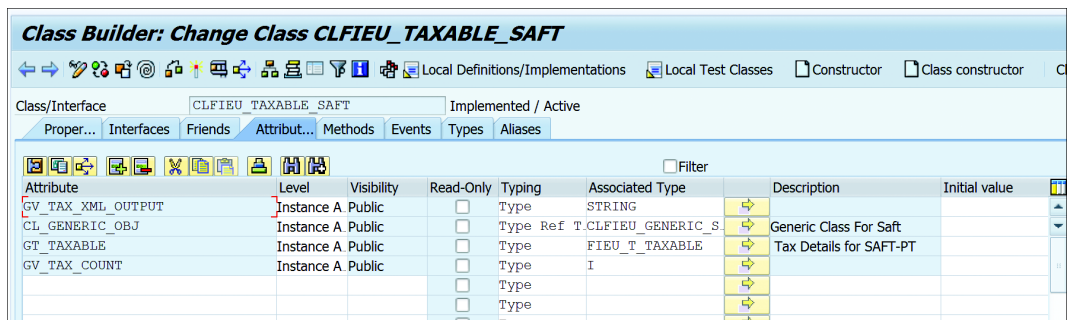


Figure 20.5 Editing Class Components

Similarly, you can change the method's signature by adding parameters or exceptions or by modifying original parameters. You can edit the code in a method

using the INSERT, DELETE, REPLACE, and UNDO buttons as explained in [Section 20.3.1](#) (see [Figure 20.6](#)).

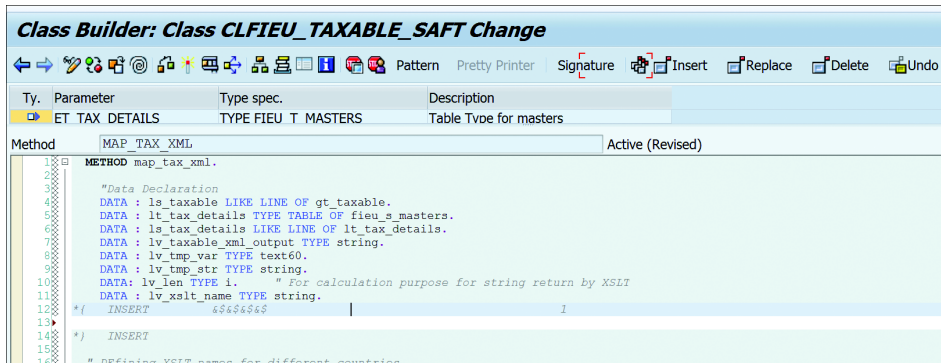


Figure 20.6 Modifying Method Code

20.3.3 Modifications to Screen Painter

Within Screen Painter, you can modify screen attributes, the element list, screen flow control, and layout. Screen attributes can be changed any way you want—for example, by changing the screen type or screen settings. In the screen flow logic, you can change existing events or add new events. Just like in ABAP Editor, you can add, replace, or delete statements in the flow logic by using the INSERT (add), REPLACE, and DELETE buttons (see [Figure 20.7](#)).

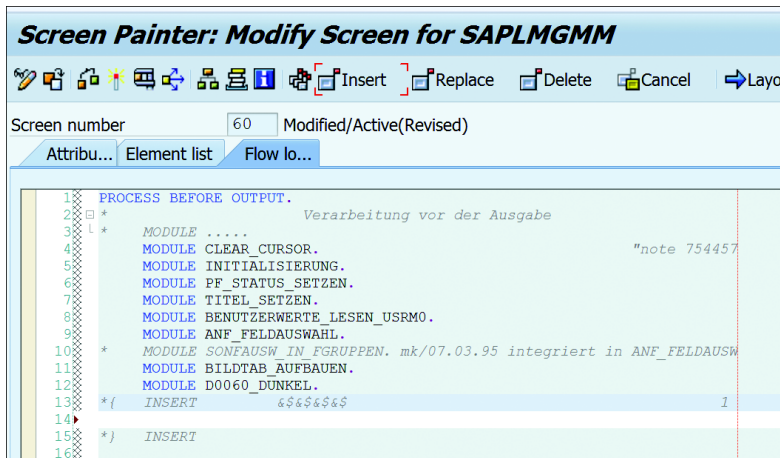


Figure 20.7 Screen Flow Logic

In the layout editor, you can add or modify any screen elements, but you can't delete existing screen elements. However, you can make any screen element invisible by changing the appropriate attribute for the screen element.

20.3.4 Modifications to Menu Painter

You can modify buttons, icons, and functions in the menu bar, and more within Menu Painter. Use the INSERT and DELETE buttons in the toolbar to add or delete functions, as shown in [Figure 20.8](#).

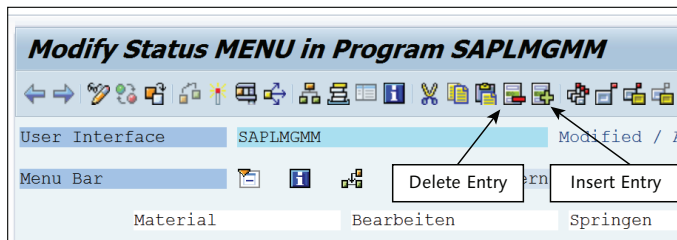


Figure 20.8 Add or Delete Functions in Menu Painter

20.3.5 Modifications to ABAP Data Dictionary

Using Modification Assistant, you can modify tables, structures, and data elements within ABAP Data Dictionary. ABAP Data Dictionary tables and structures are modified by adding append structures, as discussed in [Chapter 10](#). Append structures can be added by opening the table or structure in display mode in ABAP Data Dictionary, so you don't need to generate the access key from SSCR to modify them.

To modify any attributes of a data element using Modification Assistant, first open the data element in change mode in ABAP Data Dictionary. Select the attribute you want change and click the CHANGE SUBJECT button in the toolbar, as shown in [Figure 20.9](#).

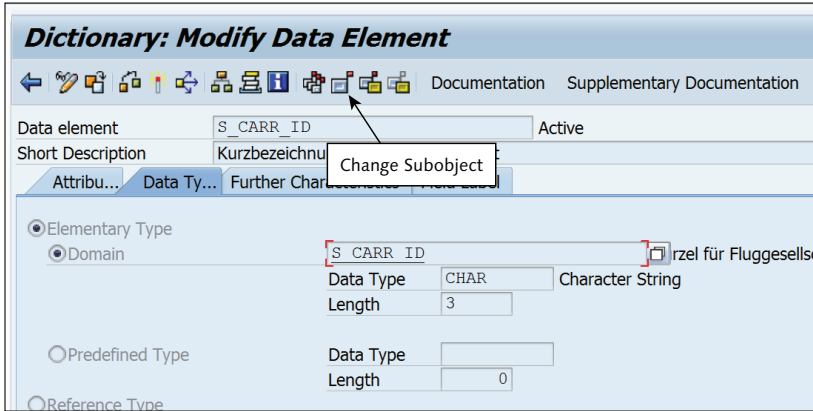


Figure 20.9 Modifying Data Element

20.3.6 Modifications to Function Modules

You can create new function modules in a customer namespace and add them to the standard function group or modify existing ones using Modification Assistant. The source code of the function module can be changed just like in ABAP Editor via the INSERT, REPLACE, and DELETE buttons in the toolbar.

You can also modify the parameter interface of the function module by adding new parameters or by changing the attributes of the existing parameters. When adding new parameters, the parameter will be made optional automatically by setting the OPTIONAL flag if no default value is provided (see [Figure 20.10](#)).

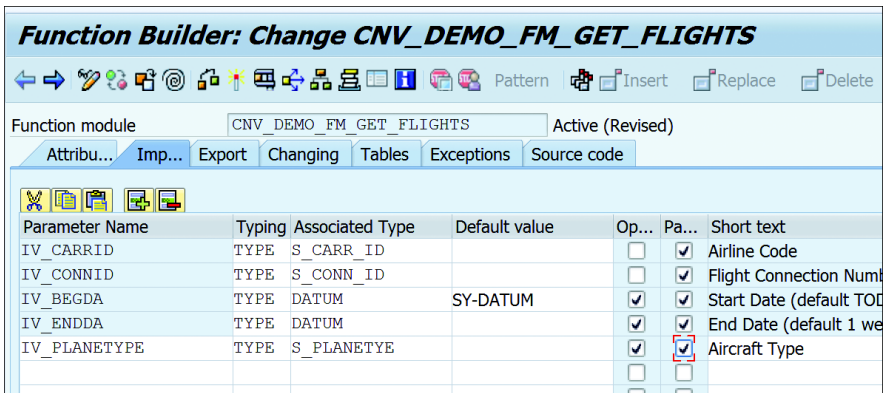


Figure 20.10 Modifying Function Module

20.3.7 Resetting to Original

If you want to reset a repository object to its original state and delete all the modifications carried out for the object, you can do so by following the menu path **EDIT • MODIFICATION OPERATIONS • RESET TO ORIGINAL** on the initial screen of the respective ABAP Workbench tool (see [Figure 20.11](#)).

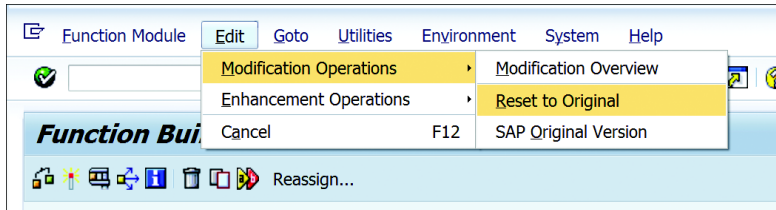


Figure 20.11 Reset Modifications from ABAP Workbench Tool

You can also use Modification Browser (Transaction SE95) to reset modifications by selecting the repository object and choosing **MODIFICATIONS • RESET TO ORIGINAL** from the menu (see [Figure 20.12](#)).

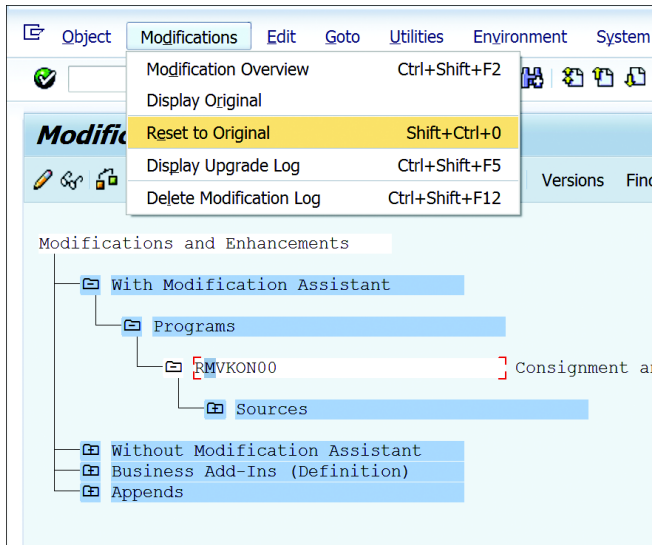


Figure 20.12 Reset Modifications from Modification Browser

You can only reset the complete repository object to ensure consistency. If you want to undo only specific modifications to the object, then you need to manually

identify a change and undo it. When the repository object is reset, all the modifications will be deleted and can't be restored. Any subobjects belonging to the repository object also will be reset to their original states.

20.4 Using Modification Browser

Modification Browser provides an overview of the modifications and enhancements carried out in the system. You can open Modification Browser using Transaction SE95. The initial Modification Browser screen provides a selection screen to search for modifications using different criteria, as shown in [Figure 20.13](#).

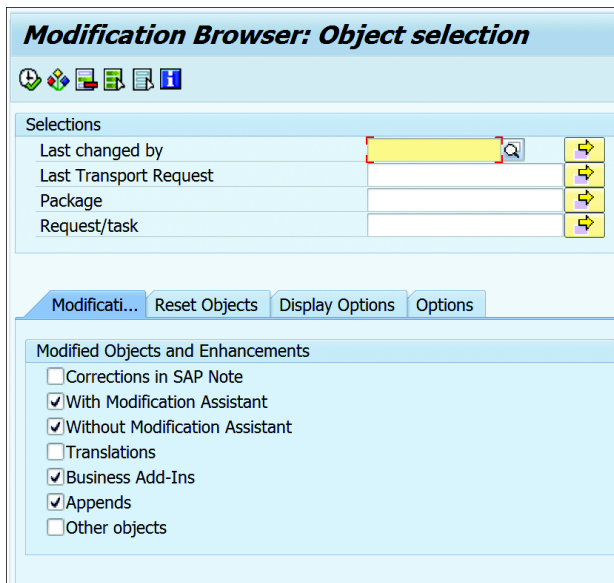


Figure 20.13 Modification Browser Initial Screen

Here, you can search for modifications that are applied through SAP Notes or carried out using Modification Assistant. You can also search for modifications that are carried out without Modification Assistant or changes that were performed before Modification Assistant was introduced. Changes to BAdIs and ABAP Data Dictionary tables, views, and search helps can also be checked via Modification Browser.

Once the selection is made and executed, Modification Browser displays all objects grouped per the category selected under the DISPLAY OPTIONS tab on the initial screen, as shown in [Figure 20.14](#).

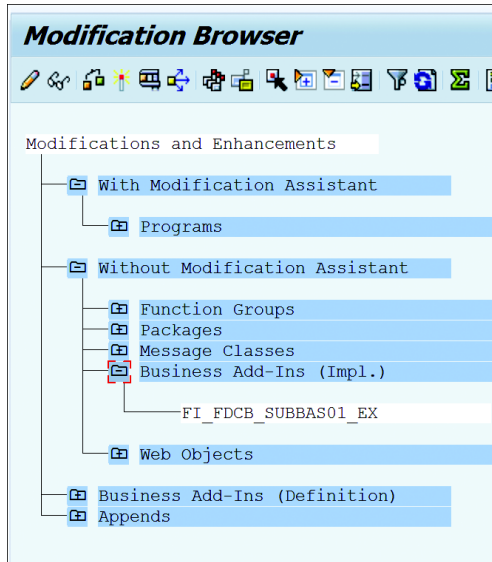


Figure 20.14 List of Modified Objects

Modification Browser supports forward navigation: Double-click a displayed object name to open the object in the appropriate tool.

20.5 Enhancements Overview

To address problems with modifying standard SAP programs and to enable customers to customize standard SAP applications, SAP provides enhancement techniques. These enhancement techniques have evolved over the years, and each new technique tries to address problems with the previous techniques.

In order to ensure backward compatibility, SAP never removes legacy code; therefore, many different enhancement techniques are available, including user exits, customer exits, business transaction events (BTEs), classic BADIs, new BADIs, and enhancement points. We'll discuss all of these techniques in the following sections, which should not only enable you to work with new techniques

but also help you understand and maintain enhancements implemented using old techniques.

Many companies are reluctant to convert their old code to new technology; it requires a lot of effort without necessarily providing any new benefits over the old implementation. SAP recommends that we use the latest available technology for any new implementations but doesn't force customers to convert their existing implementations.

The enhancement concept works on the basis that SAP can leave an opening in the code while developing the standard SAP program, and customers can add their code in these openings (*hooks*) to enhance the application. This concept depends heavily on the developer of the standard SAP program anticipating where customers might want to add their code in the standard SAP program and providing a hook accordingly.

In some situations, suitable hooks may not be available for you to enhance the application. For enhancements to meet customer requirements, the hook should be provided at the right place in the program flow and provide access to the data that a customer wants to change.

For example, when a user creates a sales order in a standard SAP application, the application doesn't put any restriction on item quantity. However, if your organization deals with wholesale retail and you want to put a restriction on the minimum quantity that can be ordered for each item, you need to validate the order quantity for the item and allow only bulk orders over a specified order quantity to be saved.

To enhance the standard SAP application to implement this custom requirement, you need a hook that's triggered before the document is saved and provides access to the item quantity for validation. If the developer of the standard SAP application has provided a hook *after* the document is saved, it's of no use for your requirement.

When studying if a particular enhancement meets our requirements, we look at when the enhancement is triggered in the standard SAP program flow and what data is available to manipulate. If we can't find any enhancements for a standard SAP program that meet our requirements, we need to explore other options for customizing the application, such as modifications or new developments. This

problem is addressed to a great extent with enhancement points, which we'll discuss in [Section 20.9](#).

20.6 User Exits

User exits are one of the oldest enhancement techniques introduced by SAP. User exits are implemented as a call to a subroutine in which we can add custom code. The developer of the standard SAP program inserts a `PERFORM` statement (see [Figure 20.15](#)) at a specific point in the program code that acts as a hook. The subroutines are modularized in an include program in which we can edit a subroutine to add custom code.

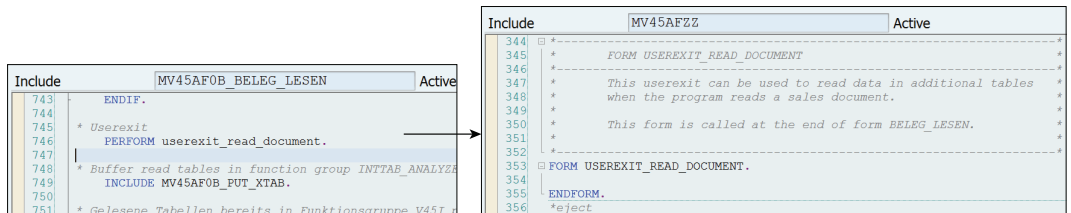


Figure 20.15 User Exit

All the include programs that contain user exits are saved in the `VMOD` package. We can check the objects belonging to the `VMOD` package via Object Navigator. User exits are typically provided for applications in the SAP ERP Sales and Distribution (SD) module.

When the standard program is called, it calls the user exit subroutine when the program encounters the `PERFORM` statement at a specific point in the program flow. Once the subroutine is called, all the custom code added to the subroutine is executed at that specific point in the main program flow. Within the user exit, you can access the global data objects of the main program and manipulate them per your requirements. For example, if the user exit is called before the sales order is saved, we can validate the item quantity by accessing the global data of the program to ensure each item meets the minimum order quantity requirement.

Even though user exits are convenient, they can be unstable because they manipulate the global data of the program. Also, there is no user-friendly way to search

for user exits; you need to dig through the include programs in which the subroutines are maintained. Other than checking the comment lines maintained for the subroutine, user exits don't offer any documentation to explain their purposes. They also don't include support for the Switch Framework, so if you don't want to use an enhancement in the future, you'll have to manually delete or comment out the code.

The *Switch Framework* allows you to externally control the visibility of a repository object, so when an enhancement technique supports the Switch Framework, it can simply be turned off when not required, without needing to delete the enhancement.

All enhancement techniques other than user exits support the Switch Framework. SAP no longer provides user exits for new standard SAP programs, and many user exits have alternatives provided via modern enhancement techniques.

20.7 Customer Exits

Customer exits, also known as *CMOD enhancements*, are the successor to user exits. Customer exits implement hooks using function modules. Customer exit function modules are implemented with the naming convention `EXIT_XXXXX_NNN`, where `XXXXX` is the program name in which the customer function is called and `NNN` is the exit number. In a standard SAP program, a function module is called using the `CALL CUSTOMER-FUNCTION 'NNN'` statement.

For example, as shown in [Figure 20.16](#), the customer exit function module `EXIT_SAPLV60B_001` is called from the standard SAP program `SAPLV60B`. The function module contains a fixed interface and includes an include program that's in the customer namespace.

When the function module is called, certain data from the main program is exported to the function module, which can be manipulated by writing the code within the Z include program that's included by default in the function module.

With customer exits, we can only work with the data available to the function module through its parameter interface. You can't change the global data of the calling program when using customer exits.

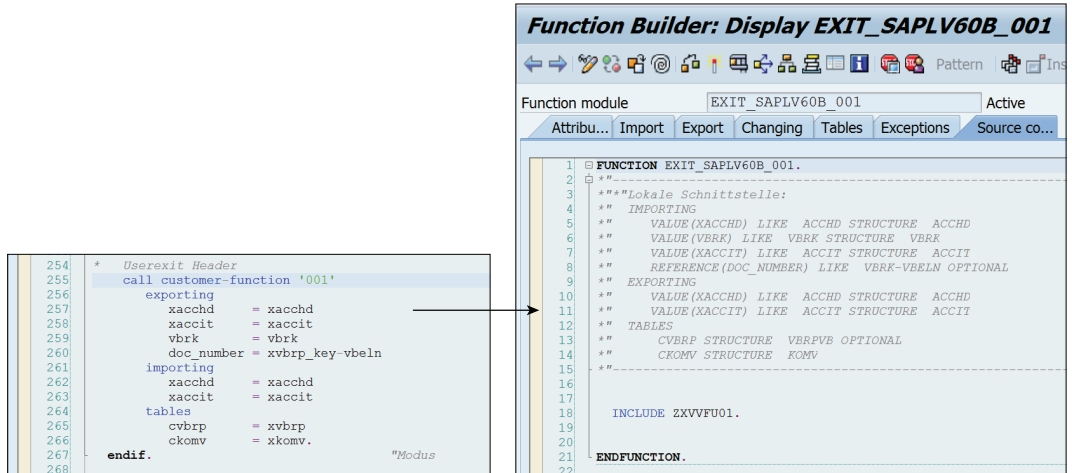


Figure 20.16 Customer Exit

This fixed interface restricts what can be achieved from within a customer exit. For example, if you want to manipulate the item data of a document and none of the customer exits provide the item data in the interface, then you can't use any of the available customer exits for your requirements. Similarly, like user exits, where the customer exit is called in the program flow also plays an important role in deciding if the available customer exit meets your requirement.

Customer exits are implemented by creating a project via Transaction CMOD. They support the Switch Framework, which allows you to turn off an enhancement if you no longer need it. Another advantage of customer exits is that you can search for available enhancements easily using Transaction SMOD (see [Figure 20.17](#)). For example, you can search for all the customer exits available for a particular application component or application package.

There are four types of customer exits that can be implemented:

► **Function module exits**

Function module exits are implemented to enhance the functionality of the program by manipulating the data per customer requirements. For example, if you want to validate the item quantity, then you can implement a suitable function module exit that provides access to the required data.

► **Screen exits**

Screen exits allow you to enhance standard SAP program screens to add custom fields. For example, if you want to capture additional information while creating a purchase order using the standard SAP Transaction ME21N, then you can use the screen exit to add additional fields to the Transaction ME21N screen.

► **Menu exits**

Using a *menu exit* (or *GUI exit*), you can enhance the standard SAP menu. You can add custom function codes to the GUI status of a standard SAP transaction to provide additional functionality; for example, you can add additional menu items to standard SAP applications to provide additional functionality for the user.

► **Field exits**

Field exits allow you to process the input fields of a screen. These are obsolete and should no longer be used; you should use conversion routines instead. Because these exits are no longer used and not recommended, we won't discuss them further here.

There are different ways to search for a customer exit. You can search using Transaction SMOD, in which you can use help in the ENHANCEMENT field to search based on the exit name, short description, package, or application component, as shown in [Figure 20.17](#). Another method to search for available enhancements is to use lookup table MODSAP in ABAP Data Dictionary. All enhancements are saved in database table MODSAP, so you can search table MODSAP by using wild cards for a program name (e.g., SAPLV60B).

In addition, customer exits are called in standard SAP programs using the `CALL CUSTOMER` statement; therefore, you can search the standard SAP program code using the `CALL CUSTOMER` keyword to identify the customer exits called in the program. This isn't a straightforward method to find customer exits and may not be useful if many other programs are called from the main program and the call to the customer exit is implemented in one of the called programs.

In this section, first we'll walk through the process of creating a customer exit. Then, we'll look at function module exits, screen exits, and menu exits in more detail.

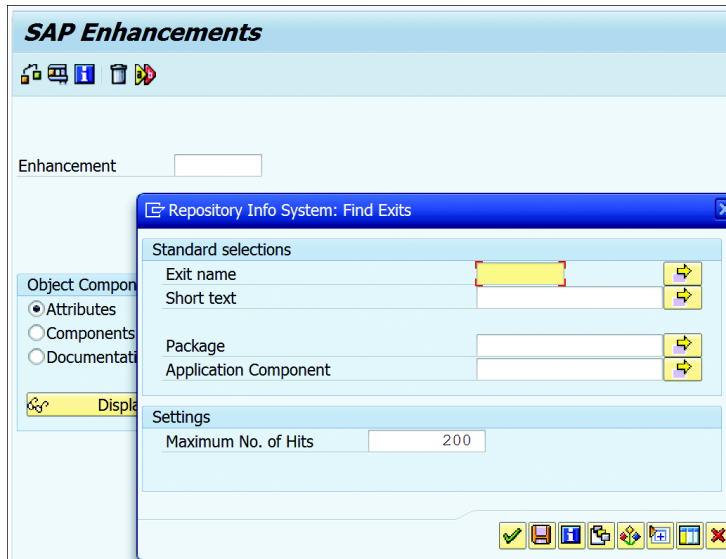


Figure 20.17 Using Transaction SMOD to Search for Enhancements

20.7.1 Create a Customer Exit

The following steps walk you through implementing a customer exit:

1. First, create a project using Transaction CMOD. On the initial screen of Transaction CMOD, enter the name of the project and click CREATE, as shown in [Figure 20.18](#).

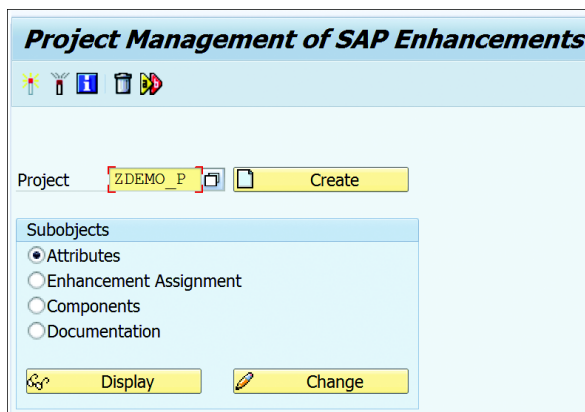


Figure 20.18 Transaction CMOD Initial Screen

2. Maintain a short description for the project and save the project.
3. To assign an enhancement, click the **ENHANCEMENT ASSIGNMENTS** button (see [Figure 20.19](#)).

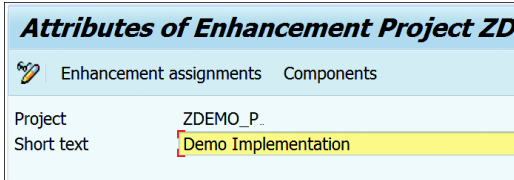


Figure 20.19 Project Attributes

4. On the next screen, enter the name of the enhancement under the **ENHANCEMENT** column and press , as shown in [Figure 20.20](#).

If the enhancement is already used in a different project, the application will show an error message displaying the project in which this enhancement is used (see [Figure 20.21](#)). In such cases, you can edit the enhancement in the existing project or delete it from the existing project to add it to the new project.

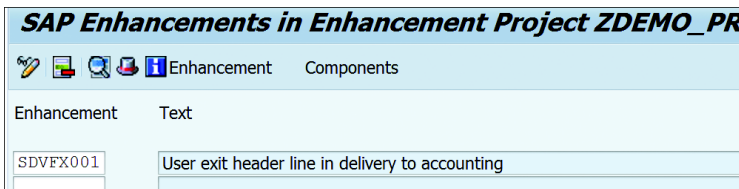


Figure 20.20 Assigning Enhancement to Project

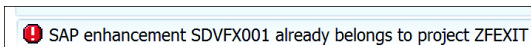
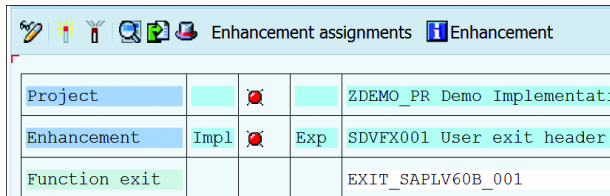


Figure 20.21 Error Message if Enhancement Is Already Implemented

5. Save the project.
6. Click the **COMPONENTS** button in the toolbar to see the components of the enhancement. You can double-click the displayed component to open it in the appropriate ABAP Workbench tool. In this case, the enhancement consists of one function exit, as shown in [Figure 20.22](#).

- Double-click the function module name to open it in Function Builder. Within this function module, you'll see a Z include program. Double-click the Z include program to create it and write your code. Make sure to activate the Z include program.

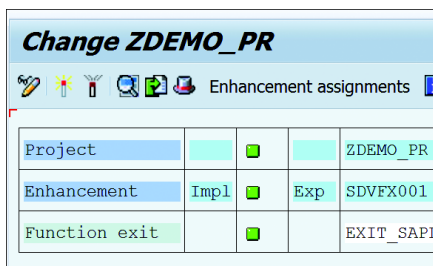


The screenshot shows a table titled "Enhancement assignments" with a toolbar above it. The table has three rows and four columns. The first row is "Project" with a green cell, a red dot, a green cell, and "ZDEMO_PR Demo Implementati". The second row is "Enhancement" with "Impl", a red dot, "Exp", and "SDVFX001 User exit header". The third row is "Function exit" with a green cell, a red dot, a green cell, and "EXIT_SAPLV60B_001".

Project				
Project		●		ZDEMO_PR Demo Implementati
Enhancement	Impl	●	Exp	SDVFX001 User exit header
Function exit		●		EXIT_SAPLV60B_001

Figure 20.22 Enhancement Component

- Once the Z include program in the function module is implemented, you can go back and activate the enhancement. To do so, click the DISPLAY/CHANGE button (if you aren't already in change mode) so that the ACTIVATE and DEACTIVATE icons appear in the toolbar (see [Figure 20.23](#)). Click on the ACTIVATE icon to activate the enhancement, at which point the red dot next to the objects will change to green.



The screenshot shows a table titled "Change ZDEMO_PR" with a toolbar above it. The table has three rows and four columns. The first row is "Project" with a green cell, a green square, a green cell, and "ZDEMO_PR". The second row is "Enhancement" with "Impl", a green square, "Exp", and "SDVFX001". The third row is "Function exit" with a green cell, a green square, a green cell, and "EXIT_SAPL".

Project				
Project		■		ZDEMO_PR
Enhancement	Impl	■	Exp	SDVFX001
Function exit		■		EXIT_SAPL

Figure 20.23 Activating Enhancement

- At this point, you can test the application to see if your enhancement is being called by checking the expected behavior or by adding a breakpoint in the function module you just implemented. If you want to deactivate the enhancement at any point in the future, simply click the DEACTIVATE icon (see [Figure 20.23](#)). Once deactivated, the enhancement will not be called by the standard program.

SAP also implements many enhancements and supplies them in deactivated mode by default. Customers can activate these enhancements to implement additional

functionality. You can activate any enhancement implementation in Transaction CMOD.

Implementing enhancements is straightforward; most of the complications lie in identifying the enhancements that meet your requirements. When working with enhancements, you'll spend most of your time searching for a suitable enhancement.

20.7.2 Function Module Exits

Function module exits are called in standard SAP programs that allow you to work with the data supplied using the function module interface. To evaluate if a particular function module exit meets your requirements, you must check at what point in the program flow the exit is called and what data is available through its interface.

Once you find a suitable enhancement, you can implement the enhancement by creating a project in Transaction CMOD. Customer exits can be implemented only once, meaning that if an enhancement is already implemented in a different project, you can't implement it again in a new project. We recommend that you implement all related enhancements in a single project so that they're easy to maintain and can be activated and/or deactivated together.

20.7.3 Screen Exits

Screen exits allow you to enhance standard SAP screens to add additional fields. When a field is added to the screen, the corresponding standard SAP table also needs to be enhanced to add the extra field. SAP provides customer includes in standard SAP tables in which you can add your extra fields. Implementing a screen exit is similar to implementing function module exits; you first create a project and then add the enhancement to the project.

To allow screen enhancements, SAP calls a subscreen in the PBO event of the main screen via the `CALL CUSTOMER-SUBSCREEN` statement, as shown in [Figure 20.24](#). You're allowed to edit this subscreen to add custom fields. Once the fields are added to the subscreen, they'll be displayed on the main screen as though they're part of it. The same fields should be added to the corresponding standard SAP table in the provided CI include. The fields names should be prefaced with ZZ.

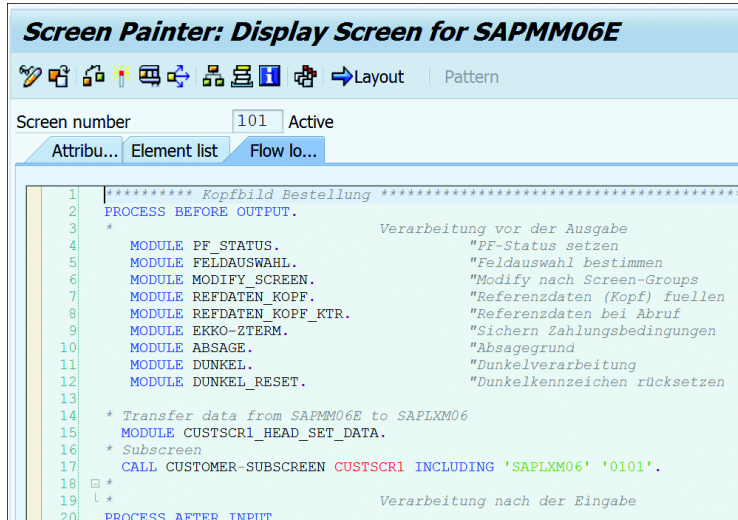


Figure 20.24 Calling Screen Enhancement

For example, Transaction ME21N belongs to the ME package. If you search for available enhancements for the ME package in Transaction SMOD, you'll see the results shown in [Figure 20.25](#).

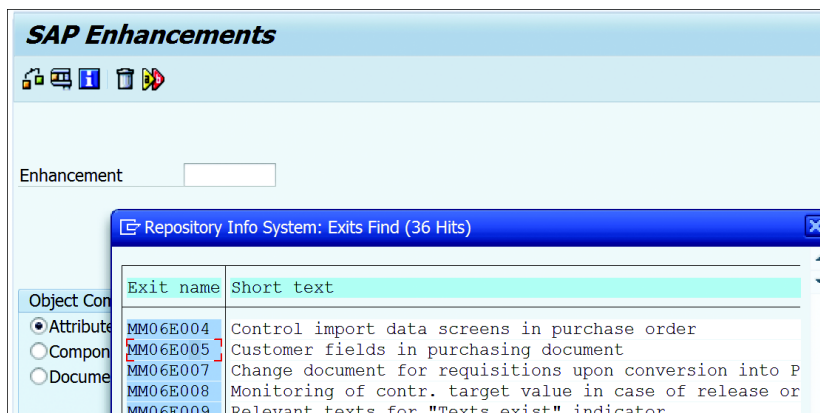


Figure 20.25 Enhancements for ME Package

If you open the MM06E005 enhancement and look at the components, you'll see that the enhancement consists of function module exits, screen exits, and CI includes, as shown in [Figure 20.26](#).

Under the SCREEN AREAS section, note that information about the CALLING SCREEN and CALLED SCREEN with the respective program names and screen numbers is listed. To add custom fields, you edit the screen of the called screen. For example, in [Figure 20.26](#), we're adding custom fields to screen 0101 of program SAPLXM06 to add extra fields to a PO header. The same fields should be added to the CI include of the purchase order header table. In this case, we'd add the fields to CI_EKKODB.

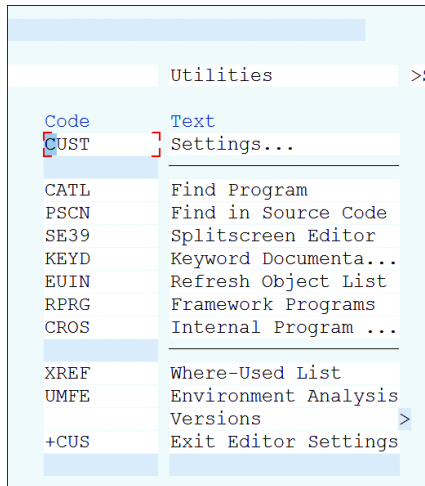
Components in SAP Enhancement MM06E005					
Function module exits					
Function module	Short Text				
EXIT_SAPMM06E_006	Export Data to Customer Subscreen for Purchasing Do...				
EXIT_SAPMM06E_007	Export Data to Customer Subscreen for Purchasing Do...				
EXIT_SAPMM06E_008	Import Data from Customer Subscreen for Purchasing ...				
Function codes					
Program	Code	Short text			
Screen areas					
Calling screen	No.	Area	Called screen	No.	Short Text
SAPMM06E	0101	CUSTSCRI	SAPLXM06	0101	Subscreen: PO header
SAPMM06E	0111	CUSTSCRI	SAPLXM06	0111	Subscreen: PO item
SAPMM06E	0201	CUSTSCRI	SAPLXM06	0201	Subscreen: agreement hea...
Includes					
CI_EKKODB					
CI_EKPODB					

Figure 20.26 Components of Enhancement MM06E005

20.7.4 Menu Exits

Menu exits enable you to add custom function codes to standard SAP applications. This allows you to provide additional functionality when the function code is selected. Customer function codes are named +CUS.

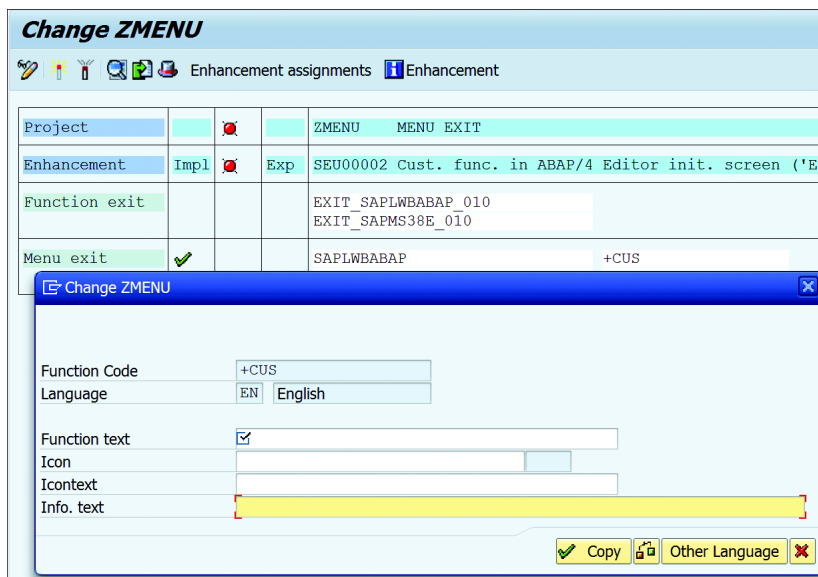
For example, if you look at the GUI status for ABAP Editor (from ABAP Editor, follow the menu path SYSTEM • STATUS and double-click the GUI STATUS field), you'll see that a customer function code exists under the UTILITIES menu, as shown in [Figure 20.27](#). When we implement the enhancement and activate it, the menu option will be added to ABAP Editor.



Code	Text
CUST	Settings...
CATL	Find Program
PSCN	Find in Source Code
SE39	Splitscreen Editor
KEYD	Keyword Documenta...
EUIN	Refresh Object List
RPRG	Framework Programs
CROS	Internal Program ...
XREF	Where-Used List
UMFE	Environment Analysis Versions
+CUS	Exit Editor Settings

Figure 20.27 Customer Function Code

To implement the menu exit, we create the project (or use an existing project) in Transaction CMOD, similar to other enhancements. Once the enhancement is added to the project, navigate to COMPONENTS and double-click the displayed menu exit, as shown in [Figure 20.28](#).



Project			ZMENU MENU EXIT
Enhancement	Impl	Exp	SEU0002 Cust. func. in ABAP/4 Editor init. screen ('En
Function exit			EXIT_SAPLWBABAP_010 EXIT_SAPMS38E_010
Menu exit	✓		SAPLWBABAP +CUS

Change ZMENU

Function Code: +CUS
 Language: EN English
 Function text:
 Icon:
 Iconcontext:
 Info. text:

Copy Other Language

Figure 20.28 Menu Exit

Here, we're implementing the `SEU00002` enhancement for ABAP Editor. Provide the function text and other optional information. Each menu exit will have a corresponding function exit in which you can write the code to perform any actions when the menu is selected. In this example, we'll write the code in function exit `EXIT_SAPLWBABAP_010`. Once the enhancement is activated, you should be able to see the menu option in the standard SAP application.

20.8 BAdIs

Business Add-Ins (BAdIs) were first introduced with the SAP ERP 4.7 release. User exits and customer exits use a procedural programming approach; BAdIs were SAP's first attempt at an object-oriented approach. As you may have guessed, BAdIs are provided as classes, the methods of which we can use to implement custom functionality.

In addition to being object-oriented, BAdIs provide a couple of advantages over customer exits: They can be implemented multiple times, and each implementation can be restricted using filter criteria.

In this section, we'll provide an overview of the structure of a BAdI, then dive into steps for defining and implementing BAdIs. We'll then look at implementing fallback classes for multipurpose BAdIs before looking at the steps for calling a BAdI.

20.8.1 Overview

A BAdI has two parts: a *definition part* and an *implementation part*. SAP defines a BAdI using an interface that consists of a method (or a set of methods). We can maintain the implementation for this BAdI by defining an implementation class, which uses the BAdI interface.

The class `CL_EXITHANDLER` is used in the program to check if the implementation of the BAdI is active and to get a reference to the BAdI. The `get_instance` static method of the `cl_exithandler` global class queries the database table to see if at least one implementation of the BAdI is active and accordingly returns the reference to the BAdI implementation. This approach allows you to have multiple implementations, with each implementation called based on a filter value set for the implementation.

However, this process has one major disadvantage: It slows applications down considerably, because they now need additional database accesses to check if the BAdI is implemented. The problem is compounded when an application has too many BAdIs.

To address this problem, SAP has introduced new *kernel-based BAdIs* in the SAP ERP 6.0 system, in which this check is moved from the database level to the kernel level. Kernel-based BAdIs are called *new BAdIs*, and the old ones are called *classic BAdIs*.

You could create classic BAdI definitions prior to SAP ERP 6.0, but no longer; as of SAP ERP 6.0, you can only create new BAdIs. In fact, SAP has changed the Enhancement Framework completely with the release of SAP ERP 6.0.

You may be thinking that if enhancements are ways for SAP to provide openings in standard SAP applications, so we should only need to implement BAdIs. Why, then, would we need to create BAdI definitions? The answer is that BAdIs provide such great flexibility to enhance applications that we can use them for our custom applications as well. For example, say you're developing an application that will be used in many plants, and each plant wants to do certain things differently.

To accommodate such a requirement, you may have to construct a `CASE` statement in the program and write different logic for each plant, as shown in [Listing 20.1](#). The challenge here is that every time a particular plant wants to change certain things, you need to change the logic in the program. Because this program is used by many plants, you need to perform intensive testing to ensure no other plants are effected by the change.

```
CASE werks.
WHEN '1000'.
  " Write custom logic for plant 1000
WHEN '2000'.
  " Write custom logic for plant 2000
WHEN OTHERS.
  " Write logic for other plants
ENDCASE.
```

Listing 20.1 Logic Using CASE Statement

However, you can accommodate such requirements easily by using BAdIs. First, you define a BAdI that's called in the program. Multiple implementations for this BAdI can be maintained, with each implementation having a plant number as the

filter criterion. The custom logic is maintained in each respective implementation. This way when a particular plant needs a change in logic, only the logic in the respective implementation needs to be changed. Similarly, if the plant no longer needs the specific logic to be executed, you can simply deactivate the implementation.

In this section, we'll create a new BAdI and understand the process to define and implement the BAdI. There are two transactions available to work with BAdIs: Transaction SE18 is used to define the BAdI (and to search for existing BAdIs), and Transaction SE19 is used to implement the BAdI.

To determine all the classic BAdIs that are called in a standard SAP application, you can set a breakpoint in the `get_instance` method of the `cl_exithandler` class and run the application. The breakpoint is triggered each time a classic BAdI is called in the application. New BAdIs are instantiated differently, so they don't trigger this method.

20.8.2 Defining a BAdI

There are three steps involved in defining a BAdI:

1. Create an enhancement spot.
2. Create the BAdI definition.
3. Create the BAdI interface.

In the following sections, we'll walk through each of these steps.

Creating Enhancement Spot

Enhancement spots act as wrappers for BAdI definitions. One enhancement spot can have multiple BAdI definitions.

To create an enhancement spot, proceed with the following steps:

1. Go to Transaction SE18 (BAdI Builder), and select the **ENHANCEMENT SPOT** radio button. Provide a name for the enhancement spot and click **CREATE** (see [Figure 20.29](#)).

Alternately, you can open the package in Object Navigator (Transaction SE80). Right-click the package and select **CREATE • ENHANCEMENT • ENHANCEMENT SPOT**.

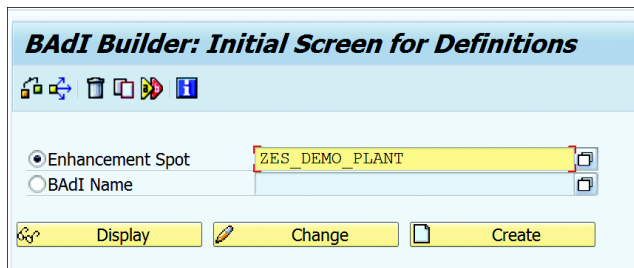


Figure 20.29 BAdI Builder: Initial Screen

2. On the CREATE ENHANCEMENT SPOT screen, as shown in [Figure 20.30](#), provide a short description. In the TECHNOLOGY dropdown box, select BAdI DEFINITION. In the COMPOSITE ENHANCEMENT SPOT field, you can enter the composite enhancement spot name, which simply groups the related enhancement spots. Enhancement spots grouped into a composite enhancement spot are displayed in a tree structure, allowing you to check dependencies and change all related enhancement spots together. This only has a semantic meaning and is similar to grouping related objects into a package.
3. Click the green checkmark (OK) to continue.

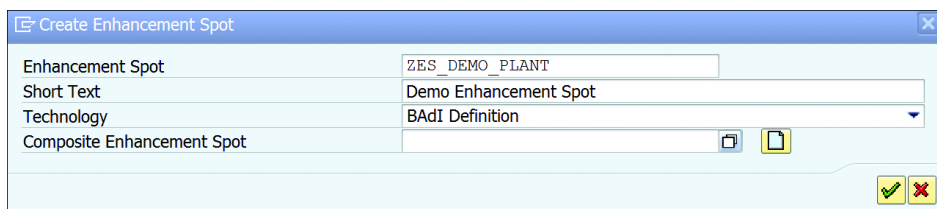


Figure 20.30 Create Enhancement Spot

Creating a BAdI Definition

Once you click the green checkmark on the CREATE ENHANCEMENT SPOT screen shown in [Figure 20.30](#), the system will take you to the ENHANCEMENT SPOT CHANGE screen, which contains four tabs.

From here, proceed with the following steps:

1. Under the ENH. SPOT ELEMENT DEFINITION tab, as shown in [Figure 20.31](#), click the CREATE BAdI button.

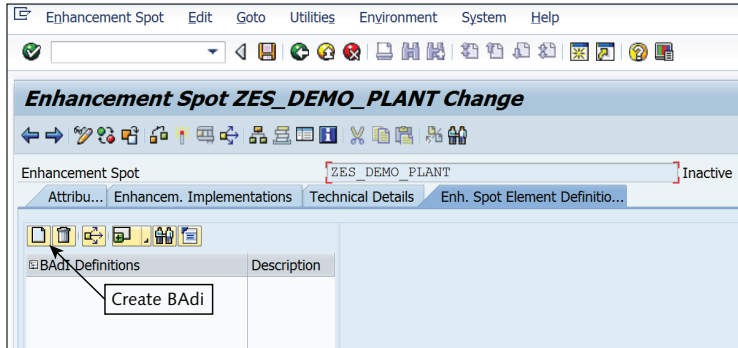


Figure 20.31 Create BAdI

2. This will open a dialog box in which you can provide the NAME for the BAdI definition and a SHORT DESCRIPTION. Click the green checkmark to continue (see [Figure 20.32](#)).

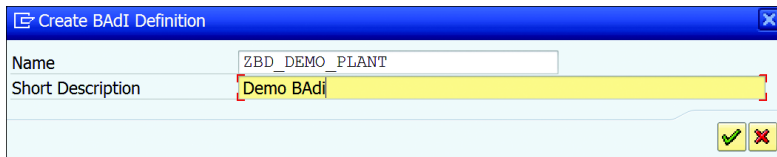


Figure 20.32 Create BAdI Definition

3. On the BAdI definition page, as shown in [Figure 20.33](#), you'll see the BAdI definition with its subobjects in a tree structure. Once the BAdI is selected, you can set certain attributes. Within the ENH. SPOT ELEMENT DEFINITION tab, implement the following changes:
 - ▶ In the USABILITY section, you can select or deselect the MULTIPLE USE checkbox, depending on whether you want to allow multiple implementations for this BAdI to be called or if only a single implementation can be active at any given point in time. As previously mentioned, multiple implementations are useful when you want to provide different logic for different countries, company codes, plants, and so on. The LIMITED FILTER USE checkbox allows you to set a filter criterion for BAdI implementation. You'll learn more about implementing filters in the next subsection. Selecting the AMDP BAdI checkbox creates a special BAdI for ABAP Managed Database Procedures

(AMDP). AMDP is a framework which currently supports database procedures and functions from the SAP HANA database.

- ▶ In the INSTANCE CREATION MODE section, you can select the following:
 - NEWLY CREATING INSTANTIATION: Select this if you don't want the BAdI to buffer the data when it's called multiple times by the program.
 - REUSING INSTANTIATION: Select this if you're fine with reusing data when the BAdI is called multiple times by the program. This option is useful if you're selecting data from the database and want to avoid database access each time the BAdI is called in the program. This implements a singleton pattern in which only one instance exists.
 - CONTEXT-SPECIFIC INSTANTIATION: Selecting this option allows you to specify a context while calling the BAdI; the context controls the BAdI instantiation. If the BAdI is called with the same context, the same BAdI instance is reused; otherwise, a new instance is created. To specify the context, the context class must implement the IF_BADI_CONTEXT interface.
- ▶ The FALLBACK CLASS option allows you to implement a fallback class. A *fallback class* is the default class called if no implementation is active. The fallback class is also called for multiple-use BAdIs with filter criteria when none of the implementations satisfy the filter criteria.

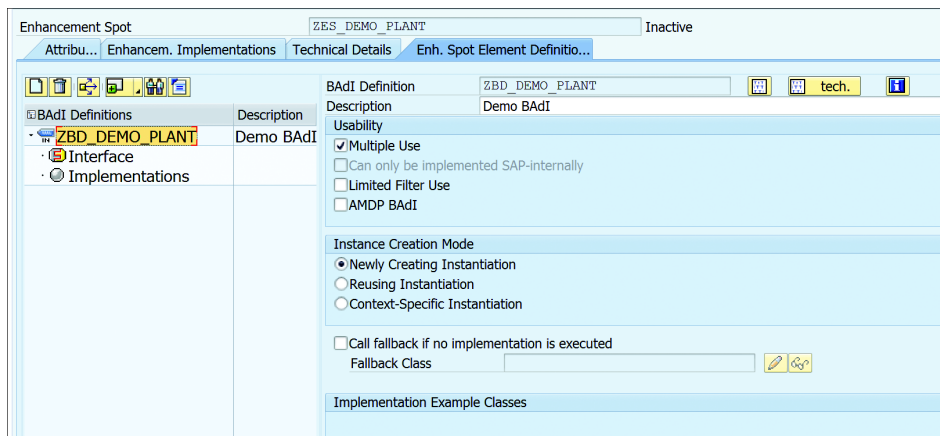


Figure 20.33 Create BAdI Definition

4. You can define a filter for the BAdI by right-clicking on the BAdI definition name and selecting CREATE FILTER from the context menu (see [Figure 20.34](#)).

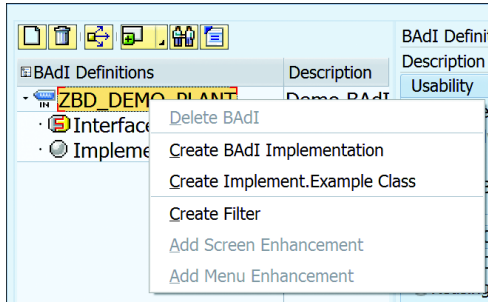


Figure 20.34 Creating Filter

5. Setting a filter adds the field to the parameter interface. The calling program needs to supply this filter value to call the BAdI. The CREATE FILTER screen (see [Figure 20.35](#)) allows you to define the filter field. In this example, set the plant number as the filter.

You can provide the BAdI FILTER name, FILTER TYPE, and DESCRIPTION. For the FILTER TYPE, you can select any elementary type, such as character, numeric, string, packed, and so on, using the **F4** help for the field. The CONSTANT FILTER VALUE checkbox forces the program to pass a constant during the call.

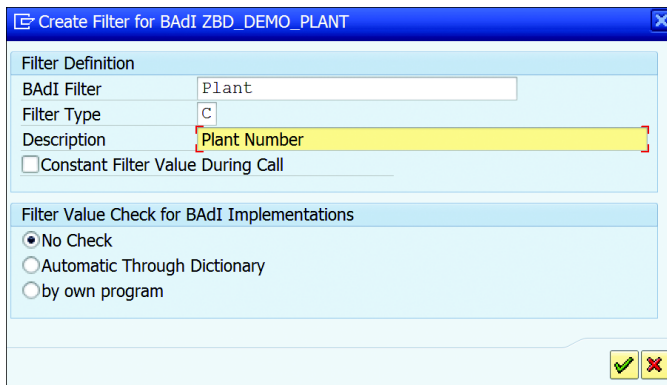


Figure 20.35 Create Filter

6. Because the filter value is passed by the program, it's always a good practice to validate the value being passed in order to avoid any runtime errors. Under FILTER VALUE CHECK FOR BAdI IMPLEMENTATIONS, there are three options:

- ▶ **NO CHECK:** This option doesn't perform any check, and the program is free to pass any value. This can result in runtime errors if incompatible values are passed.
- ▶ **AUTOMATIC THROUGH DICTIONARY:** Selecting this option will show a field in which you can assign a data element (or domain). This allows you to check for type compatibility with the field of the calling program. For example, if the filter is set as TYPE p, then it will raise a syntax error if a field of another type is passed in the calling program. During the implementation of the BADI, this option uses the value range defined for the domain to list the possible values. Only one of the possible values defined in the value range can be set as the filter value (see [Figure 20.36](#)).

Filter Value Check for BADI Implementations	
<input type="radio"/> No Check	Dictionary Object
<input checked="" type="radio"/> Automatic Through Dictionary	Object Name <input type="text" value="werks_d"/>
<input type="radio"/> by own program	<input type="checkbox"/> Object is domain

Figure 20.36 Assigning Data Element as Filter Check

- ▶ **BY OWN PROGRAM:** This option allows you to implement your own check by creating a class, as shown in [Figure 20.37](#). Once you provide the class name and click OK, you'll be taken to Transaction SE24 (Class Builder), where you can implement the class.

Filter Value Check for BADI Implementations	
<input type="radio"/> No Check	Program
<input type="radio"/> Automatic Through Dictionary	Class <input type="text" value="zcl_demo_filter_check"/>
<input checked="" type="radio"/> by own program	Input Length <input type="text" value="4"/>
	Decimal Places <input type="text" value="0"/>

Figure 20.37 Filter Check by Own Program

This class implements an `IF_FILTER_CHECK_AND_F4` interface that has four methods, as shown in [Figure 20.38](#):

- ▶ **SHOW_F4_ICON:** This method has a returning parameter, which you can set as `abap_true(x)` to show the F4 icon to the developer implementing the BADI; this tells the developer that `F4` help is available to select the filter value.

- ▶ F4: This method is similar to using a POV event for dialog programs in which you can write the custom logic to show `F4` values.
- ▶ CHECK: In this method, you can access the value that's passed for the filter and validate it. If the filter value passed is invalid, you raise the `CX_ENH_BADI_FLTR_VALUE_INVALID` exception.
- ▶ INIT: This method is called during initialization, before any of the other methods in this interface are called. In this method, you can access the enhancement spot name, BAdI name, enhancement name, and implementation name.

Method	Level	Visibility	M...	Description
IF_FILTER_CHECK_AND_F4~SHOW_F4_ICON	Instance	Public		Display Input Help Icon?
IF_FILTER_CHECK_AND_F4~F4	Instance	Public		Input Help
IF_FILTER_CHECK_AND_F4~CHECK	Instance	Public		Value Check
IF_FILTER_CHECK_AND_F4~INIT	Instance	Public		Initialization of Instance

Figure 20.38 Filter Class

Once the filter is added, it will be displayed under the FILTER subobject, as shown in [Figure 20.39](#).

BAdI Filter Name	Filter Type	Consta	Description
PLANT	Character-Type	<input type="checkbox"/>	Plant Number

Figure 20.39 Filter

The next step is to define the interface.

Creating Interface

Now that we've created the enhancement spot and BAdI definition, the next step is to create the interface. Proceed with the following steps:

1. Double-click the INTERFACE subobject (see [Figure 20.40](#)), to define an interface for the BAdI. This interface will be used in the implementation class.

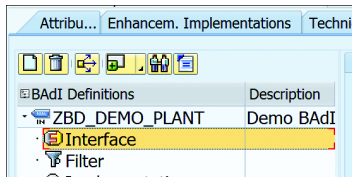


Figure 20.40 Define Interface

2. On the screen now displayed, provide the interface name and click on the pencil icon, as shown in [Figure 20.41](#).
3. This will take you to Class Builder, where you can define methods and other components for the interface.

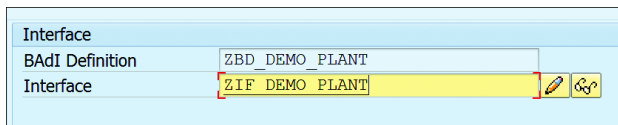


Figure 20.41 Defining Interface

Defining this interface is similar to defining any other regular interface in Class Builder, as discussed in [Chapter 8](#). However, if the `MULTIPLE USE` option is selected for the BAdI, then the methods can have only importing and changing parameters. You can't have exporting or returning parameters, because when multiple implementations are called, they're called in random order, and exporting parameters in such a scenario doesn't make sense. Once the interface is defined, activate it.

This concludes the BAdI definition, so go ahead and activate the enhancement spot by clicking the `ACTIVATE` icon in the toolbar.

20.8.3 Implementing a BAdI

Now that you understand how to define a BAdI, let's look at implementation: To implement a BAdI, we create an enhancement implementation and then the BAdI implementation. For the BAdI implementation, we create an implementation

class that inherits the methods from the interface class defined for the BAdI definition.

Proceed with the following steps:

1. Implement the BAdI either using Transaction SE19 or simply by right-clicking the BAdI definition name and selecting CREATE BADI IMPLEMENTATION (see [Figure 20.34](#)).
2. This will open a dialog box in which you can enter the ENHANCEMENT IMPLEMENTATION name and SHORT TEXT, as shown in [Figure 20.42](#). Similar to how you can create multiple BAdI definitions within an enhancement spot, you can create multiple implementations within an enhancement implementation.

Figure 20.42 Enhancement Implementation

3. Once you input the necessary settings, click the green checkmark. This will immediately open the CREATE BADI IMPLEMENTATION dialog box, in which you can enter the BADI IMPLEMENTATION name, DESCRIPTION, and IMPLEMENTING CLASS name (see [Figure 20.43](#)).

Figure 20.43 Create BAdI Implementation

4. Once the details are entered, click the green checkmark to go to the ENH. IMPLEMENTATION ELEMENTS tab, as shown in [Figure 20.44](#). Double-click the IMPLEMENTING CLASS subobject to see the methods of the interface. Double-click the method to implement it. The system will automatically open the method in Transaction SE24, where you can write the code like you would for any regular method.

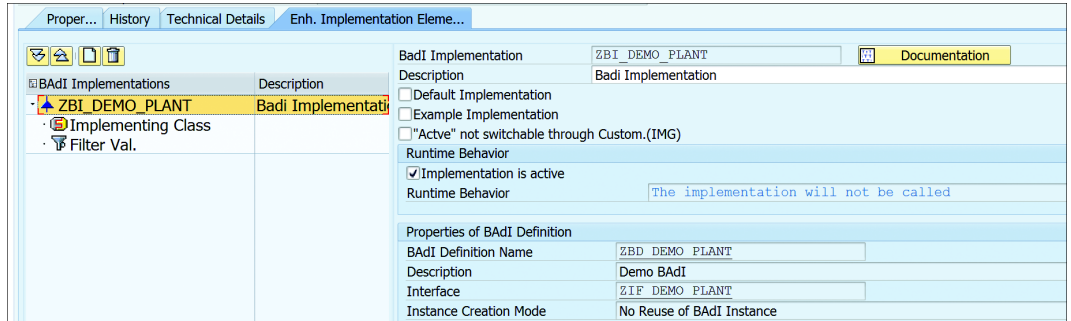


Figure 20.44 Enhancement Implementation

- Once the method is implemented, activate the method and come back to the ENHANCEMENT IMPLEMENTATION screen. At this point, you should see a message indicating that the implementation will be called (see [Figure 20.45](#)).

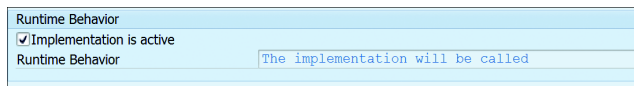


Figure 20.45 Implementation Is Active

- Double-click the FILTER VAL. subobject to set the filter criteria. This will open a screen in which you can assign filter values. Click the COMBINATION button, as shown in [Figure 20.46](#), and double-click the question mark symbols to set the filter value.

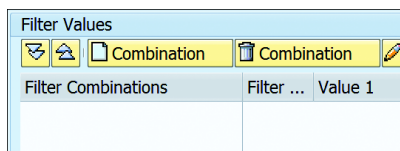


Figure 20.46 Setting Filter Criteria

- Once the filter value is set, the system will display the filter (see [Figure 20.47](#)). You can click on the AND or OR button to add multiple combinations.

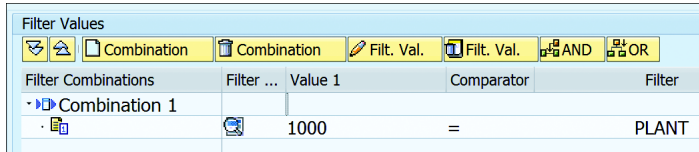


Figure 20.47 Filter Values

20.8.4 Implementing a Fallback Class

When you define a single-use BAdI, it will result in an error if the BAdI is called and no implementations are active. In such scenarios, you can implement a fallback class for the BAdI, which will automatically be called if there are no active implementations. A fallback class is also useful for calling multiuse BAdIs for which the filter criteria aren't satisfied for any of the active implementations. In such cases, the fallback class is called automatically.

To implement a fallback class, proceed with the following steps:

1. Double-click the BAdI definition in the enhancement spot. Select the **CALL FALLBACK IF NO IMPLEMENTATION IS EXECUTED** checkbox to enable the **FALLBACK CLASS** field, in which you can input the class name and click the pencil icon to create the class (see [Figure 20.48](#)).

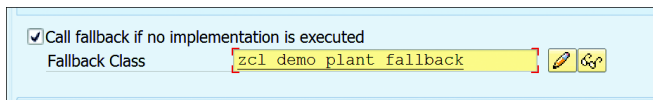


Figure 20.48 Fallback Class

2. After you click the pencil icon, you'll be asked if the fallback class should be used as an example class (see [Figure 20.49](#)). This will allow other developers to understand how the methods should be implemented. If you want the fallback class to be used as an example, click **YES**; this will take you to Transaction SE24 to implement the class.

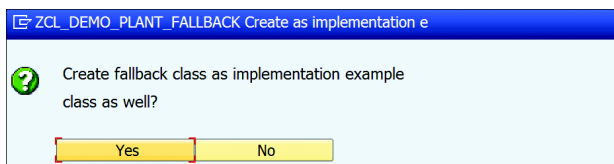


Figure 20.49 Fallback Class as Example Implementation

3. When using the fallback class as an example class, you may want to allow the fallback class to be inherited by other classes for ease of implementation. In such a case, make sure to uncheck the FINAL class flag, as shown in [Figure 20.50](#).

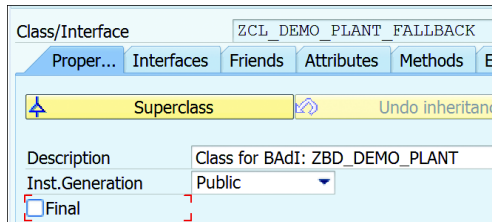


Figure 20.50 Allowing Inheritance

Now that we've defined and implemented a BAdI, let's look at how the BAdI can be called in the program.

20.8.5 Calling a BAdI

To call a BAdI, we use the `GET BADI` statement, which returns the instance of the BAdI implementation. If the filter criteria are set for the BAdI, we pass the filter values to the `GET BADI` statement.

Once the `GET BADI` statement instantiates the BAdI object with the implementation reference, we can call the methods using the `CALL BADI` statement, as shown in [Listing 20.2](#).

```
DATA lo_badi TYPE REF TO ZBD_DEMO_PLANT.
TRY.
  GET BADI lo_badi
  FILTERS
  plant = lv_werks. "1000
  CALL BADI badi->handle_plant
  EXPORTING
  werks = lv_werks.
  "This exception is only for single-use BAdI
  CATCH cx_badi_not_implemented.
ENDTRY.
```

Listing 20.2 Calling BAdI

In [Listing 20.2](#), we're defining a reference object, `lo_badi`, a type of BAdI definition. We're using the `GET BADI` statement to get the reference to the active

implementation of the BAdI. Because the BAdI uses plant number as the filter criterion, we're passing the filter value with the `GET BADI` statement using the `FILTERS` addition. Once we have the reference to the implementation, we can access its methods by using the `CALL BADI` statement.

As you can see, by implementing the BAdI we can avoid using the `CASE` statement from [Listing 20.1](#) to handle plant-specific logic in the main program. Each plant can have its own implementation, which can be changed without changing the main program. If new plants are added, we can simply add new implementations to the BAdI.

Good object-oriented design calls for the separation of tasks. By implementing the BAdI, we separated the task of finding the plant and running the tasks specific to the plant. The program's job here is to find the plant, and the BAdI's job is to run plant-specific tasks. The program doesn't need to know what tasks are being performed for the plant, and the BAdI doesn't need to know how the program is determining the logic for each plant.

20.9 Enhancement Points

One major challenge with the enhancement concepts that we've discussed so far is that they're called at predefined locations in standard SAP programs and have fixed interfaces. They're sufficient to meet the requirements of customers most of the time, but they do have several limitations for customizing standard SAP applications. The whole concept behind these enhancements relies on SAP anticipating how customers might want to enhance standard SAP programs and correctly deciding where to provide the opening in the code.

The *Enhancement Framework* was introduced in SAP ERP 6.0. This framework brought the concept of enhancing SAP applications to a different level with the introduction of enhancement points.

Enhancement points allow you to bend standard SAP applications as you wish. Enhancement points are source code plugins added to the source code of the standard SAP programs directly. These source code plugins are compiled with the main program and are as good as writing the code directly in the main program.

There are two types of enhancements available: explicit and implicit. In the following subsections, we'll discuss each type.

20.9.1 Explicit Enhancements

Explicit enhancements are similar to all the other types of enhancements, such as customer exits and BADIs, in that they're provided explicitly in the code. If an enhancement isn't provided explicitly by SAP, you can't add your custom code.

There are two types of explicit enhancement points:

► **Static explicit enhancement points**

Static enhancement points are introduced by the `ENHANCEMENT-POINT` statement and end with the `STATIC` keyword, as shown in [Figure 20.51](#). Static enhancement points aren't controlled by the Switch Framework.

► **Dynamic explicit enhancement points**

Dynamic enhancement points don't include the `STATIC` keyword at the end. Dynamic enhancement points are controlled by the Switch Framework.

We use static enhancement points to add data declarations and dynamic enhancement points to write program logic. The enhancements are implemented between `ENHANCEMENT` and `ENDENHANCEMENT` statements, with each enhancement assigned a number that's unique within the program.

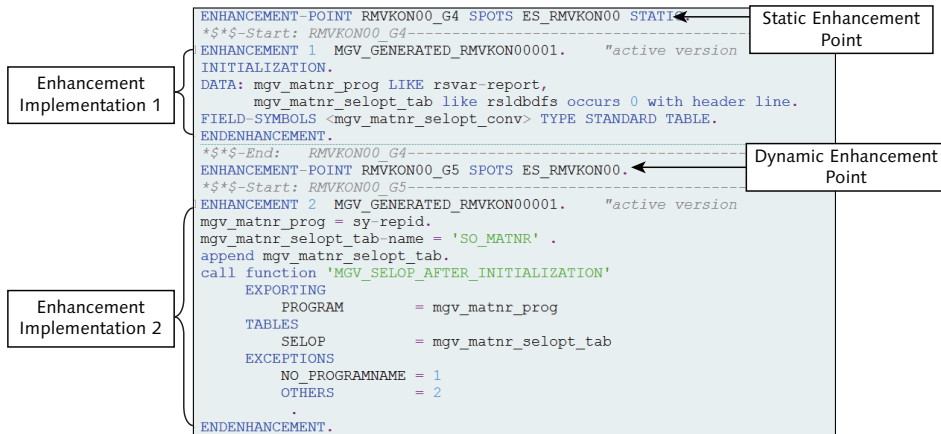


Figure 20.51 Explicit Enhancements

If you see the `ENHANCEMENT-POINT` statement in a standard SAP program, then you can add your source code plugin at that point in the source code. Implementing an enhancement is pretty straightforward.

Proceed as follows:

1. Open the standard SAP program that you want to enhance in display mode. Click on the seashell icon in the application toolbar, as shown in [Figure 20.52](#). This will enable the program to change or implement enhancements.

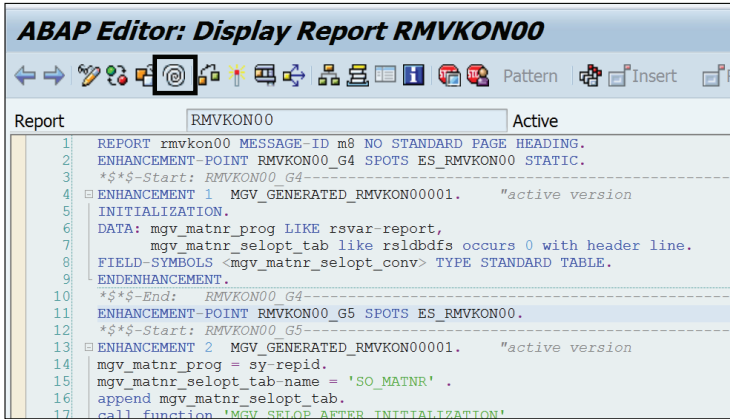


Figure 20.52 Enhancement Button

2. Right-click the enhancement point and select ENHANCEMENTS • CREATE IMPLEMENTATION from the context menu. This will open the SELECT OR CREATE ENHANCEMENT IMPLEMENTATION window (see [Figure 20.53](#)). Here, you can select any existing enhancement implementation or click the CREATE button to create a new implementation.

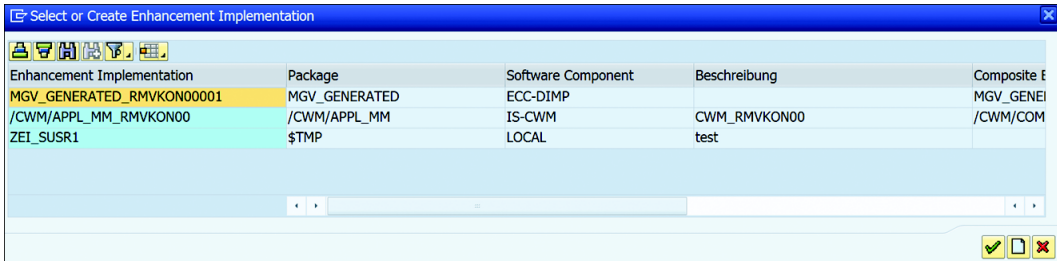


Figure 20.53 Selecting or Creating Implementation

3. If you click the CREATE button to create a new implementation, the system will present the CREATE ENHANCEMENT IMPLEMENTATION window, in which you can enter an ENHANCEMENT IMPLEMENTATION name and SHORT TEXT description for

your implementation, as shown in [Figure 20.54](#). Save the enhancement implementation to your customer package. The new enhancement implementation now will appear in the list shown in [Figure 20.53](#). Double-click the enhancement implementation name to select it.

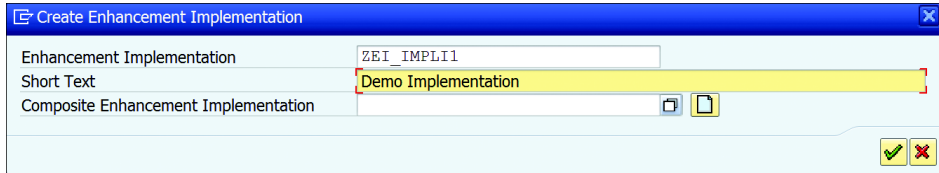


Figure 20.54 Creating New Implementation

4. This will create an `ENHANCEMENT ENDENHANCEMENT` section in the code in which you can add your custom code, as shown in [Figure 20.55](#).

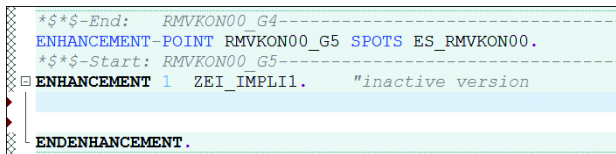


Figure 20.55 Enhancement Section

5. Once you add custom code, click on the `ACTIVATE ENHANCEMENTS` button in the application toolbar.

20.9.2 Implicit Enhancements

Unlike explicit enhancements that are provided in programs by SAP, implicit enhancements are available by default in the program. They are available at the beginning and end of a procedure (i.e., in subroutines, function modules, and methods), at the end of a structure definition, and at the end of an executable program.

To show implicit enhancements, follow the menu path `EDIT • ENHANCEMENT OPERATIONS • SHOW IMPLICIT ENHANCEMENT OPERATIONS` from ABAP Editor, Function Builder, or Class Builder.

This will show a comment line under the `SOURCE CODE` tab that indicates where you can create the implementation, as shown in [Figure 20.56](#).

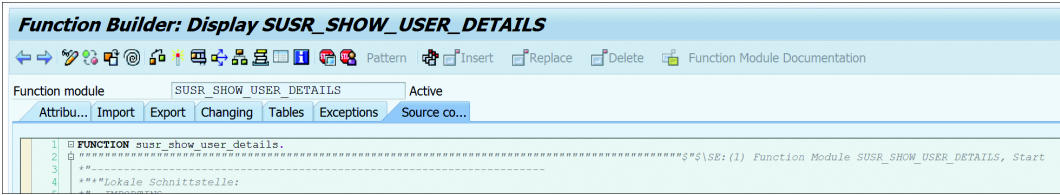


Figure 20.56 Implicit Enhancement Point

Even though you have more flexibility with implicit enhancement points, you still can't use them to, for example, add code in the middle of a procedure.

Implementing implicit enhancements is similar to explicit enhancements, in that you create an enhancement implementation. We recommend keeping all related enhancements in the same implementation so that they can be switched on or off together. When you switch off an enhancement implementation, it won't be compiled.

Let's walk through how to enhance a function module interface to add custom parameters:

1. Open the function module you want to enhance and navigate to the **IMPORT**, **EXPORT**, **CHANGING**, or **TABLES** tab, depending on which tab you want to add the custom parameter to. For this example, go to the **EXPORT** tab. The process is the same for other parameters.
2. Click on the **ENHANCE** icon (see [Figure 20.57](#)) in the toolbar. The system will prompt you to create an enhancement implementation. Create an enhancement implementation as explained in [Section 20.9.1](#).

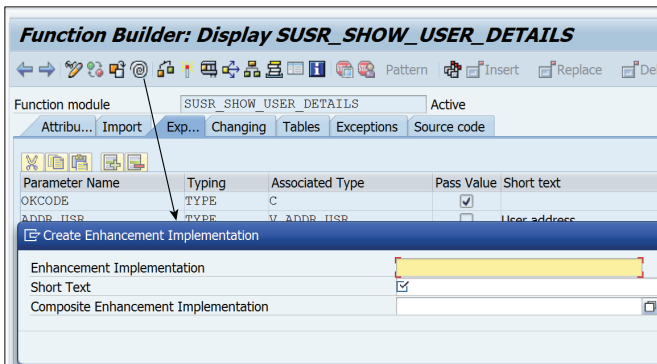


Figure 20.57 Create Enhancement

3. Add the custom field to the interface, as shown in [Figure 20.58](#). If you're adding import, changing, or tables parameters, make sure to check the `OPTIONAL` checkbox to ensure that your enhancement won't break the existing programs calling this function module.

As you can see in [Figure 20.58](#), the existing parameters of the function module are input-disabled. You can also see the enhancement implementation name in the `ENHANCEMENT IMPLEMENTATION` column.

Parameter Name	Typing	Associated Type	Pass Value	Short text	Long Text	Enhancement Implementation
OKCODE	TYPE	C	<input checked="" type="checkbox"/>			Create
ADDR_USER	TYPE	V_ADDR_USER	<input type="checkbox"/>	User address		Create
EX_USER	TYPE	SYST-UNAME	<input type="checkbox"/>	ABAP-Systemfeld: Name des aktuellen Ben		Create ZEI_IMPLI3

Figure 20.58 Adding Custom Parameter to Export Tab

- The parameter added to the interface should be filled in the source code of the function module. Navigate to the `SOURCE CODE` tab. If you don't see a suitable explicit enhancement point in the code, you can use the implicit enhancement point to update the parameter, as explained in the following steps.
- Use the menu path `EDIT • ENHANCEMENT OPERATIONS • SHOW IMPLICIT ENHANCEMENT OPERATIONS` to display the implicit enhancement points.
- Select the implicit enhancement point and follow the menu path `EDIT • ENHANCEMENT OPERATIONS • CREATE IMPLEMENTATION`, as shown in [Figure 20.59](#).

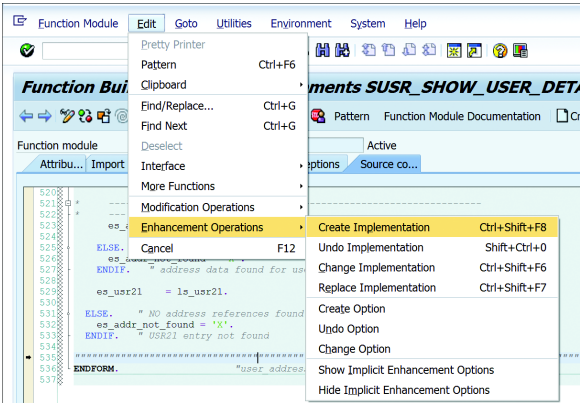


Figure 20.59 Creating Implementation

7. You'll see a dialog box to choose the enhancement mode (see [Figure 20.60](#)). Choose the DECLARATION button if you want to add data declarations (static enhancement), and choose the CODE button if you want to add program logic (dynamic enhancement). For this example, select the CODE button.

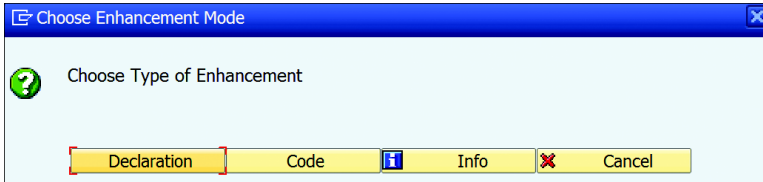


Figure 20.60 Enhancement Mode

8. Choose an existing implementation if the code you're adding is related to another enhancement; otherwise, create a new enhancement implementation. Add the program logic between `ENHANCEMENT` and `ENDENHANCEMENT`, as shown in [Figure 20.61](#). Within the source code of the function module, you have access to both the existing parameters and the new parameter that you added. You can also access the global data objects of the function group, but use caution when changing global data of the function group.

```

ENDFUN.
*****
*$$-Start: 9999-----
ENHANCEMENT 2 ZEI_IMPLI21.      "inactive version
ex_user = sy-uname.
ENDENHANCEMENT.
*$$-End: 9999-----
ENDFUNCTION.

```

Figure 20.61 Implementing Enhancement

The steps to enhance a method are similar to the steps to enhance a function module. You can also enhance a class to add new methods. The bonus is that these new methods can also access the private components of a class.

20.10 Business Transaction Events

Business transaction events (BTEs) are a type of enhancement specifically available for the SAP ERP Financial Accounting component. BTEs are similar to user exits in

that you can only make program changes, as opposed to customer exits and BAdIs that support screen and GUI enhancements.

BTEs are implemented using function modules, unlike BAdIs, which use ABAP Objects. BTEs preceded BAdIs and are an older enhancement technique, but we're covering them here for the sake of completeness.

BTEs work on the principle that the system triggers an event for which a suitable function module is maintained to execute custom code. SAP provides sample function modules for all events (called *samples*), and we can copy these sample function modules and write our custom code. The interface of these function modules shouldn't be changed.

There are two types of interfaces available:

► **Publish and subscribe interfaces**

These interfaces allow you to interrupt standard SAP application flow to carry out additional work outside of the application. These interfaces don't export any data back to the application, so you can't use them to modify the application data. They import certain application data, which can be used to perform additional steps external to the application. For example, when creating a vendor, you may want to update certain custom tables with additional data using the vendor number.

► **Process interfaces**

These interfaces allow you to interrupt the standard SAP application and modify its data. They allow you to change how a standard SAP application behaves and influence the application data, and they implement importing and changing parameters, which can be used to modify the application data.

Let's use a vendor master record as an example to explain how BTEs are implemented. For this example, we want to execute some custom code when a vendor master record is saved via Transaction FK02.

20.10.1 Implementing a BTE

BTEs are implemented using Transaction FIBF. Most of the configurations are carried out by functional consultants; as ABAP developers, our role is limited to creating the function module and adding custom code.

The following steps walk you through implementing a BTE:

1. On the initial screen of Transaction FIBF, follow the menu path ENVIRONMENT • INFO SYSTEMS (P/S) to find a suitable interface, as shown in [Figure 20.62](#).

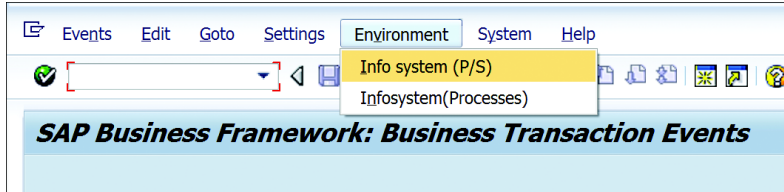


Figure 20.62 Menu to Locate Interface

2. On the selection screen for publish and subscribe interfaces, enter "A" (for *application component*) in the ATTRIBUTE TYPE field and click EXECUTE (see [Figure 20.63](#)).

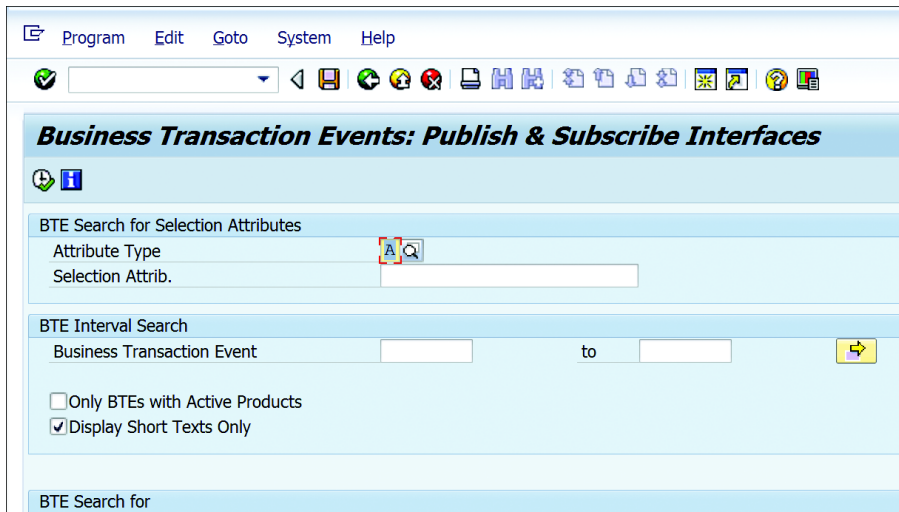


Figure 20.63 Search Publish and Subscribe Interfaces

3. The program will list all the available events. Search for the event you want to implement. For this example, implement event 00001420, as shown in [Figure 20.64](#).

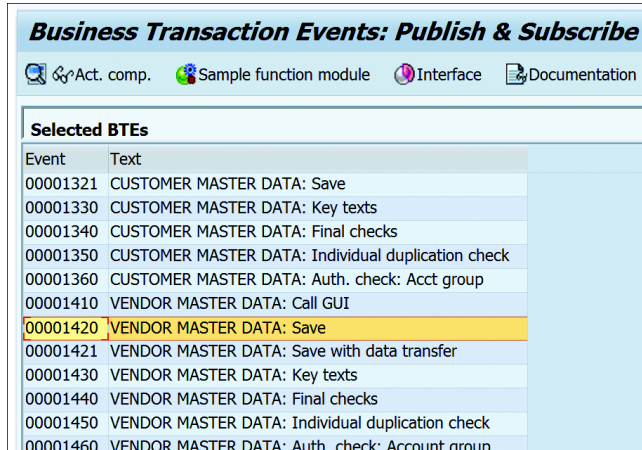


Figure 20.64 BTE Events

4. You can select the event and click DOCUMENTATION to learn more about it. Similarly, click the INTERFACE icon to check the interface of the corresponding function module. The function module called for the event should implement this interface.
5. Click the SAMPLE FUNCTION MODULE icon to open the sample function module in Function Builder. You can copy this function module into the customer namespace by using the copy function in Function Builder, as shown in [Figure 20.65](#). The sample function module provided for this event is SAMPLE_INTERFACE_00001420.

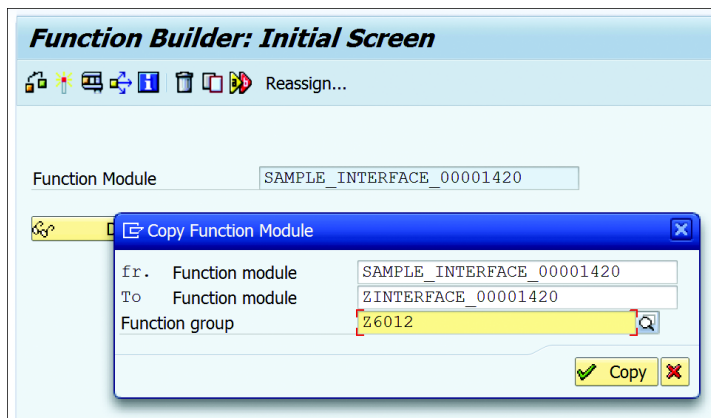


Figure 20.65 Copy Sample Function Module

6. You can maintain the custom logic in this copied function module. For this example, simply hard-code a breakpoint (see [Figure 20.66](#)) to test if the function module is called when the vendor master record is changed. The function module interface shouldn't be changed.

```

1 FUNCTION ZINTERFACE_00001420.
2
3  *** Local Interface:
4  *** IMPORTING
5  ***   VALUE(I_LIFNR) LIKE LFA1-LIFNR
6  ***   VALUE(I_BUKRS) LIKE LFB1-BUKRS OPTIONAL
7  ***   VALUE(I_AKTYP) LIKE T020-AKTYP OPTIONAL
8  ***   VALUE(I_XVBUP) LIKE OFIWA-XVBUP DEFAULT 'X'
9  ***
10
11  BREAK-POINT.
12
13
14
15 ENDFUNCTION.
    
```

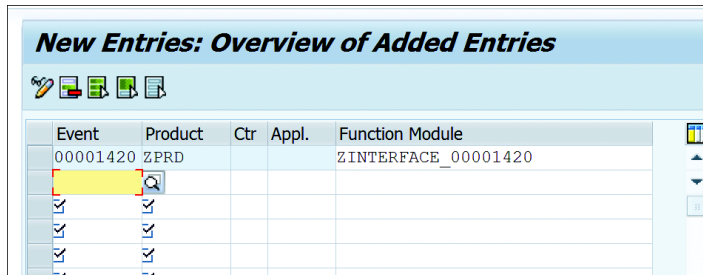
Figure 20.66 Copied Function Module

- Once the function module is maintained, we need to map it to a product. To do so, first create a product and then map it to the function module. From the initial screen of Transaction FIBF, select the menu path **SETTINGS • PRODUCTS • ..OF A CUSTOMER**.
- On the **CUSTOMER PRODUCTS** screen, click on **NEW ENTRIES** and add a product, as shown in [Figure 20.67](#). You can optionally provide an RFC destination. Click the checkbox in the **A** column to mark the product as active. Once the product is added, save it.

Product	Text	RFC destination	A
zprd	Sample Product		<input checked="" type="checkbox"/>
			<input type="checkbox"/>
			<input type="checkbox"/>

Figure 20.67 Add Product

9. Now, link the event and the product with the function module that you created. Link the function module, and from the initial screen of Transaction FIBF, follow the menu path **SETTINGS • P/S MODULES • ..OF A CUSTOMER**. On the **PUBLISH AND SUBSCRIBE** screen, click the **NEW ENTRIES** button and provide the details shown in [Figure 20.68](#). You can use the **F4** help to select the event and product.
10. Save the record.



Event	Product	Ctr	Appl.	Function Module
00001420	ZPRD			ZINTERFACE_00001420

Figure 20.68 Linking Event with Function Module

20.10.2 Testing a Custom Function Module

Let's test if the custom function module is called when the vendor master record is saved. Proceed with the following steps:

1. Open any vendor record in Transaction FK02, edit the address of the vendor, and click **SAVE**.
2. This should trigger the function module in debug mode because you hard-coded a breakpoint in the function module (see [Figure 20.69](#)). Also, the vendor number is passed to the function module.

We successfully interrupted the standard SAP application to execute custom code before saving the vendor record. Finding a suitable event for the customization is generally the task of a functional consultant; our role in implementing BTEs is limited to function module development.

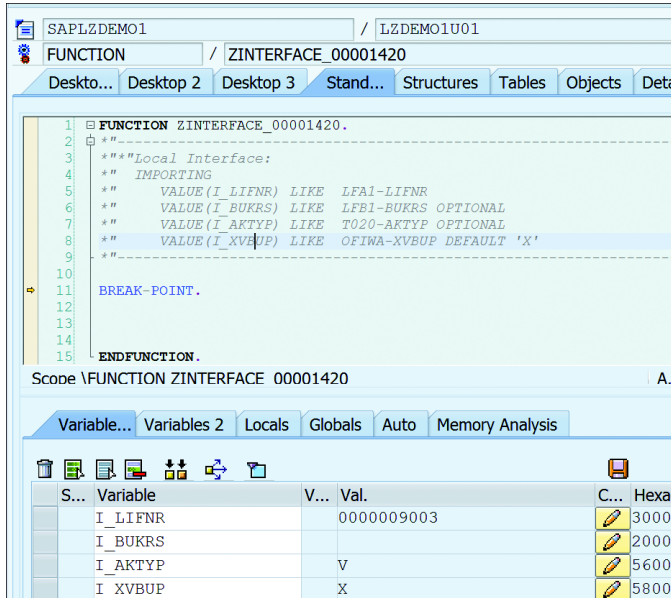


Figure 20.69 Function Module in Debug Mode

20.11 Summary

Enhancements are vital for any company that implements SAP. Enhancements allow a company to customize standard SAP per its needs. In this chapter, we first looked at Modification Assistant, which allows you to modify various repository objects. We then discussed all the various enhancement techniques, both old and new. Understanding enhancement techniques is important; you may need to not only implement new enhancements but also modify old enhancements. This chapter should give you enough knowledge and confidence to work with different enhancement techniques. Because business processes evolve continuously, the need for enhancements will be a constant battle.

Enhancements should be implemented with care, because many applications are closely integrated. Implementing an enhancement for one application may cause problems in another application. Enhancements should always be tested thoroughly and should go through integrated testing before moving to production.

In the next chapter, we'll discuss various tools provided by SAP for testing and analyzing ABAP programs.

This chapter discusses native ABAP tools used to test and analyze ABAP programs.

21 Test and Analysis Tools

Testing software applications for bugs and analyzing their performance is an important part of the software development lifecycle. As programmers, we always aim to write code that is bug-free and performs optimally. However, no matter how experienced you are as a programmer, bugs do creep into the code. Sometimes, you may make mistakes inadvertently, and the chances of bugs and impacts on performance increase as more people become involved in the development process.

To ensure that applications perform as expected, every company invests in *quality assurance*. The need for a well-rounded process to test applications for productive use can't be stressed enough. An untested application not only runs the risk of unexpected behavior, but also can cause a company monetary loss. For example, if an application that generates an invoice applies the wrong discount under certain specific conditions due to a bug, it can cause all sorts of troubles.

Similarly, if an application uses too many resources due to a bad `SELECT` query, it may not always be possible to catch it in the development system due to a lack of test data. If such an application is moved to the production environment, it may bring down the system or affect the performance of other applications.

To ensure applications perform optimally and as expected, SAP provides various tools that can be used to test and analyze ABAP programs. Most of these tools are targeted to be used by ABAP developers, quality managers, and system administrators. In this chapter, we'll look at some of the important tools available in our arsenal for testing and analyzing ABAP code.

In [Section 21.1](#), we will provide an overview of all the tools we'll be discussing throughout the chapter.

21.1 Overview of Tools

Let's look at all the various tools that can be used to check ABAP programs. Some tools allow you to identify potential problems, and others help trace definitive problems.

We'll discuss the following tools in this chapter:

- ▶ **ABAP Unit**

ABAP Unit is a test-driven development (TDD) tool used to test an application as it's being developed and if any new changes to the program affect the original functionality. We'll discuss ABAP Unit further in [Section 21.2](#).

- ▶ **Code Inspector**

Code Inspector allows you to perform a static check on your program to identify any problem areas in the code. This is probably the first tool we run to check our code statically to identify any potential problems. Unlike other runtime tools that check only the code that's executed at runtime, Code Inspector checks the complete code of the program.

Code Inspector is integrated into ABAP Workbench and can be accessed from various workbench tools, such as ABAP Editor, Function Builder, and so on, to run ad hoc tests. It can also be run separately via Transaction SCI. We'll provide further details of the Code Inspector in [Section 21.3](#).

- ▶ **Selectivity analysis**

The *selectivity analysis* tool (Transaction DB05) allows you to perform an analysis of a database table to identify its size and data distribution. This tool helps analyze the memory requirements for table buffering and index design. More details on this tool can be found in [Section 21.4](#).

- ▶ **Process analysis**

The *process analysis* tool (Transactions SM50 and SM66) is mostly used by system administrators to check the work process overview. However, this tool can also be used to check the performance of programs that take longer durations to execute. This tool is also useful for checking programs that run in the background, and it provides information about currently running programs. We'll provide a thorough overview of this tool in [Section 21.5](#).

- ▶ **Memory Inspector**

Memory Inspector is part of the ABAP Debugger. The ABAP Debugger is used to analyze the memory occupied by the data objects. By performing a memory

analysis, you can identify, for example, large internal tables that may negatively impact the performance. From within the debugger, we can create memory snapshots that can be further analyzed using Memory Inspector (Transaction S_MEMORY_INSPECTOR). More information on the Memory Inspector can be found in [Section 21.6](#).

► **Table call statistics**

The *table call statistics* tool (Transaction ST10) allows you to analyze the call statistics for database tables. This tool is used by system administrators to analyze the tables that are buffered and check the efficiency of buffer settings. The administrator can also analyze unbuffered tables to check if they're technically suitable for buffering. For more information on this tool, see [Section 21.7](#).

► **Performance trace**

A *performance trace* (Transaction ST05) is one of the most important and most frequently used analysis tools. Performance traces consist of a SQL trace, RFC trace, enqueue trace, and table buffer trace. Of all the traces, we most commonly perform a SQL trace to identify how long SQL statements are taking to execute.

Not all performance-related issues will be due to database access. However, database access often is the cause of sluggish ABAP program performance. In such cases, analyzing and optimizing SQL queries greatly improves performance. We'll provide further information on this trace tool in [Section 21.8](#).

► **ABAP Trace/runtime analysis**

ABAP Trace, also called *runtime analysis* (Transaction SAT), helps to analyze the performance of ABAP programs. This tool is primarily used to analyze performance problems at the application server level. You can identify such things as issues with CPU usage or modularization units that are called unnecessarily during program execution. More information on ABAP Trace can be found in [Section 21.9](#).

► **Single-transaction analysis**

Single-transaction analysis (Transaction ST12) is similar to runtime analysis, but it provides certain additional options to analyze a trace.

For more details on single-transaction analysis, see [Section 21.10](#).

► **Dump analysis**

When a program terminates due to an exception, a short dump is generated.

Dump analysis (Transaction ST22) allows you to check the short dumps generated by the programs.

The last section of this chapter, [Section 21.11](#), provides further details on dump analysis.

All these tools are used at different stages in the software development lifecycle. Some of the tools are used in the development system, and others are used in the quality system. Performance analysis tools shouldn't be used in production environments, except in very rare and extreme situations in which the dataset for testing isn't available in the quality system. For example, Code Inspector is always used in the development system, whereas Memory Inspector is generally used in the quality assurance system because it needs data for analysis.

When analyzing a program, Code Inspector is used before executing the program; almost all other tools are used while the program is running, because they need to analyze the data that's processed or analyze execution of various commands.

Now, let's explore each tool in the following sections.

21.2 ABAP Unit

One of the key aspects of software development is testing. *Testing* allows you to ensure a software application performs as expected and to identify any issues at an early stage, before the code enters productive use.

The traditional way of testing is to first develop the application and then perform some basic tests in the development system before the code is moved to the quality system for further testing. Sometimes, due to a lack of time, aspects of the application behavior aren't tested for specific scenarios, resulting in costly errors. Irrespective of how seasoned a developer is, mistakes are bound to creep into the application development; one way to ensure that each aspect of the application is tested is to follow the *test-driven development* (TDD) approach.

TDD turns testing on its head and requires that you first write tests before the application is developed, rather than testing after the application is developed. The tool for implementing the TDD approach in ABAP programs is called *ABAP Unit*.

The idea behind creating tests first using ABAP Unit is to help you test each aspect of the application as it's developed and, more importantly, test if any new changes to the program affect the original functionality. Once ABAP Unit establishes the test for the program, you can test the program anytime using the menu path PROGRAM • EXECUTE • UNIT TESTS (see [Figure 21.1](#)).

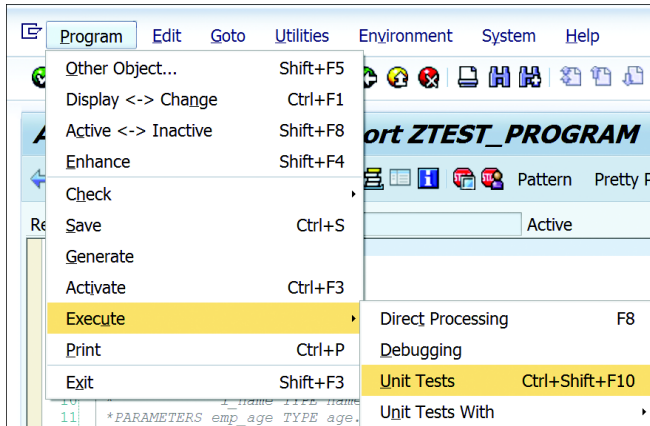


Figure 21.1 Execute Unit Tests

Using a TDD approach involves three main steps:

1. Eliminating dependencies in the program
2. Implementing mock objects
3. Writing and implementing test classes

21.2.1 Eliminating Dependencies

For unit tests to work, the program under the test shouldn't have dependencies—for example, dependencies on user input or a dependency on the database to fetch the data.

To explain this, let's consider a regular ABAP program that calls various function modules to validate and update data. For example, the program calls one function module to check user authorizations, one function module to ask confirmation from the user, and so on.

[Listing 21.1](#) shows some typical ABAP code for implementing this example.

```

FORM update.
*Check user authorization
CALL FUNCTION 'ZUSER_SOME_AUTH'
EXPORTING
user = sy-uname
IMPORTING
auth = lv_auth.
*Business logic
IF lv_auth EQ abap_true.
*Prepare for update
ELSEIF lv_auth EQ abap_false.
*Do something else
ENDIF.
*Ask user for confirmation
CALL FUNCTION 'POPUP_TO_CONFIRM'
*Business logic
CASE user_response.
WHEN 'Y'.
  "Update data
WHEN 'N'.
  "Cancel
WHEN others.
RETURN.
ENDCASE.
ENDFORM.

```

Listing 21.1 Typical ABAP Logic

Listing 21.1 implements the logic to check user authorizations and updates the data in the database accordingly. In this program, you want to test how the program behaves when the authorization fails or when the user doesn't confirm the data update. However, with the way the code is implemented, you need an actual user to check the authorizations or to respond to the confirmation pop-up.

These are the examples of *dependencies*; in this case, the program is depending on the user to provide inputs. Similarly, if the program reads a database table to check some settings, then we can say the program is dependent on the availability of actual data in the database to perform a test. We can't implement TDD for a program that has such dependencies.

To implement TDD, we need to remove the dependencies of the program, such as the dependency on the user or database in our example. One way to do so is to follow the *separation of concerns* approach to development in which you don't burden one program with carrying out all the activities; instead, you implement different classes to carry out different activities. In our example, instead of

burdening the program with validating the user authorizations, validating the user response, updating the data in the database, and so on, we can implement a different class for each of these activities. This way, we ensure that each class does only one job and does it well.

Listing 21.2 implements classes instead of the function modules used in Listing 21.1. The code in Listing 21.2 does exactly the same thing as Listing 21.1, but in Listing 21.2 we're using classes to handle dependencies instead of function modules or subroutines. By replacing function modules or subroutines with classes to handle dependencies, we can implement mock objects that can simulate external dependencies.

```
FORM update.
*Check user authorization
lv_auth = lo_auth->check_auth( sy-uname ).
*Business logic
IF lv_auth EQ abap_true.
*Prepare for update
ELSEIF lv_auth EQ abap_false.
*Do something else
ENDIF.
*Ask user for confirmation
user_response = lo_user->popup_to_confirm( ).
*Business logic
CASE user_response.
WHEN 'Y'.
  "Update data
WHEN 'N'.
  "Cancel
WHEN others.
RETURN.
ENDCASE.
ENDFORM.
```

Listing 21.2 Using Classes for Different Tasks

21.2.2 Implementing Mock Objects

Once the dependencies of the program are isolated into their own classes, you can change your ABAP programs to take advantage of the ABAP Unit framework. To make use of ABAP Unit, we need to implement mock objects that appear to do the same thing as real objects, such as database access or a user response. We also need to ensure that the class under test uses these mock objects instead of real objects when the test is underway; this is known as *injection*.

To implement a mock object, you can take advantage of OOP by inheriting the real objects and redefining the methods to return dummy values. For example, if the method of the real object returns data from the database, you can redefine the method in the mock object to return some hard-coded values.

21.2.3 Writing and Implementing Unit Tests

To perform a unit test, we need to implement test classes. *Test classes* can be defined locally in the program or globally in Class Builder (Transaction SE24). To define a test class, we use the `FOR TESTING` addition with the `CLASS DEFINITION` statement. While defining a test class, we need to set certain properties, such as `RISK LEVEL` and `DURATION`. The system will use these properties to determine if the test class should be executed.

[Listing 21.3](#) shows the syntax to define a test class.

```
CLASS lcl_test DEFINITION FOR TESTING
                        RISK LEVEL HARMLESS
                        DURATION SHORT.
ENDCLASS.
```

Listing 21.3 Define Test Class

For `RISK LEVEL`, you can set the property as `HARMLESS`, `DANGEROUS`, or `CRITICAL`. The `HARMLESS` property indicates that executing the test class doesn't affect any customizations or perform any database changes. The `DANGEROUS` property implies that the test class performs database changes. The `CRITICAL` property implies that the test class performs changes to customization and the database.

For `DURATION`, you can set the property as `SHORT`, `MEDIUM`, or `LONG`. The `SHORT` property implies the test class runs fast and usually takes less than a minute to execute. The `MEDIUM` property implies that the test class takes anywhere from one to five minutes. The `LONG` property implies that the test class may take up to one hour to execute.

The `RISK LEVEL` and `DURATION` properties use client settings maintained via Transaction `SAUNIT_CLIENT_SETUP`, as shown in [Figure 21.2](#).

The test class should implement at least one method with the `FOR TESTING` addition. The ABAP Unit framework provides a `CL_ABAP_UNIT_ASSERT` class, which contains static methods for the verification of test expectations.

System	
<input type="checkbox"/>	is SAP inhouse system

Client	
<input type="checkbox"/>	Prohibit execution
Limit of Risk Level	Harmless
Changed by	

Application Server	
Long Duration	3,600
Medium Duration	300
Short Duration	60
Last change by	

Figure 21.2 Test Class Properties

In order to execute the test, we need to provide context for the test class. The context of a unit test is called a *fixture*. ABAP Unit executes certain predefined methods before executing the test method to prepare the system under test or to perform clean up after testing. We can use the fixture to prepare the components of the system under test or to load the test data.

There are two predefined methods called by ABAP Unit:

► **SETUP**

This method is called before each test within a test class. We can use this method to set up the required context for the test, such as test data.

► **TEARDOWN**

This method is called after each test within the test class. We can use this method to clear the context after each test.

Let's explain how to implement test classes with a working example. For this example, we'll use a class that provides the total tax due for each customer. The purpose of this class is to calculate the total tax amount that the customer is due. The class can be defined as shown in [Listing 21.4](#).

```
CLASS lcl_account DEFINITION.
  PUBLIC SECTION.
    TYPES: BEGIN OF ty_customer,
           acc_no TYPE char5,
           tax_amt TYPE bseg-mwsts,
```

```

        END OF ty_customer.
    TYPES tt_customer TYPE STANDARD TABLE OF ty_customer.
    METHODS: set_customer IMPORTING im_customer TYPE tt_customer,
             get_tax_due IMPORTING im_acc_
no TYPE char5 RETURNING VALUE(rt_tax_amt) TYPE bseg-mwsts.
    PRIVATE SECTION.
    DATA it_customer TYPE STANDARD TABLE OF ty_customer.

ENDCLASS.
CLASS lcl_account IMPLEMENTATION.
    METHOD set_customer.
        it_customer = im_customer.
    ENDMETHOD.
    METHOD get_tax_due.
        FIELD-SYMBOLS <fs_customer> LIKE LINE OF it_customer.
        LOOP AT it_customer ASSIGNING <fs_customer> WHERE acc_no EQ im_acc_
no.
            rt_tax_amt = <fs_customer>-tax_amt.
        ENDOLOOP.
    ENDMETHOD.
ENDCLASS.

```

Listing 21.4 Class to Calculate Tax Due

In [Listing 21.4](#), we defined a `get_tax_due` method, which imports the customer account number and returns the total tax due for the customer. We also implemented the `set_customer` setter method to initialize the customer table. For normal testing, this class would depend on data from the database to return the tax due for customer, but ABAP Unit is all about eliminating these external dependencies.

The test class for this class can be defined as shown in [Listing 21.5](#) to perform the test with some mock data.

```

CLASS lcl_test DEFINITION FOR TESTING
    RISK LEVEL HARMLESS
    DURATION SHORT
.
PRIVATE SECTION.
DATA o_customer TYPE REF TO lcl_account.
METHODS : setup,
          teardown,
          check_valid_tax FOR TESTING,
          check_invalid_tax FOR TESTING.

ENDCLASS.

```

Listing 21.5 Test Class Definition

In [Listing 21.5](#), we defined the `LCL_TEST` test class using the `FOR TESTING` addition and set the `RISK LEVEL` and `DURATION` properties. In the private section of the test class, we defined the `O_CUSTOMER` attribute as a reference object of the class under test.

We also defined the `SETUP` and `TEARDOWN` methods, which will be called automatically by ABAP Unit before it calls the `CHECK_VALID_TAX` and `CHECK_INVALID_TAX` test methods. We defined these test methods using the `FOR TESTING` addition. Note that we defined two test methods to ensure that the class under test is tested for both valid and invalid scenarios.

[Listing 21.6](#) shows the implementation of the test class.

```

CLASS lcl_test IMPLEMENTATION.
  METHOD setup.
    TYPES: BEGIN OF ty_customer,
            acc_no TYPE char5,
            tax_amt TYPE bseg-mwsts,
          END OF ty_customer.
    DATA : lt_customer TYPE STANDARD TABLE OF ty_customer,
            lw_customer LIKE LINE OF lt_customer.
    lw_customer-acc_no = '1001'.
    lw_customer-tax_amt = '50'.
    APPEND lw_customer TO lt_customer.

    lw_customer-acc_no = '1001'.
    lw_customer-tax_amt = '80'.
    APPEND lw_customer TO lt_customer.

    lw_customer-acc_no = '1002'.
    lw_customer-tax_amt = '100'.
    APPEND lw_customer TO lt_customer.

    lw_customer-acc_no = '1002'.
    lw_customer-tax_amt = '200'.
    APPEND lw_customer TO lt_customer.
    CREATE OBJECT o_customer.
    o_customer->set_customer( lt_customer ).

  ENDMETHOD.
  METHOD teardown.
    CLEAR o_customer.
  ENDMETHOD.
  METHOD check_valid_tax.
    DATA lv_tax_amt TYPE bseg-mwsts.
    lv_tax_amt = o_customer->get_tax_due( '1001' ).
    cl_abap_unit_assert=>assert_equals(

```

```

        EXP                = 130
        act                = lv_tax_amt
        msg                = 'Tax calculation is incorrect'
    ).
ENDMETHOD.
METHOD check_invalid_tax.
    DATA lv_tax_amt TYPE bseg-mwsts.
    lv_tax_amt = o_customer->get_tax_due( '2000' ).
    cl_abap_unit_assert=>assert_equals(
        EXP                = 0
        act                = lv_tax_amt
        msg                = 'Tax calculation is incorrect'
    ).
ENDMETHOD.
ENDCLASS.

```

Listing 21.6 Test Class Implementation

For the implementation of the test class in [Listing 21.6](#), we used the `SETUP` method to load some test data to the class under testing. We're using the `TEARDOWN` method to clear the reference to the class under testing. This ensures that each test is performed with a new context.

Within the `CHECK_VALID_TAX` and `CHECK_INVALID_TAX` test methods, we're calling the `GET_TAX_DUE` method of the class under testing by passing different customer account numbers. We're using the `ASSERT_EQUALS` static method of the `CL_ABAP_UNIT_ASSERT` class to compare the returned value with the expected value and show a message accordingly when the test is run.

[Listing 21.7](#) shows the complete code example of the program we've discussed so far.

```

REPORT zdemo_abap_unit.
CLASS lcl_account DEFINITION.
    PUBLIC SECTION.
        TYPES: BEGIN OF ty_customer,
                acc_no TYPE char5,
                tax_amt TYPE bseg-mwsts,
            END OF ty_customer.
        TYPES tt_customer TYPE STANDARD TABLE OF ty_customer.
        METHODS: set_customer IMPORTING im_customer TYPE tt_customer,
                get_tax_due IMPORTING im_acc_no TYPE char5 RETURNING VALUE(rt_
tax_amt) TYPE bseg-mwsts.
    PRIVATE SECTION.
        DATA it_customer TYPE STANDARD TABLE OF ty_customer.
ENDCLASS.

```



```

CLASS lcl_account IMPLEMENTATION.
  METHOD set_customer.
    it_customer = im_customer.
  ENDMETHOD.
  METHOD get_tax_due.
    FIELD-SYMBOLS <fs_customer> LIKE LINE OF it_customer.
    LOOP AT it_customer ASSIGNING <fs_customer> WHERE acc_no EQ im_acc_
no.
      rt_tax_amt = <fs_customer>-tax_amt.
    ENDLOOP.
  ENDMETHOD.
ENDCLASS.

CLASS lcl_test DEFINITION FOR TESTING
  RISK LEVEL HARMLESS
  DURATION SHORT
  .
PRIVATE SECTION.
  DATA o_customer TYPE REF TO lcl_account.
  METHODS : setup,
            teardown,
            check_valid_tax FOR TESTING,
            check_invalid_tax FOR TESTING.

ENDCLASS.

CLASS lcl_test IMPLEMENTATION.
  METHOD setup.
    TYPES: BEGIN OF ty_customer,
            acc_no TYPE char5,
            tax_amt TYPE bseg-mwsts,
          END OF ty_customer.
  DATA : lt_customer TYPE STANDARD TABLE OF ty_customer,
          lw_customer LIKE LINE OF lt_customer.
  lw_customer-acc_no = '1001'.
  lw_customer-tax_amt = '50'.
  APPEND lw_customer TO lt_customer.

  lw_customer-acc_no = '1001'.
  lw_customer-tax_amt = '80'.
  APPEND lw_customer TO lt_customer.

  lw_customer-acc_no = '1002'.
  lw_customer-tax_amt = '100'.
  APPEND lw_customer TO lt_customer.

  lw_customer-acc_no = '1002'.
  lw_customer-tax_amt = '200'.
  APPEND lw_customer TO lt_customer.
  CREATE OBJECT o_customer.
  o_customer->set_customer( lt_customer ).

```

```

ENDMETHOD.
METHOD teardown.
  CLEAR o_customer.
ENDMETHOD.
METHOD check_valid_tax.
  DATA lv_tax_amt TYPE bseg-mwsts.
  lv_tax_amt = o_customer->get_tax_due( '1001' ).
  cl_abap_unit_assert=>assert_equals(
    exp          = 130
    act          = lv_tax_amt
    msg          = 'Tax calculation is incorrect'
  ).
ENDMETHOD.
METHOD check_invalid_tax.
  DATA lv_tax_amt TYPE bseg-mwsts.
  lv_tax_amt = o_customer->get_tax_due( '2000' ).
  cl_abap_unit_assert=>assert_equals(
    exp          = 0
    act          = lv_tax_amt
    msg          = 'Tax calculation is incorrect'
  ).
ENDMETHOD.
ENDCLASS.

```

Listing 21.7 ABAP Program with ABAP Unit

Once the test class is implemented, the main class can be tested against the test class and improved by way of any required corrections. The concept of TDD revolves around the idea that you develop the class, test for expected behavior, fix issues or redesign, and test again until you get the right behavior. For our class, the right behavior meant having the `GET_TAX_DUE` method return the total tax amount due for a customer.

However, if you examine the implementation of the method in [Listing 21.4](#), you'll see that we intentionally included a bug: The method returns the tax value of the last record instead of returning the sum of the values. Let's run the test and see if this bug is identified.

To run the test, follow the menu path `PROGRAM • EXECUTE • UNIT TESTS`. The system will perform the test and show the test results as in [Figure 21.3](#).

As you can see, the `CHECK_INVALID_TAX` test has passed, and the `CHECK_VALID_TAX` test has failed. We can change the tax calculation logic in the `GET_TAX_DUE` method of our `LCL_ACCOUNT` class and run the test again via `PROGRAM • EXECUTE • UNIT TESTS` until the class performs as expected.

ABAP Unit: Result Display						
Task/Program/Class/Method	Sta...	Fail...	Exce...	Runt...	War...	Failures and Messages
TASK_E9685324_20160902	1	1	0	0	0	Prio Message Critical Assertion Error: 'Tax calculation is incorrect' Analysis Different Values: Expected [130] Actual [80.00] Test 'LCL_TEST->CHECK_VALID_TAX' in Main Program 'ZDEMO_ABAP_UNIT'. Stack Include: <ZDEMO_ABAP_UNIT> Line: <73> (CHECK_VALID_TAX)
ZDEMO_ABAP_UNIT	1	1	0	0	0	
LCL_TEST	1	1	0	0	0	
CHECK_INVALID_TAX	0	0	0	0	0	
CHECK_VALID_TAX	1	1	0	0	0	

Figure 21.3 Test Results

Implementing test classes for each function of an application may be a time-consuming task, but it ensures that each function is tested thoroughly and that future changes won't break the behavior of the existing functionality.

21.3 Code Inspector

Code Inspector is one of the tools most frequently used by ABAP developers. This tool allows you to analyze program code to identify any potential problems. Code Inspector combines various tools to perform a check on different program aspects, such as performance-related, security-related, syntax-related, and UI-related aspects, in order to provide a broad analysis. Code Inspector is intended to be used by developers and quality managers.

The following use cases can be implemented in Code Inspector:

- ▶ Run an ad hoc check from ABAP Workbench and within Code Inspector.
- ▶ Manage exceptions.
- ▶ Define public and private reusable code inspections.
- ▶ Perform special test tasks.
- ▶ Identify usages of simplified SAP objects in customer objects (useful when migrating custom code to SAP S/4HANA).

You can run an ad hoc check on a program by navigating to the menu option (PROGRAM/FUNCTION MODULE/OBJECT TYPE, etc., depending on the object) and selecting CHECK • CODE INSPECTOR (or by going to Transaction SCII), as shown in [Figure 21.4](#).

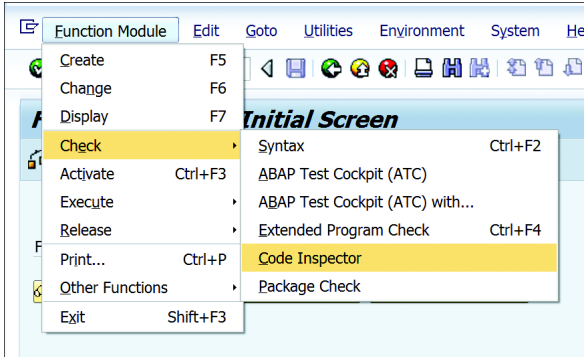


Figure 21.4 Accessing Code Inspector

When Code Inspector is called directly from an object, it will immediately execute the check on the object and display the results. When you access Code Inspector using Transaction SCII, it will display the initial screen shown in [Figure 21.5](#).

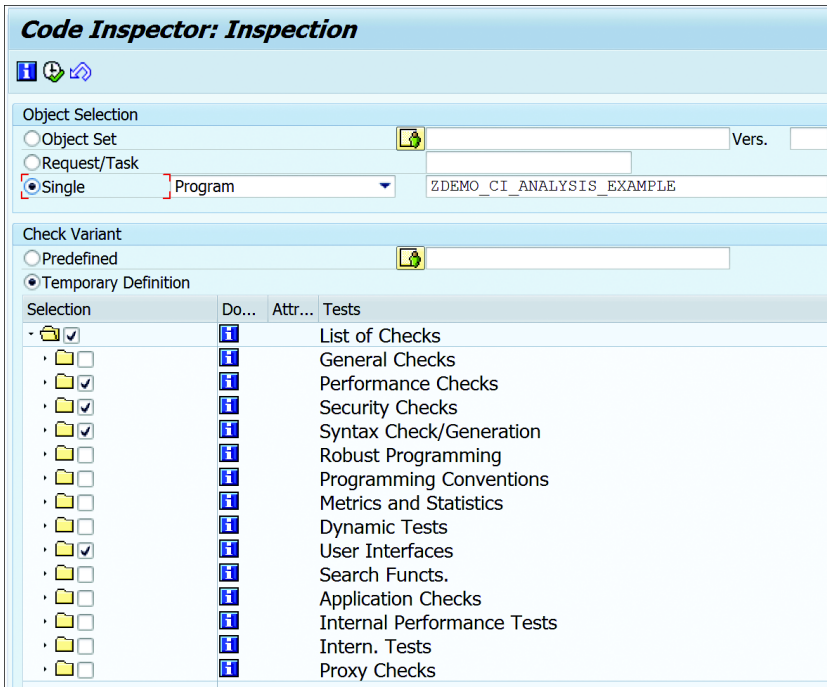


Figure 21.5 Code Inspector Initial Screen

On this initial screen, you can specify the program name and select the check categories. The results are grouped in a hierarchical structure, as shown in [Figure 21.6](#).

We can navigate through the hierarchical folder structure to analyze the checks on various parameters. The result also displays a summary of the statistics of all the potential problems, categorized as ERRORS, WARNINGS, or INFORMATION messages. You can double-click the message text to navigate to the corresponding code in the program. Ideally, the program should show a zero in each one of these statistics columns.

Documentation	Exceptions	Tests	Error	Warnings	Information
		List of Checks	1	0	1
		Performance Checks	1	0	0
		Analysis of WHERE Condition for SELECT	1	0	0
		Errors	1	0	0
		Message Code 0501	1	0	0
		Program ZDEMO_CI_ANALYSIS_EXAMPLE Include ...	1	0	0
		Large table CDHDR: No WHERE condition ==> Large table CDHDR: No WHERE condition	0	0	0
		Analysis of WHERE Condition in UPDATE and DELE...	0	0	0
		SELECT Statements That Bypass the Table Buffer	0	0	0
		Low Performance Operations on Internal Tables	0	0	0
		Low-Perform. Parameter Transfers	0	0	0
		Instance Creation of BADIs	0	0	0
		Security Checks	0	0	0
		Syntax Check/Generation	0	0	1
		Extended Program Check (SLIN)	0	0	1
		Information	0	0	1
		Message Code 1701 ==> The text symbol 001 is not defined in the text... ZDEMO_CI_ANALYSIS_EXAMPLE in the original lan...	0	0	1
		User Interfaces	0	0	0

Figure 21.6 Code Inspector Results

Sometimes, it isn't possible to correct the errors shown in Code Inspector. In such cases, an error can be masked using a pseudocomment. If the statement can be masked using a pseudocomment, then a comment icon will be displayed in the EXCEPTIONS column.

The blue INFORMATION icon provides documentation explaining each type of check and what exactly is checked under that category. Clicking the blue INFORMATION icon next to a message will display suggestions to correct the corresponding error.

The results of the ad hoc test aren't stored; they'll be lost when you exit the screen. You can also access Code Inspector using Transaction SCII, which allows you to configure, select, and store individual tests in a *check variant*. From Transaction SCII, you can also create an object set to perform the test on a group of objects.

21.4 Selectivity Analysis

Selectivity analysis is performed via Transaction DB05. The selectivity analysis tool requires a certain amount of data volume to show meaningful results. This test is typically performed in the quality assurance system, which is generally a replica of the production system.

This tool helps analyze the data distribution and is used to define the size of the tables that should be buffered. The selectivity analysis tool also helps with index design.

To run selectivity analysis, on the initial screen of Transaction DB05 (see [Figure 21.7](#)), enter the table name for which the analysis should be performed. You can perform the analysis on the key fields or enter up to five fields of the table. By selecting the SUBMIT ANALYSIS IN BACKGROUND checkbox, you can run the test in the background. Running the check in the background is useful for large tables when you don't want to block the dialog work process for a long time or if you're receiving any timeout errors when the test is run in the foreground.

DB05: Analysis of Table with Respect to Indexed Fields

Table
Table: T100T

Analysis for
 Primary Key
 Specified Fields

Fields	
1	SPRSL
2	STEXT
3	
4	
5	

Submit Analysis in Background

Figure 21.7 Initial Screen of Transaction DB05

When the test is run in the background, you can check the result using Transaction SM37. On the result screen, as shown in [Figure 21.8](#), various statistics of the table are displayed.

The first half of the screen shows information such as the number of rows, buffer size, and buffer mode. This information helps identify the memory the table is occupying in the buffer. In our example, table T001T requires about 4.8 MB to be fully buffered.

The second half of the screen shows the number of times the values for the combination of fields occur. In our example, the field SPRSL occurs thirty-one times distinctly, out of which three values occur between one and ten times in a total 40,064 rows. Four values occur between eleven and one hundred times, thirteen values occur between 101 and one thousand times, ten values occur between 1001 and ten thousand times, and one value occurs between 10,001 and one hundred thousand times. There are 37,787 combinations of the SPRSL and STEXT fields, out of which 37,776 occur between one and ten times, and so on.

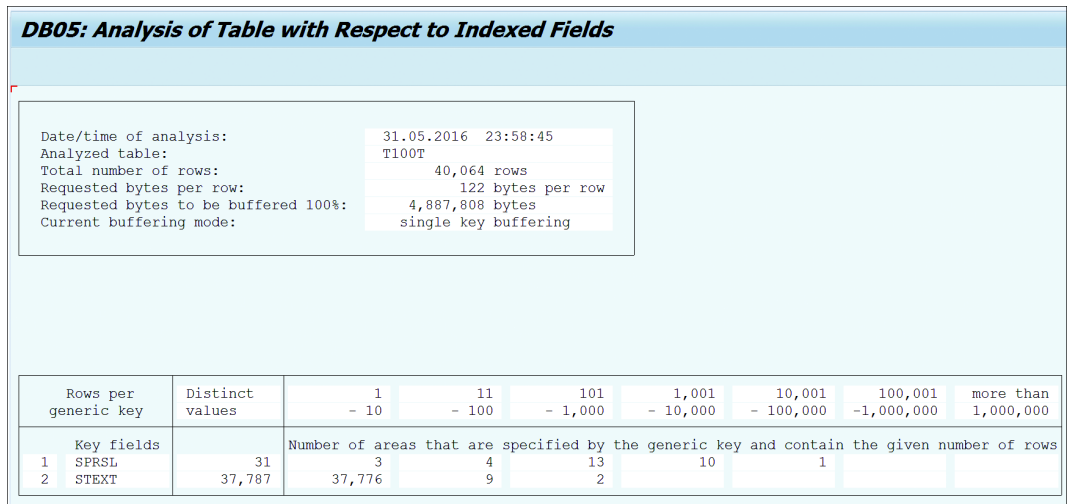


Figure 21.8 Result of Transaction DB05

Based on this information, you can make decisions about table buffering. For example, if we switch to generic buffering for the SPRSL and STEXT fields from the current single key buffering, we'll end up buffering two generic areas that contain two records between 101 and one thousand records. If this key isn't in the buffer,

then the system will have to copy all these records to the buffer when the `SELECT` statement is executed in order to read the line.

21.5 Process Analysis

Using Transactions SM50 and SM66, we can check the work process overview. These transactions are generally used by system administrators. However, we can use these transactions to check the programs that are currently running, and if our program is running for a long time, then we can check what step of the program is taking the most time. This information helps us identify which part of the program should be analyzed further. These transactions also help to check programs that run in background mode.

Figure 21.9 shows the `WORK PROCESS OVERVIEW` screen (Transaction SM50). Work processes will have statuses such as the following:

- ▶ **WAITING**
When the work process has a `WAITING` status, it means it's freely available to take a request.
- ▶ **RUNNING**
If the work process has a `RUNNING` status, it means the work process is currently busy processing a request.
- ▶ **ON HOLD**
The `ON HOLD` status signifies that the work process is waiting for some response, such as the release of a resource or a response from an RFC call. If the program is opened in debug mode, the corresponding work process that's processing the request will have a `HOLD` status. The column next to the `STATUS` column displays the reason for the hold. `F1` helps provide information about all the possible reasons for a `HOLD` status.
- ▶ **STOPPED**
If the work process status shows a `STOPPED` status, the work process was aborted and didn't restart.

Time		01.06.2016, 01:23:26											
Total Number of Work Processes		16											
Dialog		10 / 8 (Total/Free) , Average Load = (1.108 / 0.599 / 0.286)											
Update		1 / 1 (Total/Free)											
Background		3 / 3 (Total/Free) , Average Load = (0.218 / 0.295 / 0.165)											
Number	Type	PID	Status	"On Hold"	Failures	CPU	Time	Time	Priority	Executed	Program	Current Action	Info
0	DIA	3,252	Waiting				0:34:46						
1	DIA	5,316	Waiting				0:26:04						
2	DIA	7,160	Running				0:01:07		High		CL_SERVER_INFO=====CP		
3	DIA	2,948	On Hold	Debugging			0:29:20		7 High		CL_DSH_DDIC_ASSIGNMENT=====CP	SAP GUI Response	
4	DIA	5,992	Waiting				0:18:58						
5	DIA	4,936	Waiting				0:04:36						
6	DIA	4,440	Running				0:01:47		High		SAPMSSY2		
7	DIA	2,236	Waiting				0:59:37						
8	DIA	2,780	Waiting		1		0:01:14						
9	DIA	4,656	Waiting				1:02:57						
10	UPD	5,484	Waiting				0:02:53						
11	BTC	2,748	Running				8:40:32				SAPLKIKI		
12	BTC	6,556	Running				8:31:21		1		SAPLSENA		
13	BTC	5,748	Running				9:17:16				SAPMSSY2	DB Insert	TST01
14	SPO	6,944	Waiting				0:00:26						
15	UPD2	5,792	Waiting				0:00:04						

Figure 21.9 Work Process Overview

The WORK PROCESS OVERVIEW screen provides some important information, such as the time the program is taking to complete the current step, the current action being performed by the program, the status of the program, and any failures in program execution.

Note

If you don't see all the columns in the overview screen, you can change the ALV layout via the menu path SETTINGS • LAYOUT • CURRENT.

The information in the CURRENT ACTION column helps identify if the program is stuck while performing any particular action, such as database actions like sequential READ, DIRECT READ, INSERT, UPDATE, DELETE, and so on. If the program is taking long to perform any of these actions, then the program can be analyzed further using a tool such as a SQL trace.

The CURRENT ACTION column also shows different SAP system actions, such as LOAD REPORT, ROLL-IN, ROLL-OUT, and so on. These actions mean that the work process is occupied on the SAP system side and not related to our program performance. If the work process is spending lot of time performing these SAP

system actions, then the system administrator should analyze them further, because these actions usually don't take much time.

If the CURRENT ACTIONS column is empty, the program is processing ABAP code, such as LOOP statements or READ statements. You can check the CPU TIME column to analyze the performance of these statements. If the program is taking too long with an empty CURRENT ACTION column, then you can perform a more detailed analysis using the runtime analysis tool.

Transaction SM50 provides a local process overview of the current application server to which we're logged in; Transaction SM66 instead provides a *global* process overview. In Transaction SM50, you can double-click any row to see more detailed statistics, such as memory statistics, database statistics, and so on, as shown in [Figure 21.10](#)

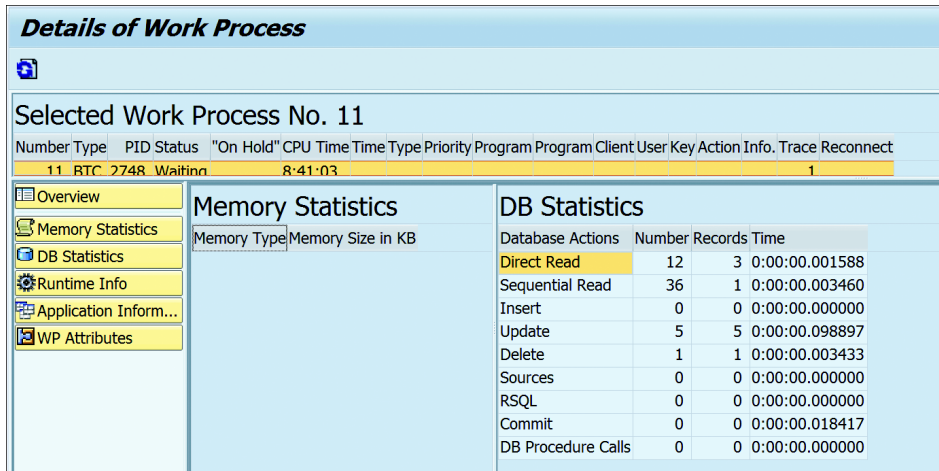


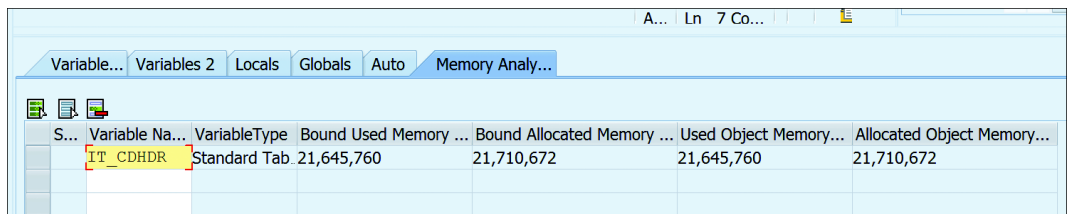
Figure 21.10 Detail Overview Screen

Process analysis can be used as a first step to identify a problem area and then used to continue the analysis in the corresponding area. For example, the CURRENT ACTION column guides you in a particular direction to carry out further analysis using other tools. However, this tool is useful only if the program takes a long time to execute.

21.6 Memory Inspector

Memory analysis can be performed using a combination of the New Debugger and Transaction S_MEMORY_INSPECTOR. The debugger is generally used to troubleshoot the logical problems in ABAP programs. However, you can use the debugger to identify the memory occupied by the data objects as well. This information is useful to identify large internal tables, which can create problems if their access isn't optimized. For example, performing a linear search on a huge table will affect performance considerably.

In the New Debugger, you can check the MEMORY ANALYSIS tab to see the memory occupied by the data objects, as shown in [Figure 21.11](#).



S...	Variable Na...	VariableType	Bound Used Memory ...	Bound Allocated Memory ...	Used Object Memory...	Allocated Object Memory...
	IT_CDHDR	Standard Tab	21,645,760	21,710,672	21,645,760	21,710,672

Figure 21.11 Memory Analysis Tab View

In this section, we'll discuss how you can create and compare memory snapshots.

21.6.1 Creating Memory Snapshots

You can create a memory snapshot for further analysis of the memory usage by the program. A *memory snapshot* is a snapshot of the memory consumed by the ABAP program at a particular point in time. By comparing memory snapshots taken at different periods, you can analyze the memory usage of the program over a period of time or in different scenarios.

Follow these steps to create a memory snapshot:

1. Select the NEW TOOL button and then select MEMORY ANALYSIS under the MEMORY MANAGEMENT folder, as shown in [Figure 21.12](#).
2. Once the memory analysis tool is displayed, select the SERVICES OF THE TOOL button and select CREATE MEMORY SNAPSHOT, as shown in [Figure 21.13](#). You can also use the menu path SYSTEM • UTILITIES • MEMORY ANALYSIS • CREATE MEMORY SNAPSHOT.

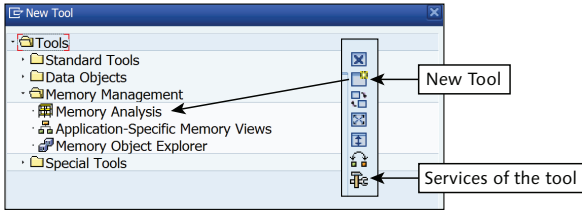


Figure 21.12 Memory Analysis Tool

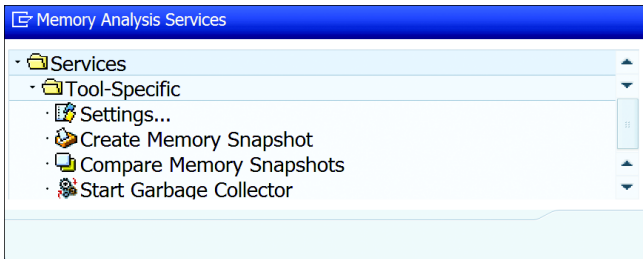


Figure 21.13 Create Memory Snapshot

21.6.2 Comparing Memory Snapshots

You use the Memory Inspector tool to analyze the memory snapshots. Comparing memory snapshots helps you analyze how tables or other dynamic data objects are growing between intervals. Identifying the rate at which tables grow can help identify potential problems.

To compare memory snapshots, follow these steps:

1. Go to Transaction S_MEMORY_INSPECTOR to compare memory snapshots or use the menu path SYSTEM • UTILITIES • MEMORY ANALYSIS • COMPARE MEMORY SNAPSHOT. The initial screen lists all the memory snapshots available for analysis, as shown in [Figure 21.14](#).
2. Double-click the snapshots you wish to compare and then select T_0-T_1 in the MEMORY SNAPSHOT dropdown field at the center of the screen. This will compare the two snapshots and allow you analyze the changes between the snapshots, as shown in [Figure 21.15](#). For example, you can see the difference in memory occupied by the dynamic data objects between memory snapshots.

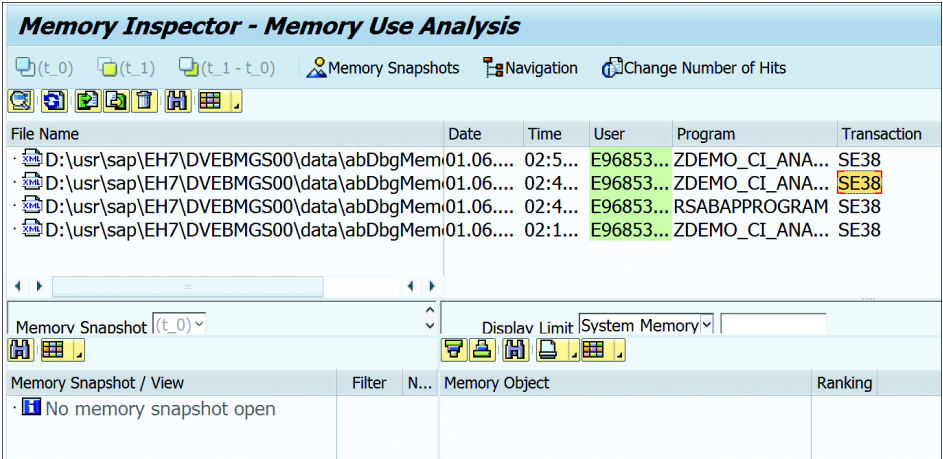


Figure 21.14 Memory Snapshots

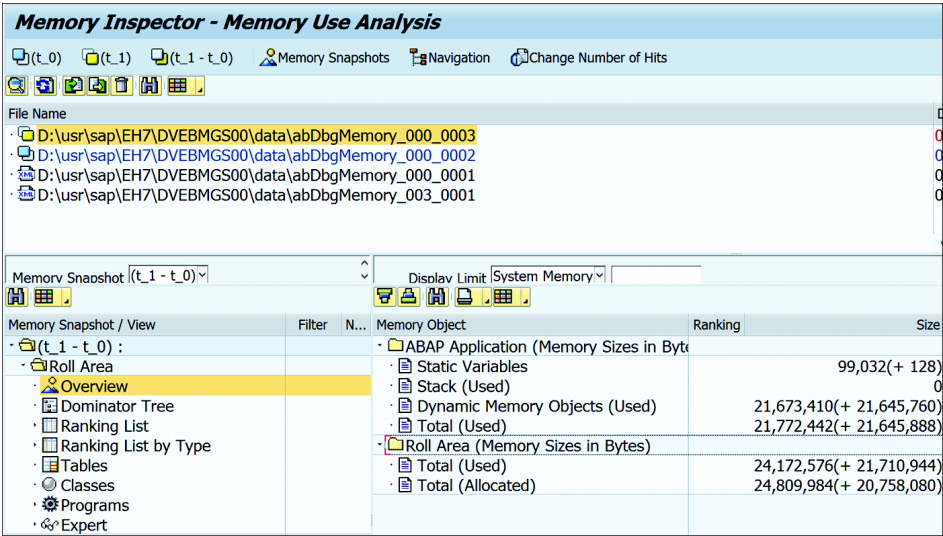


Figure 21.15 Comparing Memory Snapshots

21.7 Table Call Statistics

Table call statistics can be checked via Transaction ST10. This transaction is mostly used by system administrators to check the statistics of tables that are buffered; it

allows them to optimize the buffering settings accordingly. By checking the table statistics, system administrators can also analyze the unbuffered tables to see how frequently the records are changing and decide if they should be buffered.

Buffering helps speed up applications, and this tool provides the required information to identify the technical feasibility of buffering and determine if there's a strong business case to buffer tables.

As ABAP programmers, we can view the table call statistics to check the buffer state of a table. For example, if you notice that buffering isn't used for a particular table when checking the SQL trace, you can confirm if the table is buffered at all in Transaction ST10. If the table is buffered, but the SQL statement is bypassing the buffer, then the problem is with the SQL statement, which you need to correct.

To check the table call statistics, open Transaction ST10 and proceed with the following steps:

1. On the initial screen (see [Figure 21.16](#)), select SINGLE KEY BUFFERED or GENERIC KEY BUFFERED based on the buffer type of the table you want to check.

Table Call Statistics

Tables

All Tables

Not Buffered

Single Key Buffered

Generic Key Buffered

Time Frame

Since Startup

Today Previous Day

This Week Previous Week

This Month Previous Month

Server

All Servers This Server

Show Statistics

Figure 21.16 Table Call Statistics Initial Screen

- Under SERVER, select THIS SERVER to check the statistics for the application server you're currently logged on to. You can also check all servers or any specific server by choosing the relevant option.
- Under TIME FRAME, select SINCE STARTUP and click SHOW STATISTICS. This will show the results screen, listing all the tables matching the selection criteria.

The results screen displays the buffer state of each table next to the table name, as shown in [Figure 21.17](#).

Table	Buffer State	Buf key opt	Invalidations	Total	ABAP Processor requests		Changes	DB activity	
					Direct reads	Seq. reads		Calls	Rows affected
Total			573545	78,342,990	71,147,382	7,162,536	33,072	6,464,353	175,052,296
T047	valid	sng	0	5	3	1	1	2	2
T048B	absent	sng	0	1	0	1	0	3	1
T053R	absent	sng	0	2	0	1	1	6	77
T053S	absent	sng	0	2	0	1	1	10	999
T100	valid	sng	0	8,512,540	8,512,304	224	12	289,277	52,873,354
T100A	valid	sng	0	195	181	14	0	97	703
T100O	absent	sng	0	64	0	64	0	192	64
T100T	valid	sng	0	1,049	543	506	0	513	160,702
T100U	valid	sng	0	75	39	36	0	150	493
T159L	valid	sng	0	4	3	1	0	0	2
T161V	absent	sng	0	1	0	1	0	4	5
T16FW	absent	sng	0	1	0	1	0	3	1
T185E	valid	sng	0	319	319	0	0	0	0
T327	absent	sng	0	1	0	1	0	4	2
T370A	valid	sng	0	14	14	0	0	0	2
T399D	valid	sng	0	20	19	1	0	0	3
T437D	valid	sng	0	2	1	1	0	0	1
T552A	valid	sng	0	395,099	395,099	0	0	2	721
T599B	valid	sng	0	42	40	0	2	3	0
T599C	valid	sng	0	58	58	0	0	0	3
T599R	valid	sng	0	114	114	0	0	0	0
T599W	valid	sng	0	40	40	0	0	0	2
T5H4Y	valid	sng	0	2	1	1	0	0	0
T604	valid	sng	0	7	7	0	0	0	1
T604I	absent	sng	0	1	0	1	0	3	1
T604S	valid	sng	0	23	23	0	0	0	0
T604T	valid	sng	0	4	4	0	0	0	1

Figure 21.17 Table Call Statistics Result Screen

Most of the other information displayed on the results screen is generally used by system administrators for buffer analysis. For example, the values in DIRECT READS indicate if the table is suitable for single-record buffering, because sequential reads aren't executed using single-record buffering.

Similarly, the CHANGES column provides information about how many times records were changed. This information, along with the number of records read, allows you to calculate the change rate. If the change rate is high, then the table may not be suitable for buffering because it may cause too much system overhead. We always want to buffer tables that are read frequently but seldom changed. The INVALIDATIONS column gives you information about how many times the table is invalidated due to changes.

Even though most of this information is useful for system administrators, as an ABAP developer the most crucial point is to check the buffer state of the table. If

you notice that a table that should be buffered isn't being buffered, you can ask the system administrator to analyze it further. On the other hand, if the table is buffered, but the SQL statement in the program is bypassing the buffer, you can fix the SQL statement (e.g., the table is set for generic buffering, but you didn't include the complete generic key in your SQL query, which causes the statement to bypass the buffer).

21.8 Performance Trace

Performance traces are carried out in the application server and allow you to log activities such as database access, remote calls of programs, locking activities, table buffer calls, and more. You can access a performance trace using Transaction ST05 or via the menu path `SYSTEM • UTILITIES • PERFORMANCE TRACE`. Either method opens the screen shown in [Figure 21.18](#).

There are four types of traces available:

- ▶ **SQL trace**

A SQL trace allows you to monitor the database access of programs.

- ▶ **Enqueue trace**

An enqueue trace monitors locking activity.

- ▶ **RFC trace**

An RFC trace allows you to monitor RFCs between different instances.

- ▶ **Buffer trace**

A table buffer trace allows you to check the buffer access by programs.

In addition to these traces, Transaction ST05 can be used to trace external requests, such as HTTP (HTTP trace), ABAP push channels (APC trace), and ABAP messaging channels (AMC trace) requests. The HTTP, APC, and AMC traces are useful when tracing web applications such as Web Dynpro ABAP or SAPUI5. Discussion of these external traces is beyond the scope of this book.

In this section, we'll discuss how to activate performance traces and each of the different types of trace that can be performed.

Figure 21.18 Transaction ST05 Initial Screen

21.8.1 Activating and Filtering a Performance Trace

To run a trace, it must first be activated. When the trace is activated, it's activated for the current application server. Only one trace can be activated at a time, so if the trace is already activated by another user, you can't activate it until the other user deactivates it.

The initial screen of Transaction ST05 shows the activation status (refer back to [Figure 21.18](#)). Click the **ACTIVATE TRACE** button to activate the trace for your user ID. Under the **SELECT TRACE TYPE** section, you can select the type of the trace you want to activate. You can activate multiple types at once.

If you want to activate a trace for other users or only for certain programs, transactions, tables, and so on, select the **ACTIVATE TRACE WITH FILTER** button. This will open a dialog box, as shown in [Figure 21.19](#), in which you can provide the filter criteria; for example, you can list the programs that should be traced in the **INCLUDE** column or list the programs that should be excluded in the **EXCLUDE** column (see [Figure 21.19](#)).

Figure 21.19 Trace with Filter

As soon as the trace is activated, the activated traces are displayed under the TRACE STATUS section of the ST05 PERFORMANCE TRACE screen ([Figure 21.18](#)) and the recording starts.

Any actions executed before the trace was activated won't be displayed in the log; the trace only records the activities carried out after the trace is activated. Once the trace is activated, execute the transactions that you want to analyze. Once all the transactions that you want to analyze are executed, deactivate the trace by selecting it in the TRACE STATUS section and clicking the DEACTIVATE button.

All the activities will be written to a log file, which you can check by clicking the DISPLAY TRACE button ([Figure 21.18](#)). The size of the log file is predefined to avoid system overflow. If the size of the log file exceeds the predefined value, the file contents will be overwritten, with old records overwritten by new records. If the size is exceeded during the trace, you'll see a message about the overwritten status while you view the trace. You can check the saved traces anytime in the future by clicking the DISPLAY TRACE DIRECTORY button in the toolbar.

Once you click the DISPLAY TRACE button, a selection screen is presented, as shown in [Figure 21.20](#), in which you can set the filter conditions for the trace log. For example, you can check the trace records for a specific period, those executed

by a specific client or specific user, those run for a specific program, and so on. Click EXECUTE to see the trace log.

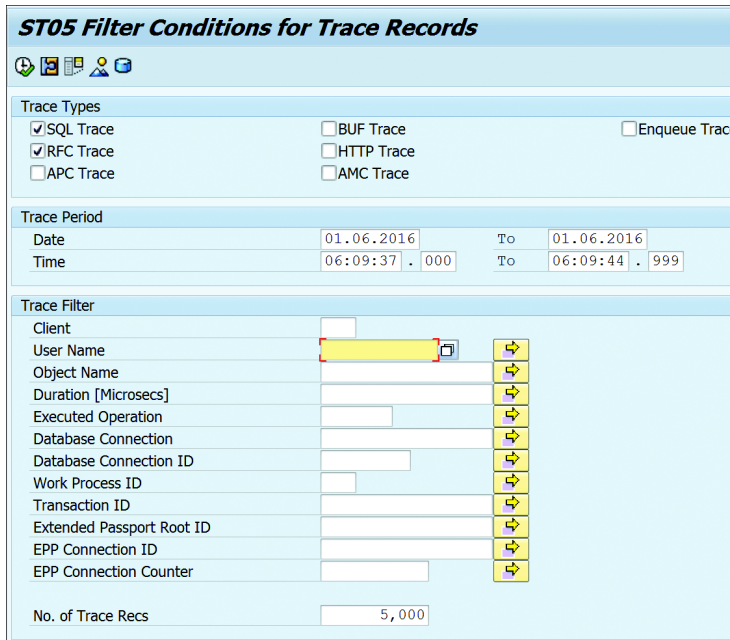


Figure 21.20 Display Trace

The trace log shows all the recorded activities in an ALV display. Select the right-most icon in the toolbar to see different views, as shown in [Figure 21.21](#).

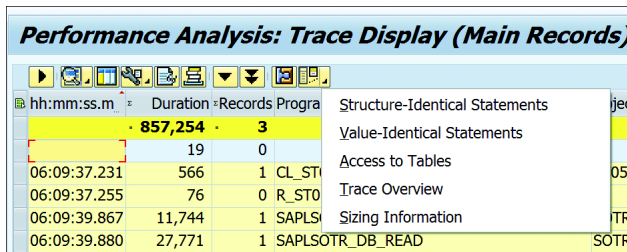


Figure 21.21 Trace Log

Now that you understand how to activate a performance trace, in the next few sections we'll look at the different types of performance traces.

21.8.2 SQL Trace

The *SQL trace* option in Transaction ST05 records all the SQL statements that were sent from the database interface to the database. As discussed in [Chapter 2, Section 2.2.3](#), when we write an Open SQL statement in an ABAP program, the database interface of the work process takes care of converting the Open SQL statements to Native SQL statements. The SQL trace allows you to see how the Open SQL statements are converted to Native SQL statements by the database interface and how the parameters are passed with the Native SQL statements to the database.

The system starts recording all the database activity from the moment the SQL trace is activated and logs all the converted Native SQL statements, along with the parameters that are passed from the database interface to the database. All the statistics, such as number of entries inserted or deleted, time spent on each activity, programs that were run, statements that were executed, and so on, are logged in the trace file. The log file also stores information about the runtime of the statement and the position in the program where the statement occurs to support further analysis.

We can analyze the SQL trace file to identify the following:

- ▶ The different SQL statements that are executed by the program
- ▶ The values that are passed to the database
- ▶ How the Open SQL statements are converted to Native SQL statements
- ▶ Which part of the program makes repeated database access
- ▶ What data in the database is being updated by the program
- ▶ At what point in the program flow the program performs database commits

Before turning on the trace, we recommend running the program that you want to analyze. This helps to buffer any data that the program uses. The SQL trace only records table access that isn't buffered, so by running the program once before activating the trace you can check only those tables that aren't buffered. Because the trace logs all the database activity, it may affect the performance of the application server. You should always turn off the trace immediately after recording the program execution.

Once you turn on the trace, record the transaction and turn off the trace to display the results. The results screen shows the trace list of all the statements that were executed between turning on and turning off the trace. From the default view, as shown in [Figure 21.22](#), you can navigate to different views, as shown in [Figure 21.21](#). Each view provides different information about the database access, allowing you perform an in-depth analysis.

Performance Analysis: Trace Display (Main Records)

hh:mm:ss.m	Duration	Records	Program Name	Object Name	Statement	DB Cursor	Array DB Conn.
- 52,435		33					
	9	0			OTHER (SQL)		0
09:50:17.479	49,396	0	CL_ST05_TRACE_	ST05_TRACE	SELECT WHERE "MANDT"	1003	1 R/3
09:50:17.530	621	1	CL_ST05_TRACE_	ST05_TRACE	INSERT VALUES('800', 'E9	1003	1 R/3
09:50:17.535	746	0	R_ST05_TRACE_		COMMIT (forced)		0 R/3
09:50:21.356	985	0	CL_SRAL_RT_CO	SRAL_CONFIG_RT	SELECT WHERE "CLIENT"	1003	32,767 R/3
09:50:21.358	588	32	ZDEMO_SALV	MARA	SELECT WHERE "MANDT"	1003	269 R/3
09:50:21.448	90	0	SAPLOLEA		COMMIT		0 R/3

Figure 21.22 Display Trace

In the default view shown in [Figure 21.22](#), you can double-click any cell to open the respective view. For example, double-clicking a specific statement in the STATEMENT column will open the SQL statement's trace record, double-clicking an object name in the OBJECT NAME column will show information about the object, and so on.

The DURATION column shows the time taken to execute the statement. It takes into account not only the time taken to perform the action in the database, but also the time taken by the network between the database interface and the database.

As shown in [Figure 21.21](#), there are a number of selectable views to choose from:

- ▶ **STRUCTURE-IDENTICAL STATEMENTS**
This view allows you to check any redundant statements that were executed.
- ▶ **VALUE-IDENTICAL STATEMENTS**
This view allows you to check any statements that return identical values.
- ▶ **ACCESS TO TABLES**
This view provides the cumulative time spent on each table for different operations.

▶ **TRACE OVERVIEW**

This option provides various database access statistics, such as time spent on `SELECT`, `INSERT`, or `COMMIT` or the number of times these statements were executed. It also provides suggestions to address any unnecessary database access.

▶ **SIZING INFORMATION**

This view provides information about the data volume, such as the number of records inserted, modified, or deleted per table and the size of the tables.

Along with ABAP Trace, the SQL trace is an important tool that can provide great insight into database access. We recommend that you start your analysis from the tables that take up a long runtime and branch out to other views.

21.8.3 RFC Trace

The *RFC trace* allows you to check the RFCs that are executed between instances. This trace records both incoming and outgoing RFC calls. Using this trace, you can identify the time the programming is spending with RFC calls and therefore identify bottlenecks.

Among other things, using the RFC trace helps to analyze the following:

- ▶ Different function modules called remotely by the program
- ▶ Execution status of the RFC (i.e., if the RFC was executed successfully)
- ▶ The total time taken for processing the RFC call
- ▶ The size of the data sent and received during the RFC call
- ▶ Information about the instance for which the RFC call was executed
- ▶ The technical parameters of the instance

The RFC trace display view is similar to the SQL trace view and allows you to check information about each action. The important details are the duration it took to execute the RFC, the program that executed the RFC, and details about the client and the server.

The system differentiates between client and server RFCs. The client calls the RFC, and the server executes the RFC.

The PERFORMANCE ANALYSIS: TRACE OVERVIEW screen, as shown in [Figure 21.23](#), provides an overview of the number of RFCs made and how long it took to complete the execution of each.

Performance Analysis: Trace Overview			
System = EH7 Release = 740 DB-System = SAP DB 11.00.2100 01.06.2016 23:48:10			
Start of Trace Period	Date	:	01.06.2016 Time : 00:00:00,000
End of Trace Period	Date	:	01.06.2016 Time : 14:02:34,999
Time Interval		:	50,554.999s
Start of First Trace Record	Date	:	01.06.2016 Time : 06:09:39,923
End of Last Trace Record	Date	:	01.06.2016 Time : 11:01:33,016
Time Interval		:	17,513.093s
Executions	Duration	Records	
Total	7 (100.0%)	1.570s (100.0%)	0
RFC	7 (100.0%)	1.570s (100.0%)	0
Client	7 (100.0%)	1.570s (100.0%)	0

Figure 21.23 Trace Overview

21.8.4 Enqueue Trace

An *enqueue trace* allows you to analyze the lock requests made by an application. The trace list view (see [Figure 21.24](#)) provides information about all the requests made to the enqueue service.

This trace provides details about the following:

- ▶ The type of request made
- ▶ The duration of the request
- ▶ The user who made the request
- ▶ The work process type (e.g., background or dialog)

The DURATION column allows you to assess how long the request was active (between ENQUEUE and DEQUEUE).

Performance Analysis: Trace Display (Main Records)									
Time	Duration	Records	Program	Object	Statement	DB Cursor Array	DB Conn. User	Client WP	Type
14:24:12.381	487	1	SAPLSENA	EVVBRKE	ENQUEUE Excl(Cumulative)	VBRK 8000090031228	0	E9685324	800 DIA

Figure 21.24 Trace Overview Screen

Using an enqueue trace allows you to identify bottleneck situations by analyzing the time spent on enqueue requests.

21.8.5 Buffer Trace

Another useful trace is the *buffer trace*. Using a buffer trace, you can identify the following:

- ▶ Buffer accesses executed by the program statements
- ▶ Which type of buffer is used
- ▶ How the buffer is managed

In the trace list view, you can double-click a statement in the STATEMENT column to see detailed information about the buffer, such as buffer type and buffer key, as shown in [Figure 21.25](#).

Buffer Trace Record			
Details for Selected Buffer Trace Record			
Table Buffer Operation	:	OPEN	
Object Name	:	TFDIR	
Buffering Type	:	Single Records Buffered	
Key Length	:	60	
Generic Key Length	:	60	
Buffer Type Short Descr.	:	I	
ObjectLength	:	268	
Key Values for Buffer Access			
Key field name	Type	Lgth	Key Value
FUNCTNAME	CH	60	RV_KONV_SELECT

Figure 21.25 Buffer Trace

For example, accessing a fully buffered table with a key that isn't part of the primary key will have a negative impact because the system has to scan the records manually. In such a case, you can consider changing the buffering parameters or creating a secondary index.

Overall, the various types of traces we've just discussed allow you to identify any statements that are taking up more time and causing bottlenecks.

21.9 ABAP Trace/Runtime Analysis

ABAP Trace allows you to analyze the performance of your program in the application layer. It helps identify CPU-intensive modules and any unnecessary use of modularization. Whereas a SQL trace is used to analyze the performance of statements that interact with the database, the primary purpose of ABAP Trace is to analyze how CPU-intensive the program is in the application server.

ABAP Trace lets you quickly identify statements that are causing bottlenecks. For example, you can identify the statements that are called repeatedly or statements that are CPU-intensive. In this section, we'll discuss the steps for running and analyzing ABAP Trace.

21.9.1 Running ABAP Trace

You run ABAP Trace using the Transaction SAT (previously Transaction SE30; Runtime Analysis). This tool has been available since SAP NetWeaver 7.0 EHP 2 and it provides many improvements over Transaction SE30. You can check out the release notes for the new tool by visiting http://help.sap.com/saphelp_nw74/helpdata/en/4e/e6b33b556444ccb3ca9ffef95d244/content.htm.

To run ABAP Trace, proceed with the following steps:

1. Open Transaction SAT.
2. On the initial screen of the transaction, as shown in [Figure 21.26](#), you can enter the PROGRAM name, TRANSACTION, or FUNCTION MODULE to evaluate. The EVAL. IMMEDIATELY checkbox allows you to display the results immediately, whereas unchecking this checkbox will allow you to run consecutive tests.

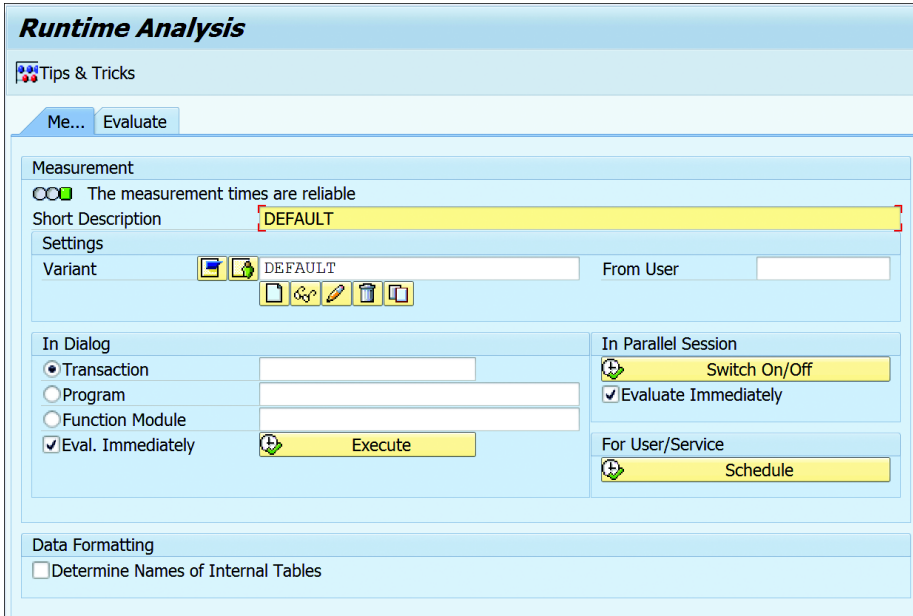


Figure 21.26 Runtime Analysis Initial Screen

3. The MEASUREMENT section shows a traffic light icon to indicate if the measurement runtimes are accurate based on the system configurations. When you open the transaction, the system loads a default variant. You can create your own variant for the test by clicking the CREATE icon under VARIANT. Creating your own variant allows you to tweak the settings.

For example, when performing an initial analysis, you may want to check aggregate runtimes to quickly identify the problem areas as opposed to individual calls to the statements. This is useful if any statements are called repetitively, such as calling a function module in a loop. For such situations, you can select the PER CALL POSITION option under AGGREGATION while creating a variant, as shown in [Figure 21.27](#).

4. Once you provide the program name, transaction, or function module on the initial screen, click the EXECUTE button to run the test in dialog mode. The system will call the program, which you can execute and perform any interactive actions.

Runtime Analysis

Variant: DEFAULT From User

Description: DEFAULT

Duration and T... Statements Program Parts

Size Limits

Maximum Size of File	10,000 KB
Maximum Execution Time	1,800 Seconds

Aggregation

None

Per Call Position

Options

Measure RFC and Update Calls

Explicit Switching On and Off of Measurement

Measure Memory Use

Default Setting

Figure 21.27 Create Variant

- Once the execution is over, click the BACK or EXIT button of the program to go back to the Transaction SAT screen. Based on whether you selected the EVAL. IMMEDIATELY checkbox, either the results are displayed automatically or you can continue to perform consecutive tests.
- Click the EVALUATE tab to see the tests that were carried out and evaluate the same. The TIPS & TRICKS button in the toolbar provides some great examples of how to efficiently use various ABAP statements.

21.9.2 Analyzing the Results

The results screen, as shown in [Figure 21.28](#), has multiple tabs that provide information for you to analyze. For example, you can check the runtime of various modularization units, the number of modularization units, and the percentage of time spent on each modularization unit.

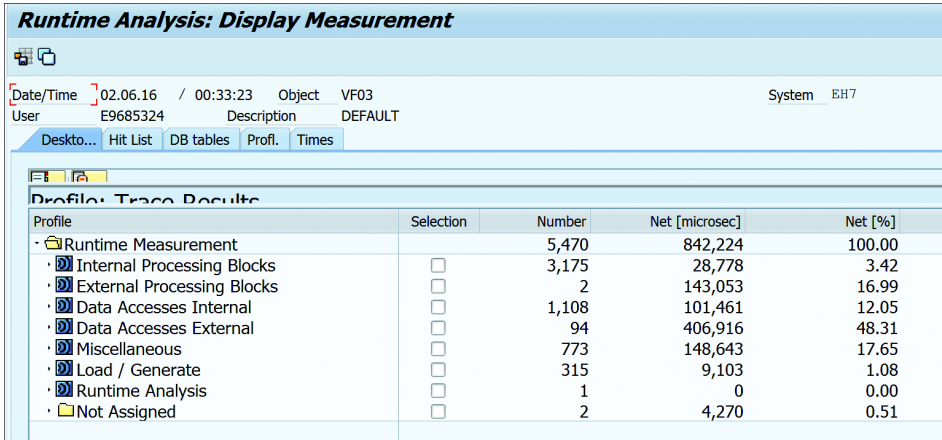


Figure 21.28 Results Screen

There are several tabs available on the results screen:

- ▶ **DESKTOP**
 The DESKTOP tab is default setup with PROFILE and HIT LIST views. It displays the information available separately in the PROFILE and HIT LIST tabs in a single view.
- ▶ **HIT LIST**
 The HIT LIST tab, as shown in [Figure 21.29](#), shows the NET [%] and GROSS [%] times for each statement or event. The entries are sorted by the net runtime by default. This should help you to zero in on problematic areas of the code.

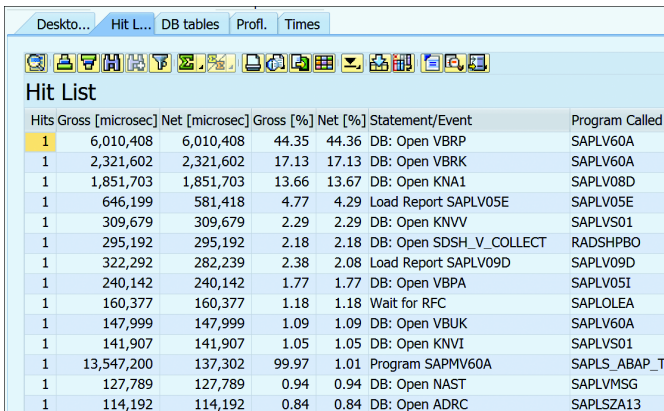


Figure 21.29 Hit List

When you double-click an entry in the HIT LIST tab, the system will open the statement belonging to the corresponding program in the appropriate ABAP Workbench tool. This helps you to check the actual statement in the program that corresponds to the entry. The value in the HITS column gives you insight into how many times a particular statement is being executed.

► DB TABLES

The DB TABLES tab shows information about the time spent on database access (see [Figure 21.30](#)). In this view, you can analyze different information about database access, such as the database tables or views that were accessed, the number of times each table or view is accessed, if a buffer is used, the type of buffer used, and the time spent on each table access.

Table Name	Access Type	Accesses	Gross [microsec]	Buffering	Buffering type	Table category
SOTR_HEAD	Open SQL	1	2,675	On	Single Record	TRANSP
SRAL_CONFIG_RT	Open SQL	1	1,207	On	Generic	TRANSP
AGR_USERS	Open SQL	1	861	(Off)	Generic	TRANSP
/PERSONAS/C_MDUI	Open SQL	2	753	(Cust)		TRANSP
TRDIR	Open SQL	1	752			VIEW
/PERSONAS/C_SCOV	Open SQL	1	512			TRANSP
/PERSONAS/USFLAS	Open SQL	1	426			TRANSP
/PERSONAS/C_CLOV	Open SQL	1	411			TRANSP
/PERSONAS/FLARES	Open SQL	1	377			TRANSP
SOTR_TEXT	Open SQL	1	234	On	Generic	TRANSP
DSHCONFIG	Open SQL	1	56	On	Complete	TRANSP

Figure 21.30 DB Tables View

► PROFL.

The PROFL. (profile) tab allows you to look at the results in a different perspective. Here, you can see the details categorized by processing blocks. Click on the OTHER HIERARCHY button in the toolbar (see [Figure 21.31](#)) to change to a different grouping, including TRACE RESULTS, PACKAGES, COMPONENTS, PROGRAMS, and SLAD OBJECTS. This allows you to broadly identify which program's or package's statements are creating problems.

► TIMES

The TIMES tab provides information about the time spent for each statement or event (see [Figure 21.32](#)). In this view, you can check the time spent by the user and the system for each statement or event. This helps you analyze if performance issues are caused by the program or by the system.

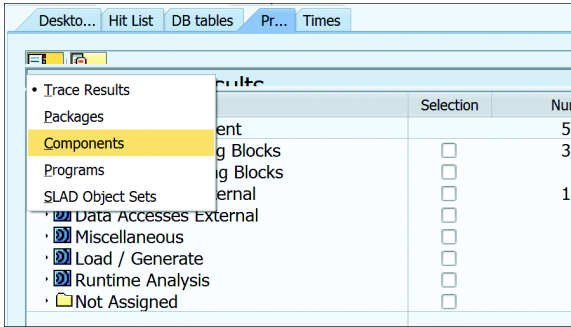


Figure 21.31 Profile View

The screenshot shows the SAP Times View interface. The top navigation bar includes 'Deskto...', 'Hit List', 'DB tables', 'Prof.', and 'Tim...'. Below the navigation bar is a toolbar with various icons. The main area displays a table titled 'Times' with the following data:

Index	Statement/Event	Gross [microsec]	Net [microsec]	Usr.T.Tot	Sys.T.Tot.
1	Runtime analysis	57,570			
2	DB: Open REPOLOAD	555	555	555	
3	DB: Fetch REPOLOAD	173	173	173	
4	Load Report ZTEST_PROGRAM	107	57	57	
5	DB Buffer: Get TABLP GET	50	50		50
6	Dynpro Entry	8,265	93		93
7	Load Dynpro SAPMSSY0 1000	77	50		50
8	Load Report SAPMSSY0	27	27		27
9	PBO Dynpro SAPMSSY0	20,914	122		122
10	Load Report SAPMSSYD	18	18		18

Figure 21.32 Times View

21.10 Single-Transaction Analysis

Single-transaction analysis (Transaction ST12) allows you to combine ABAP Trace with the performance traces from Transaction ST05 (i.e., SQL, RFC, enqueue) to make the tracing and analysis process simpler and convenient. This tool isn't exactly developed for customers; it's intended for SAP's own use to help SAP employees with service delivery. Transaction ST12 combines the options available in Transactions ST05 and SAT.

Figure 21.33 shows the initial screen of Transaction ST12. The TASK FOR section allows you to select how you want to run the trace. Here, the following options are available:

► USER/TASKS

If you select USER/TASKS, you can activate the trace for a particular username. It will trace the next actions performed by the user. Here, you can provide the username and select the task type from the TASKTYPE dropdown. The task type allows you to restrict the trace to a particular task type, such as DIALOG, BACKGROUND, UPDATE, INCOMING RFC, HTTP, and so on.

Single transaction analysis

Trace & collect

Traces on/off Collect ext. traces Statistical records

Trace for User / Tasks Workprocess Current mode Schedule >

Comment

Server

Username (Perf.trace for user)

Tasktype No. trace activations: 5

Start trace

ABAP trace

Options Particular units Further opt.

with internal tables

Size&Duration Large (20 MB / 4200 sec max.)

Clock type Auto High / Low

Performance traces

SQL RFC Enqueue

Statistical records (top 20)

Context trace (RFC,Upd)

Collected trace analyses

Selected trace analysis

State	Finished&col	Size	88,246
Startdate	03.10.2013	Time	13:16:55
Comment	Status 1		
Content	ATRA SQL SSUM ENQ RFC STAT [User=		
Duration	418	Tasktype	<input type="text"/>

Trace analyses Full screen

	Collect date	No#	S.. Trc.time	User	Cor
<input type="checkbox"/>	03.10.2013	1	✓ 13:16:55	I302319	Sta
<input type="checkbox"/>	27.03.2012	1	✓ 04:20:28	MJORDAN	MC
<input type="checkbox"/>	06.09.2011	2	✓ 23:13:32	I812195	test
<input type="checkbox"/>	06.09.2011	1	✓ 23:12:21	I812195	test
<input type="checkbox"/>	27.04.2009	1	✓ 10:55:43	WEISSM	

Evaluate ABAP trace Performance traces SQL summary Stat. records

Figure 21.33 Initial Screen of Transaction ST12

► WORKPROCESS

The WORKPROCESS trace allows you to trace parts of long-running programs, similar to using Transaction SM50.

► CURRENT MODE

The CURRENT MODE option, as shown in [Figure 21.34](#), allows you to run the trace for a specific program or transaction in current mode. This is similar to

how the task is run in Transaction SAT and requires test data to run the application to be monitored.

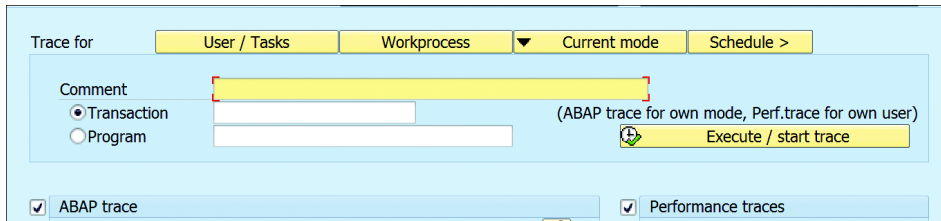


Figure 21.34 Current Mode

You can specify the trace options in the middle section of the screen shown in [Figure 21.33](#). The following are the important options to note:

▶ **WITH INTERNAL TABLES**

The **WITH INTERNAL TABLES** checkbox allows you to measure the performance of internal tables by evaluating statements such as `LOOP` or `READ`.

▶ **SIZE & DURATION**

This option lets you set the size of the trace and the duration of execution; the trace will stop once this value is reached. This option can be tweaked if you're tracing a long-running program, but otherwise leave it set to the default value.

▶ **CLOCK TYPE**

This option influences the time accuracy of the recording and is operating system-dependent. Select the **AUTO** checkbox to use the right mode for your system automatically (either high or low, depending on your operating system).

Once the trace is complete, Transaction `ST12` collects the trace files from the local file system and stores them in the database as a trace analysis. The available trace analysis files are displayed in the **COLLECTED TRACE ANALYSES** section ([Figure 21.33](#)). In this section, you can find the trace analysis files and information about the trace files, such as their status, size, duration, and start dates and times. You can select the trace analyses file and click the **EVALUATION** button to evaluate the trace in the appropriate tool. For example, clicking **PERFORMANCE TRACES** will open the trace for evaluation in Transaction `ST05`.

You can call ABAP Trace for evaluation by clicking the **ABAP TRACE** button. By default, the ABAP Trace display shows the per call statistics, as shown in [Figure 21.35](#). You can select the **PER MODUNIT** button in the toolbar to list the calls per

modularization unit. A bar chart will then show the amount of time spent at various levels, such as ABAP, DATABASE, and SYSTEM. The bar for the time spent in ABAP shows how much time it took to process the program statements. The DATABASE bar displays the time spent in database access.

The SYSTEM bar shows the amount of time spent by the system to carry out system functions such as loading programs; the time spent here should be negligible. If not, it may indicate a problem with the system that should be taken care of by the system administrator.

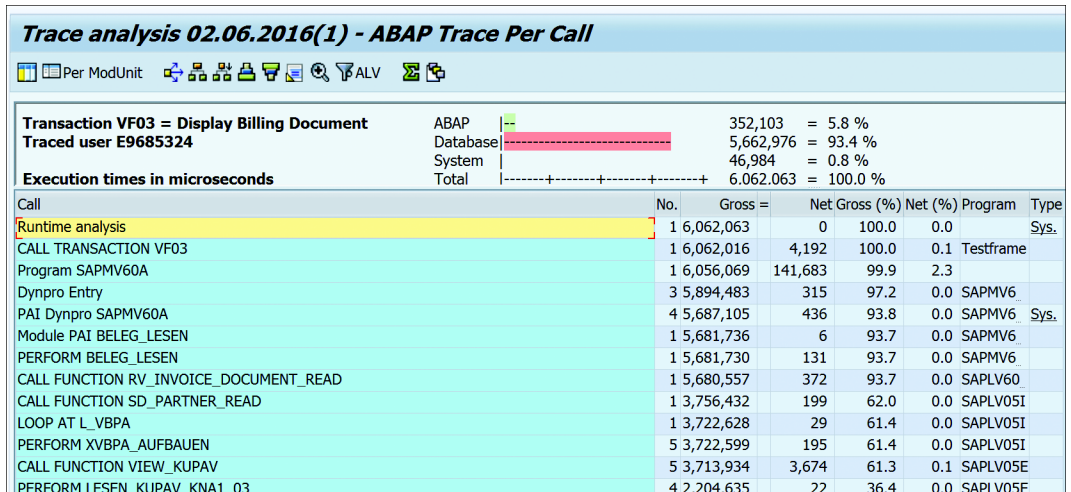


Figure 21.35 Trace Analysis

21.11 Dump Analysis

When a program encounters a runtime error, it generates a short dump. These short dumps can be accessed via Transaction ST22. When a short dump is generated, the system provides some useful information to help identify the problem. For example, we can use the memory information to identify the memory size the program occupied at the time of the runtime error.

When you open the short dump in Transaction ST22, the system presents a tree structure in which you can analyze various aspects of the error, as shown in [Figure 21.36](#).

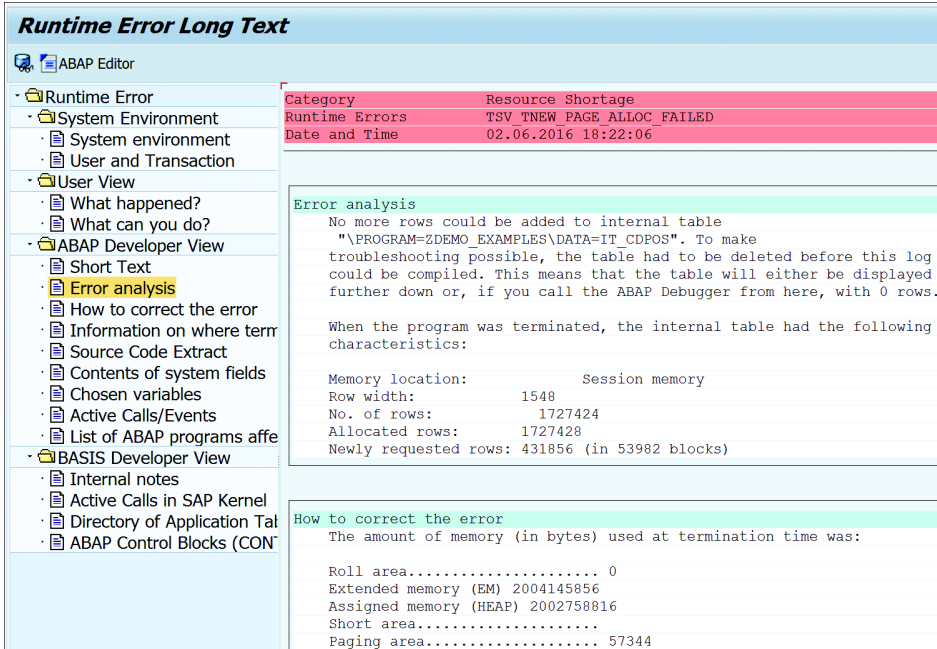


Figure 21.36 Dump Analysis

As shown, the different views are targeted for different users:

► **USER VIEW**

The USER VIEW provides information to the user so that he or she can troubleshoot a problem or contact the system administrator. This view provides options that can explain what went wrong and what steps the user can take to fix the issue.

► **ABAP DEVELOPER VIEW**

The ABAP DEVELOPER VIEW provides various information about the error that helps identify which part of the source code is causing the error, what the contents of the system fields are at the time of the error, the memory occupied by the program, a list of programs affected, the source code extract responsible for the error, and information about where the program terminated.

The contents of the system fields will help you identify details such as how many records were fetched from the database (`sy-dbcnt`) or in which loop of the internal table the error occurred (`sy-tabix`), based on context. The memory

information is useful to troubleshoot memory overflow situations in which the data being fetched from the database is more than the internal table can accommodate.

► **BASIS DEVELOPER VIEW**

The **BASIS DEVELOPER VIEW** provides technical details about the system that SAP Basis administrators can analyze.

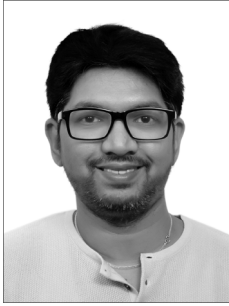
If you're an ABAP beginner, you may come across many runtime errors while you're developing programs. When you encounter a runtime error, go through the complete error in detail, because it can provide quite a lot of information, including suggestions to fix the error.

21.12 Summary

In this chapter, we discussed the tools available in our arsenal for troubleshooting problems with our code. These tools should help you to stop using guesswork and provide you with pointers to identify the problem areas in your code. We may not use all the tools every time; most of the time, a debugger is enough to identify problem areas. Nevertheless, knowing how to use a variety of tools can help unravel trickier problems.

You started your journey as a beginner, but worked your way up through some advanced concepts as we progressed in this book. We hope you enjoyed reading this book as much as we enjoyed creating it for you! The topics in this book should give you a good foundation to explore ABAP even further as you progress through your career as an ABAP consultant.

The Author



Kiran Bandari is a solution architect for one of the world's leading MNCs, and has been working with ABAP for more than 10 years. He has worked as a lead ABAP consultant on multiple SAP implementations, roll outs, and upgrade projects with a specific focus on custom development using ABAP Objects, Web Dynpro ABAP, and SAPUI5. He is also an industry trainer and has conducted ABAP training workshops for major clients like IBM, Accenture, CapGemini, and more.

Index

\$TMP, 147

A

ABAP, 43

- additions*, 110
- conversions*, 45
- environment*, 77
- extensions*, 46
- forms*, 47
- interfaces*, 44
- Reports*, 43
- RICEF*, 43
- statements*, 110, 141
- System requirements*, 48

ABAP CDS views, 396, 426, 434

- access*, 437
- create*, 434
- data definition*, 435
- define*, 434
- replacement objects*, 438
- template*, 436

ABAP channels, 913, 915

- consumer programs*, 917
- producer program*, 916

ABAP consultants, 38

ABAP Data Dictionary, 77, 86, 96, 395, 469, 535

- ABAP CDS views*, 427
- BAPI*, 827
- create data element*, 130
- create domain*, 135
- data types*, 438
- database tables*, 396
- DDL*, 473
- domains*, 451
- elementary search help*, 457
- global table types*, 282
- GTT*, 401
- I_STRUCTURE_NAME*, 629
- interface*, 779
- lock objects*, 465

ABAP Data Dictionary (Cont.)

- Modification Browser*, 932
- modifications*, 929
- non-generic data type*, 216
- objects*, 97
- screen*, 96, 395
- search helps*, 455
- selection texts*, 231
- smart forms*, 754
- table controls*, 538
- type groups*, 450
- type pools*, 574
- views*, 426

ABAP Debugger, 699

- learning tool*, 721

ABAP Developer View, 1018

ABAP Development Tools (ADT), 77, 101, 434

ABAP Dispatcher, 55, 56

ABAP Editor, 77, 88, 145, 177, 571, 893

- back-end editor*, 89
- creating variants*, 609
- front-end editor (new)*, 88
- front-end editor (old)*, 88
- modify programs*, 925
- program attributes*, 143
- settings*, 89
- your first program*, 143

ABAP in Eclipse

- ABAP CDS views*, 434

ABAP keyword documentation, 113

ABAP Managed Database Procedures (AMDP), 700, 718

ABAP messaging channels, 913

- create*, 914

ABAP Object Services, 492, 508

ABAP Objects, 244, 260, 274, 305, 967

ABAP Objects Control Framework, 561

ABAP processor, 511

ABAP programming environment, 78

ABAP programs, 107, 515

- declaration*, 247
- global declarations area*, 108
- procedural area*, 108, 109

Index

- ABAP programs (Cont.)
 - statements*, 253
 - structure*, 108, 245
- ABAP push channels, 913, 1000
- ABAP runtime environment
 - event blocks*, 255
 - processing blocks*, 243, 248
 - selection screens*, 601
- ABAP statements
 - attributes*, 144
 - call*, 142
 - control*, 142
 - declarative*, 142
 - modularization*, 142
 - OpenSQL*, 143
 - operational*, 142
 - processing blocks*, 243
- ABAP System Central Services (ASCS), 55
- ABAP Trace, 975, 1006, 1009
 - analysis*, 1017
 - analyze results*, 1011
 - call*, 1016
 - results screen*, 1012
 - run*, 1009
- ABAP Unit, 974, 976
 - code example*, 986
 - eliminate dependencies*, 977
 - fixture*, 981
 - implement mock objects*, 979
 - predefined methods*, 981
 - SETUP*, 981
 - TEARDOWN*, 981
 - unit tests*, 980
- ABAP Workbench, 47, 76, 77, 85, 87, 284
 - ABAP Data Dictionary*, 396
 - Class Builder*, 91
 - Function Builder*, 90
 - Menu Painter*, 95
 - Object Navigator*, 98
 - objects*, 147
 - Screen Painter*, 92, 509
- abap_false, 334
- ABSTRACT, 342
- Abstract classes, 341
- Abstract methods, 341
- Access control, 473
- ActiveX, 630
- Actual parameters, 275
- Add<subject>0, 824
- Additions, 110, 113
- Address
 - elements*, 759
 - node*, 789
- Adobe Document Server (ADS), 777
- Adobe LiveCycle Designer, 777
 - Layout tab*, 793
- Advance Business Application Programming (ABAP), 43
- Agent class, 495, 498
- Aggregation functions, 477, 478
- ALE, 841
 - inbound process*, 843
 - layers*, 842
 - outbound process*, 842
 - RFC destination*, 858
 - system configuration*, 856
 - tRFC*, 859
- Aliases, 359, 361
- Alternative node, 792
- ALV, 993
 - block display*, 636
 - CL_GUI_ALV_GRID*, 625
 - components*, 626
 - Control Framework*, 650
 - display*, 670
 - dynamic subroutine*, 644
 - grid display*, 630, 631
 - hierarchical sequential display*, 640
 - layout*, 636
 - library*, 625
 - list and grid display*, 627
 - object model*, 625
 - object-oriented*, 661
 - parameters*, 629
 - reports*, 625
 - reuse library*, 626
 - simple reports*, 628
 - standard reports*, 626
- ALV grid control, 561, 651, 652
- ALV object model, 261, 653, 666
 - classes*, 653, 654
 - default status*, 658

- ALV object model (Cont.)
 - hierarchical display*, 655, 657
 - set SAP GUI status*, 655
 - table display*, 654
 - tree display*, 660
 - tree object model*, 659
 - AMDP Debugger, 699, 700, 718
 - breakpoints*, 719
 - capabilities*, 719
 - prerequisites*, 718
 - American Standard Code for Information Interchange (ASCII), 118, 119
 - Analysis, 973
 - Anomalies, 193
 - Anonymous data objects, 682, 683
 - access components*, 684
 - Any
 - fields*, 171
 - tables*, 170
 - API, 332, 803
 - APPEND, 175
 - Append structure, 160, 424
 - add*, 424
 - Application flow, 566
 - Application layer, 49, 51, 54, 75, 246
 - Application Link Enabling (ALE), 835
 - Application server, 54, 471
 - components*, 55
 - files*, 499
 - Application toolbar, 515
 - Architecture, 49
 - Arithmetical operations, 484
 - Array fetch, 199
 - ASSIGNING, 676
 - Assignment operators, 110
 - asXML, 912
 - Asynchronous data update, 812
 - AT LINE-SELECTION, 259, 586
 - AT SELECTION-SCREEN, 255, 604
 - ON <field>*, 604
 - ON BLOCK*, 605
 - ON END OF <sel>*, 605
 - ON EXIT-COMMAND*, 606
 - ON HELP-REQUEST FOR <field>*, 606
 - ON RADIO BUTTON GROUP*, 605
 - OUTPUT*, 604
 - AT USER-COMMAND, 586
 - Attributes, 306, 308, 312
 - Authorization group, 415
 - Authorization objects, 506, 507, 508
 - Automatic type conversion, 124
 - Automation Controller, 561
 - Average, 478
 - AVG, 478
- ## B
-
- Background mode, 812
 - Background processing, 601
 - executing programs*, 619
 - BAdI Builder, 948
 - BAdIs, 933, 946
 - call*, 959
 - create*, 950
 - create definition*, 949
 - create interface*, 954
 - define*, 948
 - definition part*, 946
 - enhancement spots*, 948
 - fallback class*, 958
 - filter values*, 952
 - IF_FILTER_CHECK_AND_F4 interface*, 953
 - implementation*, 955
 - implementation part*, 946
 - BAPI, 261, 819, 820
 - address parameters*, 824
 - BAPI Explorer*, 820
 - business component*, 819
 - business object type*, 820, 830
 - change parameters*, 825
 - class method*, 822
 - conventions*, 827
 - development phases*, 821
 - documentation*, 834
 - extension parameters*, 825
 - implementation*, 826
 - instance method*, 823
 - releasing*, 835
 - return parameters*, 825
 - standardized parameters*, 824
 - test run parameters*, 825
 - tools*, 826
 - use cases*, 819

- BAPI Explorer, 820
 - tabs*, 820
 - Basic lists, 569, 591, 596, 598
 - BASIS Developer View, 1019
 - Batch data communications (BDC), 804
 - batch input*, 807
 - direct input*, 806
 - Batch input, 804, 807
 - BDCDATA, 810
 - CALL TRANSACTION, 812
 - create a material*, 808
 - create program*, 809
 - create program with Transaction Recorder*, 813
 - SESSION, 813
 - update modes*, 814
 - BDCDATA structure, 811
 - Billing document reports, 595
 - Binary search, 173
 - Block display, 627
 - example*, 639
 - function modules*, 636
 - Blocks, 605
 - Boxed
 - structure*, 444
 - types*, 444, 445
 - Breakpoints, 206, 702
 - AMDP Debugger, 719
 - in a call*, 711
 - setup*, 702, 720
 - view*, 706
 - BTEs, 933
 - events*, 969
 - implement*, 967
 - interfaces*, 967
 - test custom function modules*, 971
 - Buffer, 63
 - Buffer trace, 1000, 1008
 - Buffering, 407, 998
 - permissions*, 408
 - Bundling techniques, 488
 - Business Add-In (BAI), 41
 - Business Address Services (BAS), 759
 - Business Application Programming Interface, 804
 - Business component, 819
 - Business Object Repository (BOR)
 - BAPI*, 827
 - Business object type, 820
 - BOR*, 830
- ## C
-
- Calendar control, 561
 - CALL FUNCTION, 243, 292, 691
 - CALL METHOD, 244, 299
 - CALL SCREEN, 517, 557
 - Call sequence, 580
 - Call stack, 708
 - Call statements, 142
 - CALL SUBSCREEN, 547
 - CALL TRANSACTION, 262, 692, 812, 817
 - Calling program, 629
 - Cancel(), 824
 - Candidate keys, 190
 - Cardinality, 422
 - CASE statement, 484, 947
 - Casting, 350, 678, 679
 - implicit or explicit*, 679
 - narrowing cast*, 350
 - widening cast*, 350, 351, 352
 - CATCH, 376, 380
 - Catchable exceptions, 378
 - CDS views, 477, 896
 - Center of Excellence (CoE), 38
 - CHAIN, 549
 - Chained statements, 111, 112
 - Change(), 824
 - CHANGING, 278, 280, 290
 - output parameters*, 279
 - Character
 - data types*, 118
 - fields*, 118
 - literals*, 139
 - CHECK, 249, 251
 - Check table, 418, 419, 421
 - check_count, 334
 - check_status, 334
 - Checkboxes, 527, 531
 - cl_notification_api, 332
 - Class, 294, 308, 321
 - attributes*, 309

- Class (Cont.)
 - constructor*, 316
 - definition*, 309, 311, 331
 - events*, 309
 - global*, 344
 - implementation*, 309, 311
 - methods*, 309
 - pools*, 295
 - private*, 309
 - properties*, 295
 - protected*, 309
 - public*, 309
- Class Builder, 77, 91, 92, 294, 328, 336, 340, 343, 344, 346, 360, 955
 - class pools*, 573
 - components*, 927
 - exception classes*, 384
 - interface pools*, 573
 - Methods tab*, 296
 - modifications*, 927
 - persistent classes*, 494
 - test classes*, 980
 - visibility*, 296
- CLASS C1 DEFINITION DEFERRED, 331
- CLASS C2 DEFINITION DEFERRED, 331
- Class pools, 573
- Class-based exceptions, 375
- Classic BADIs, 947
- Classic Debugger, 205, 699, 700, 703
 - activate*, 701
 - additional features*, 710
 - breakpoints*, 702, 706
 - calls view*, 708
 - execute code*, 704
 - field view*, 705
 - jumping statements*, 710
 - object view*, 711
 - overview view*, 708
 - program status*, 712
 - settings*, 701, 708
 - shortcuts*, 713
 - table view*, 705
 - views*, 704
 - watchpoints view*, 707
- Classic views, 427
- Classical lists, 569
- Classical reports, 592
- Client proxy, 888
- Clients, 64
 - data*, 64, 65
 - specific data*, 64
- CLOSE DATASET, 501
- CLOSE_FORM, 741
- CLUSTD, 505
- CLUSTR, 505
- CMOD enhancements, 936
- Code Inspector, 974, 987
 - results*, 989
 - use cases*, 987
- Code pushdown, 197, 427
- COLLECT, 178
- Collective search helps, 457, 461
 - create*, 461
- Collision check, 467
- Command node, 768
- Comment, 112, 113
- COMMIT WORK, 487, 834
- Comparison operators, 480
- Complex data types, 127
- Component, 151
 - =>*, 312
 - instance*, 312
 - static*, 312
- Componentstyp, 159
- Composer, 725
- Composition, 345
 - relationship*, 345
- Constant window, 727
- Constants, 141
- Constructor, 389
- Consumer program, 919
- Container controls, 561
- Context menu, 523
- Control break statements, 182, 183
 - AT END OF*, 182
 - AT FIRST*, 182
 - AT LAST*, 182
 - AT NEW comp*, 182
 - rules*, 186
- Control Framework, 510, 560, 626, 630
 - ALV*, 650
 - grid display*, 630
 - server and client side*, 561
- Control record, 840

- Control statements, 142
 - Conversion
 - routine*, 135, 137
 - rules*, 122, 124
 - Conversion exits, 452
 - function modules*, 453
 - Conversion logic, 805
 - Conversion programs, 45
 - Conversion routines, 452
 - Core Data Services (CDS), 198
 - Count, 478
 - CREATE OBJECT, 294, 310, 321
 - Cross-client, 64
 - data*, 65
 - CRUD, 903
 - CRUDQ, 895
 - Currency fields, 405
 - Custom code
 - execute*, 416
 - Custom containers, 528, 561, 563
 - Custom controls
 - create*, 562
 - Customer development, 40, 42, 922
 - Customer enhancements, 40, 41
 - Customer exits, 933, 936, 938
 - create*, 939
 - function module exists*, 942
 - function modules*, 936
 - menu exits*, 944
 - screen exits*, 942
 - types*, 937
 - Customization, 40, 42
 - Customizing and development client (CUST), 66
 - Customizing Includes (CI Includes), 160
 - CX_DYNAMIC_CHECK, 380, 382
 - CX_NO_CHECK, 382
 - CX_ROOT, 380
 - CX_STATIC_CHECK, 381
 - CX_SY_, 378
- D**
-
- DATA, 140
 - Data browser, 401
 - Data classes, 406
 - Data clusters, 472, 503
 - administration section*, 503
 - data section*, 503
 - export*, 505
 - exporting to databases*, 505
 - import*, 506
 - media*, 503
 - persistent data*, 471
 - Data Control Language (DCL), 473
 - Data Definition Language (DDL), 96, 188, 473
 - Data definitions, 395
 - Data elements, 130, 131, 403, 439
 - activate*, 133
 - change*, 132
 - domains*, 134
 - global user-defined elementary types*, 130
 - modify*, 930
 - relationship with domains and fields*, 135
 - search helps*, 464
 - Data format, 127
 - external*, 128
 - internal*, 128
 - Data inconsistency, 124
 - Data Manipulation Language (DML), 96, 188, 473
 - Data model definition, 896
 - Data modeling, 294
 - Data node, 792
 - Data objects, 108, 137
 - anonymous*, 682
 - constants*, 141
 - DATA*, 140
 - declaration*, 117
 - field symbols*, 674
 - inline declarations*, 140
 - literals*, 138
 - named*, 682
 - PARAMETERS*, 140
 - predefined types*, 126
 - references*, 681
 - text symbols*, 141
 - user-defined types*, 126
 - variables*, 140
 - Data records, 840
 - Data references, 664, 679
 - debug mode*, 680
 - dereference*, 682

- Data references (Cont.)
 - get*, 681
 - initial*, 680
 - variables*, 680
- Data security, 506
- Data structures, 64
- Data transfer, 805
 - frequency*, 805
- Data types, 97, 115, 118, 131, 404, 438, 439
 - data elements*, 439
 - data format*, 127
 - documentation*, 440
 - further characteristics*, 440
 - output length*, 129
 - structures*, 443
 - tab*, 132, 439
 - table types*, 446
 - value list*, 133
- DATA(..), 301
- Database
 - relationship*, 192
- Database access statements, 143
- Database interface, 511
- Database kernel, 410
- Database layer, 49, 51, 61, 75
- Database locks, 487
- Database LUW, 400, 485, 508
 - database locks*, 487
 - dialog steps*, 486
- Database structure, 188
- Database tables, 156, 198, 396, 472
 - append structures*, 424
 - components*, 398
 - create*, 399
 - Currency/Quantity fields tab*, 405
 - Display/Maintenance tab*, 401
 - enhancement category*, 405
 - Fields tab*, 402
 - fields tab*, 402
 - hints*, 411
 - include structures*, 422, 423
 - indexes*, 409
 - persistent data*, 471
 - SELECT SINGLE*, 198
 - SELECT...ENDSELECT*, 199
 - technical setting*, 399
 - unique and non-unique indexes*, 412
- Database views, 426
 - create*, 427
- Databases
 - fetching data*, 474
 - persistent data*, 472
- Debug session, 718
- Debugging, 152, 205, 699, 719
 - breakpoint*, 206
 - Classic Debugger*, 205
 - exit*, 207
 - New Debugger*, 205
 - troubleshooting*, 719
- decfloat16, 121
- decfloat34, 121
- Declarative statements, 142
- Deep structures, 153
- Default key, 171
- DELETE, 188
- Delete anomaly, 194
- DELETE DATASET, 501
- Delete(), 824
- Delivery classes, 401
- Dependencies, 978
- DEQUEUE, 465
- DESCRIBE LIST, 597
- Desktop, 714
- Destination, 492
- Detail lists, 569, 588, 596
- Dialog box container, 562
- Dialog modules, 243, 246, 255, 256, 260, 512, 516
 - selection screens*, 269
- Dialog programming
 - components*, 515
 - practice application*, 564
- Dialog steps, 246, 486
 - database LUW*, 486
- Dialog transactions, 514
- Diamond problem, 357
- Direct input, 804, 806
 - manage*, 807
 - programs*, 806
- Dispatch control, 843
- Distributed environment, 841
- Distribution model, 842
- Docking container, 562

Index

- Domains, 98, 134, 191, 451
 - attach to data element*, 137
 - bottom-up approach*, 135
 - create*, 135
 - format*, 452
 - output length*, 137
 - relationship with data elements and fields*, 135
 - top-down approach*, 135
 - Downcasting, 684, 685
 - Drilldown reports, 569, 588
 - Dropdown list boxes, 527
 - Dual-stack system, 55
 - Dump analysis, 975, 1017, 1018
 - views*, 1018
 - Dynamic binding
 - CALL method*, 354
 - Dynamic date, 616
 - calculation*, 616
 - selection*, 617
 - Dynamic elements, 237
 - Dynamic enhancement points, 961
 - Dynamic messages, 237
 - Dynamic procedure calls, 693
 - Dynamic program, 211
 - Dynamic program generation, 695
 - persistent program*, 696
 - transient program*, 695
 - Dynamic programming, 171, 663, 664
 - Dynamic RFC destinations, 858
 - Dynamic subroutine pool, 696
 - Dynamic texts, 762
 - Dynamic time, 617
 - calculation*, 617
 - selection*, 618
 - Dynamic token, 664, 691, 693
 - Dynamic token specification, 691
 - Dynamic type, 349
 - Dynpro, 211, 518, 520
- ## E
-
- Eclipse, 77, 99
 - installation*, 100
 - installation wizard*, 102
 - Project Explorer*, 104
 - Eclipse IDE, 434
 - EDI, 836
 - benefits*, 837
 - inbound process*, 840
 - outbound and inbound processing*, 839
 - process*, 837
 - system configuration*, 856
 - using IDocs*, 838
 - Electronic Data Interchange (EDI), 835
 - Element bar, 530
 - Element list, 523
 - Element palette, 529, 530
 - Elementary data types, 115, 216, 439, 443
 - Elementary search helps, 457, 458
 - create*, 457
 - options*, 458
 - Elements, 743
 - addresses*, 759
 - call*, 743
 - graphics*, 758
 - maintain*, 756
 - program lines*, 767
 - tables*, 764
 - text*, 761, 762
 - Encapsulation, 305, 306, 318, 319, 325, 326
 - End users, 37
 - END-OF-PAGE, 585, 586, 598
 - END-OF-SELECTION, 267, 269
 - Enhancement category, 405
 - Enhancement Framework, 947, 960
 - Enhancement packages, 31, 32
 - Enhancement points, 960
 - explicit*, 961
 - Enhancement spots, 948
 - create*, 948
 - ENHANCEMENT-POINT, 961
 - Enhancements, 39, 921, 922, 933, 972
 - assignments*, 940
 - hooks*, 934
 - Enjoy transactions, 83
 - ENQUEUE, 465
 - Enqueue server, 55, 59
 - Enqueue trace, 975, 1000, 1007
 - Enterprise resource planning (ERP), 29
 - ERP systems, 32
 - advantages*, 35
 - departments*, 33

- ERP systems (Cont.)
 - layout*, 35
 - vs. non-ERP systems*, 33
 - Error message, 232
 - Errors, 123
 - Event blocks, 243, 246, 255, 256, 258
 - Events, 261, 312, 322, 416
 - event handlers*, 322
 - instance events*, 322
 - list events*, 270
 - program constructor*, 262
 - reporting events*, 262
 - screen events*, 271
 - selections screens*, 269
 - sender*, 323
 - sequence*, 575
 - static events*, 322
 - Exception classes, 382
 - define globally*, 383
 - define locally*, 386
 - function modules*, 385
 - messages*, 387
 - Exception handling, 284, 369
 - local classes*, 374
 - methods*, 373
 - Exceptions, 374
 - ASSIGN*, 379
 - catching*, 372, 378
 - local classes*, 375
 - maintain*, 371
 - managing via function modules*, 370
 - MESSAGE addition*, 393
 - overview*, 369
 - passing messages*, 389
 - raise*, 372, 374, 376, 385
 - Exclude
 - ranges*, 226
 - single values*, 226
 - Exclusive locks (write lock), 466
 - Executable programs, 145, 211, 260, 262, 571
 - background processing*, 619
 - flows*, 575
 - EXIT, 249
 - Exit message, 233
 - Explicit enhancement points, 961
 - Explicit enhancements, 961
 - Extended Binary Coded Decimal Interchange Code (EBCDI), 118
 - Extends, 407
 - Extensible Stylesheet Language (XSL), 908
 - Extensible Stylesheet Language Transformations (XSLT), 908
 - Extensions, 46
 - External breakpoint, 702
 - External data, 107
 - External developers, 803
 - External program, 326
- ## F
-
- Fallback class, 951, 958
 - implement*, 958
 - FIELD, 548
 - Field attributes, 463
 - Field catalog, 626, 631
 - components*, 632
 - usage*, 634
 - Field conversion, 843
 - Field exits, 938
 - Field help, 555, 556
 - Field labels, 134, 442
 - Field symbols, 664, 665, 671
 - assign data object*, 674
 - assign internal table record*, 677
 - assignment check*, 677
 - define*, 673
 - dynamic field assignment*, 675
 - field positions*, 676
 - filter functionality*, 672
 - generic types*, 673
 - making programs dynamic*, 666
 - modifying an internal table record*, 665
 - static assignment*, 674
 - structure components*, 673
 - structure fields*, 676
 - unassign*, 678
 - Fields
 - relationship with domains and data elements*, 135
 - File interfaces, 498
 - Filtering logic, 333
 - FINAL, 344

Index

- First normal form (1NF), 194
- Fixed point arithmetic, 146
- Fixture, 981
- Flat structures, 153
- Flow logic, 515, 541
 - tab*, 523
- FOR TESTING, 980
- Foreground mode, 812
- Foreground on error mode, 812
- Foreign keys, 189, 191, 200, 398, 418, 472
 - create relationships*, 420
 - field types*, 421
 - relationships*, 418
 - table*, 418
- FORM, 276
- Form Builder, 751
 - draft page*, 752
 - form styles*, 769
 - SAP Interactive Forms by Adobe*, 778, 785
 - smart forms*, 750
 - text modules*, 762
- Form Painter, 727, 729
 - create window*, 733
 - graphical*, 731
 - page layout*, 732
 - paragraph and character formatting*, 734
 - SAPscripts*, 725
- Formal parameters, 275, 279
 - typed and untyped*, 275
- Forms, 47, 723, 729
 - address node*, 789
 - addresses*, 759
 - alternative node*, 792
 - attributes*, 753
 - commands*, 768
 - create*, 785
 - data node*, 792
 - driver program*, 774
 - elements*, 743
 - global definitions*, 755
 - graphic node*, 788
 - graphics*, 758
 - interface*, 754
 - invoices*, 723
 - loops*, 768, 791
 - maintain elements*, 756
 - print*, 742
- Forms (Cont.)
 - program lines*, 767
 - SAPscripts*, 725, 746
 - single-record node*, 793
 - structure node*, 790
 - styles*, 769
 - tab positions*, 746
 - tables*, 764
 - templates*, 760
 - text*, 761
 - text node*, 792
 - windows*, 756
- Free key, 179
- Friends, 329, 331
- FROM clause, 479
- Function Builder, 77, 88, 90, 91, 145, 243,
284, 573, 627, 969
 - Attributes tab*, 288
 - BAPI*, 826, 829
 - Changing tab*, 290
 - create web service*, 884
 - Exceptions tab*, 290
 - Export tab*, 289
 - function module*, 285, 829
 - function modules*, 284
 - Import tab*, 289
 - Source Code tab*, 291
 - Tables tab*, 290
 - update function modules*, 489
- Function groups, 211, 260, 284, 572, 603
 - create*, 285
- Function module exits, 937, 942
- Function modules, 243, 256, 274, 284, 579
 - calling*, 292
 - CLOSE_FORM*, 742
 - CONVERSION_EXIT_MATN1_OUTPUT*, 767
 - create*, 285, 287
 - enhance interface*, 964
 - F4IF_SHLP_EXIT_EXAMPLE*, 464
 - FP_JOB_CLOSE*, 799
 - function groups*, 284
 - modify*, 930
 - normal*, 289
 - OPEN_FORM*, 743
 - Pattern button*, 292
 - remote-enabled*, 289
 - REUSE_ALV_FIELDATALOG_MERGE*, 635

Function modules (Cont.)

- REUSE_ALV_HIERSEQ_LIST_DISPLAY*, 643
 - test in BTEs*, 971
 - update modules*, 289
- Function pool, 243
- Functional consultants, 37
- Functional decomposition, 325

G

-
- Garbage collector, 349
 - Gateway, 55, 58
 - General dynpros, 211, 212
 - General screen, 271, 510, 511
 - Generic types, 276
 - GET
 - <table> LATE*, 266
 - BADI*, 959
 - CURSOR*, 586, 596
 - DATASET*, 501
 - SBOOK*, 267
 - table*, 265
 - GET_SOURCE_position, 381
 - GetDetail(), 824
 - GetList(), 823
 - Getter method, 310, 318, 329
 - get_*, 318
 - Global class, 294, 310
 - Global declarations, 108, 247
 - Global table type, 282
 - Graphic
 - elements*, 758
 - node*, 788
 - Graphical layout editor, 525, 542
 - Grid display, 626
 - GROUP BY clause, 476
 - GTTs, 409
 - GUI status, 513, 515, 550, 587
 - activate and load*, 552
 - create*, 550
 - maintain*, 551
 - GUI title, 552
 - GUI_DOWNLOAD, 502
 - GUI_UPLOAD, 502

H

-
- Hash algorithm, 169
 - Hash keys, 172, 174
 - Hashed tables, 169, 170
 - secondary keys*, 173
 - HAVING clause, 476
 - Help documentation, 134
 - Help views, 426, 433
 - Hexadecimal data types, 118
 - HIDE, 592
 - Hide areas, 592
 - Hierarchical display, 627
 - Hierarchical sequential display, 640
 - field catalog*, 640
 - Hierarchical sequential list
 - example*, 643
 - Hierarchical sequential report, 643
 - Hierarchical structure, 989
 - High-priority updates, 489
 - Hints, 411
 - Hold data, 522
 - Hooks, 934
 - Host variables, 484
 - HTTP trace, 1000

I

-
- I/O fields, 520, 527, 534
 - create/add*, 535
 - IDocs, 844
 - assign basic types*, 855
 - attributes*, 863
 - create basic type*, 851
 - create logical type*, 854
 - create segment*, 849
 - development and tools*, 848
 - EDI*, 838
 - inbound programs*, 864, 865
 - master IDoc*, 842
 - outbound program*, 866, 867
 - records*, 840
 - status codes*, 865
 - structure*, 846
 - system configuration*, 856

- IF_MESSAGE, 379
 - GET_LONGTEXT, 379
 - GET_TEXT, 379
 - Implementation, 52
 - Implementation hiding, 318, 332, 336
 - Implicit enhancement points, 963, 964
 - Implicit enhancements, 963
 - IMPORT/EXPORT, 580
 - Inbound interface, 44, 499
 - Inbound process code, 862
 - Inbound program, 864
 - Include programs, 145, 244, 245, 574
 - Include structures, 422
 - Include texts, 762
 - Index access, 166
 - Index tables, 170
 - Indexes, 399, 412
 - unique and nonunique, 412
 - INDX structures, 504
 - Information hiding, 305
 - Information message, 232
 - Inheritance, 305, 314, 315, 335, 337, 959
 - cl_child, 337
 - cl_parent, 337
 - inheriting from, 315
 - Inheritance relationship, 346
 - Inheritance tree, 344
 - INITIALIZATION, 255, 263, 265, 576
 - Injection, 979
 - Inline declarations, 140, 300, 483
 - assign value to data object, 301
 - avoid helper variables, 302
 - declare actual parameters, 302
 - table work areas, 301
 - Inner joins, 201, 202
 - Input fields, 607
 - Input help, 555, 556
 - INSERT statement, 175, 176, 177, 188
 - inserting a single row, 480
 - inserting multiple rows, 481
 - SY-DBCNT, 480
 - SY-SUBRC, 480
 - Insertion anomaly, 194
 - Instance component, 320, 321
 - Instance constructor, 315, 321
 - Instantiation operation, 683
 - int8, 115, 116
 - Integrated development environment (IDE), 77
 - Interactive reports, 592, 644
 - custom SAP GUI status, 645
 - TOP-OF-PAGE, 650
 - user actions, 649
 - Interface, 357
 - INTERFACE, 357
 - PUBLIC SECTION, 358
 - Interface pools, 573
 - Interface programs, 44, 45
 - Interface work area, 265
 - Interfaces, 803
 - Intermediate Documents (IDocs), 835
 - Internal tables, 151, 164, 167, 282
 - APPEND, 175
 - COLLECT, 178
 - define, 164
 - hashed tables, 169
 - INSERT, 175
 - modifying records, 181
 - processing data, 203
 - reading data, 179
 - small, 175
 - sorted tables, 168
 - usage, 175
 - work areas, 165
 - Internet Communication Framework (ICF), 882
 - Internet Communication Manager (ICM), 53, 55, 56, 882
 - Internet Demonstration and Evaluation System (IDES), 48
 - Internet Transaction Server (ITS), 53
 - INTO clause, 479, 484
 - Invoices, 723
 - iXML, 362
 - iXML library, 364, 366
- ## J
-
- JavaBeans, 630
 - JavaScript Object Notation (JSON), 911
 - Joins, 201, 426, 427
 - conditions, 428

JSON
transformation, 912

K

Key access, 166
 Keywords, 111, 113
 access, 113

L

Languages, 61
 Lazy update, 174
 LEAVE SCREEN, 557
 LEAVE TO LIST-PROCESSING, 591
 LEAVE TO SCREEN, 557
 Legacy System Migration Workbench
 (LSMW), 46, 804
 LfgrpF01, 286
 LIKE, 282
 vs. TYPE, 217
 Line type, 164
 List display, 626
 List dynpro, 212
 List events, 270, 582
 types, 582
 List screens, 148, 270, 510, 569
 list events, 582
 practice, 598
 program types, 570
 List system, 591
 Literals, 138
 Local class, 310
 Local declarations, 108, 247
 Local exception classes, 386
 Local objects, 69
 Local structures, 152
 Lock objects, 97, 465, 506
 code example, 469
 create, 465
 function modules, 467
 maintain, 466
 Logical database, 265
 Logical expressions, 480
 Logical message type, 848, 854

Logical unit of work (LUW), 284, 472, 488
 LOOP, 165, 179, 180, 550
 LOOP AT SCREEN, 622
 Loop node, 768
 Low-priority updates, 489
 LSWM, 867
 field mapping, 875
 getting started, 868
 object attributes, 870
 process steps, 869
 recordings, 871
 reusable rules, 876
 source structure, 873
 LZCB_FGTOP, 286
 LZCB_FGUXX, 286

M

Macros, 244
 Main program group, 579
 Main window, 728, 752
 Maintenance views, 414, 416, 426, 431
 create, 431
 maintenance status, 432
 options, 433
 view key, 432
 Many-to-many relationship, 193
 MAX, 478
 Maximum, 478
 Memory, 578, 581
 allocation, 446
 analysis, 995, 996
 storage, 581
 units, 138
 Memory Inspector, 974, 995
 compare memory snapshots, 996
 memory snapshot, 995
 Memory snapshots, 995
 compare, 996
 steps, 995
 Menu bar, 515
 Menu exits, 938, 944
 Menu Painter, 77, 80, 95, 509
 custom GUI status, 645
 GUI status, 515
 load custom SAP GUI status, 645

Index

- Menu Painter (Cont.)
 - modifications*, 929
 - screen elements*, 513
 - Message, 334
 - Message class, 235
 - IF_T100_DYN_MSG*, 392
 - IF_T100_MESSAGE*, 391
 - maintaining messages*, 236
 - MSGTY*, 392
 - TYPE*, 393
 - using messages*, 390
 - WITH*, 393
 - Message server, 54, 55, 59
 - Messages, 231
 - assigning attributes*, 392
 - dynamic*, 237
 - maintenance*, 235
 - message class*, 235
 - placeholders*, 238
 - statement*, 231
 - tab*, 235
 - text symbols*, 233
 - translations*, 238
 - types*, 231, 232
 - Metadata, 395
 - Methods, 244, 256, 274, 293, 306, 308, 312, 313
 - abstract*, 342
 - CALL METHOD*, 313
 - calling*, 299
 - cl_notification_api*, 332
 - class*, 294
 - create*, 293
 - create global classes*, 294
 - functional method*, 313
 - maintain code*, 298
 - me*, 313
 - set_message*, 332
 - static*, 299
 - MIN, 478
 - Minimum, 478
 - Mock objects, 979, 980
 - Modification Assistant, 923, 924
 - ABAP Data Dictionary*, 929
 - Class Builder*, 927
 - function module*, 930
 - insert*, 926
 - Modification Assistant (Cont.)
 - modifications*, 929
 - programs*, 925
 - replace*, 926
 - reset original*, 931
 - Screen Painter*, 928
 - Modification Browser, 923, 932
 - reset to original*, 931
 - Modifications, 40, 921, 922, 923
 - programs*, 925
 - registration*, 924
 - MODIFY, 181
 - Modularization, 241
 - benefits*, 242
 - include programs and macros*, 244
 - local declarations*, 108
 - processing blocks*, 242
 - Modularization statements, 142
 - MODULE, 523, 548
 - Module pools, 145, 211, 243, 260, 270, 572, 603
 - flow*, 576
 - Modules, 36
 - Multiline comment, 113
 - Multiple selection window, 223
 - tabs*, 227
- ## N
-
- Named data objects, 682
 - Narrow cast
 - child->meth3*, 351
 - parent->meth1*, 351
 - parent->meth2*, 351
 - parent->meth3*, 351
 - Native SQL, 96, 187, 473, 1004
 - GTT*, 401
 - Nested structure type, 158
 - Nested structures, 153
 - New Debugger, 205, 699, 714, 995
 - layout*, 717
 - Memory Inspector*, 995
 - sessions*, 717
 - tools*, 715
 - UI*, 714
 - New projects, 39

NODES, 266
 Non-catchable exceptions, 378
 Nonnumeric data types, 121
 Non-transportable package, 69
 Non-unique index, 412
 Normal function modules, 289
 Normalization, 193
 Numeric data types, 118, 120
 Numeric literals, 139

O

Object Navigator, 69, 77, 88, 98, 514, 893, 948
 ABAP messaging channel, 914
 create screen, 520
 create transaction, 558
 form interface, 778
 module pools, 572
 navigation and tool areas, 98
 web service consumer, 888
 Object palette, 794, 795
 Object reference variables, 310
 Object view, 711
 Object-oriented programming, 244
 Objects, 305, 308
 usage, 309
 OData, 804, 895
 data model definition, 897
 READ, 905
 service creation, 896
 service implementation, 903
 service maintenance, 901
 ON COMMIT, 491
 One-to-many relationship, 193
 One-to-one relationship, 192
 Online text repository (OTR), 387
 OOP, 305
 basics, 305
 introduction, 305
 Open Data Protocol (OData), 895
 OPEN DATASET, 499
 Open Specification Promise (OSP), 895
 Open SQL, 96, 152, 187, 508, 1004
 data in a database, 474
 database, 189

Open SQL (Cont.)
 database relationships, 192
 DDL, 188
 DELETE statement, 482
 DML, 188, 474
 GTT, 401
 INSERT statement, 480
 logical database, 265
 MODIFY statement, 482, 490
 persistent data, 471, 474
 SELECT statement, 475
 selecting data from database tables, 198
 selecting data from multiple tables, 200
 statements, 475
 UPDATE statement, 481
 OPEN_FORM, 741
 Operands, 110
 Operational statements, 142
 Optimistic lock, 467
 Oracle, 411
 ORDER BY clause, 476
 oref, 314
 Outbound interface, 44, 499
 Outbound process code, 861
 Outbound program, 866
 Outer joins, 201
 Output length, 129
 Output table, 626

P

Package Builder, 72, 73
 Packages, 68
 assign, 147
 create, 72
 encapsulation, 74
 naming conventions, 71
 nontransportable, 69
 transportable, 68
 Parallelism, 197
 Parameter table, 694, 695
 PARAMETERS, 140, 148, 214, 239
 effects, 215
 SCREEN_OPTIONS, 218
 TYPE_OPTIONS, 215
 VALUE_OPTIONS, 221

Index

- Partner profile, 859
- PC editor, 762
- PDF, 648
- PERFORM, 243, 255, 256, 264, 935
- Performance trace, 975, 1000
 - activate and filter*, 1001
 - with filter*, 1002
- Persistence mapping, 494
- Persistence service, 492, 493
- Persistent attributes, 493
- Persistent classes, 492, 493
 - Class Builder*, 495
 - create*, 493
 - mapping tool*, 496
- Persistent data, 107, 471
 - security*, 506
 - storage*, 471
- Persistent objects, 492, 498
 - working with*, 498
- Persistent program, 696
- Picture control, 561, 562
- Placeholders, 238
- Polymorphism, 305, 315, 347, 362
 - dynamic types*, 348
 - static types*, 348
- Pooled tables, 400
- Port definition, 859
- Predefined elementary ABAP types, 119
- Predefined elementary data types, 115
 - use cases*, 120
- Predefined nonnumeric elementary data types, 116
 - categories*, 118
- Predefined numeric elementary data types, 116
- Predefined types, 446
- Presentation layer, 49, 50, 75, 246, 519
 - files*, 501
 - GUI_DOWNLOAD*, 502
 - GUI_UPLOAD*, 502
 - SAP GUI*, 52
- Presentation server, 471
- Primary index, 409
- Primary key, 191, 447, 472
- Procedural area, 109
- Procedural exceptions, 370
- Procedural programming, 108
- Procedures, 243, 246, 255, 261, 272, 303
- Process after input (PAI), 271, 512, 516, 534, 541, 548
- Process analysis, 974, 992
- Process before output (PBO), 271, 512, 517, 554, 563
 - selection screen*, 604
- Process interfaces, 967
- Process on help request (POH), 272, 512, 555
- Process on value request (POV), 272, 512, 555, 577
- Processing blocks, 109, 142, 241, 242, 246, 254, 303, 571, 572
 - calling*, 248
 - CHECK*, 251
 - dynpro*, 211
 - ending*, 249
 - EXIT*, 249
 - procedures*, 243
 - RETURN*, 252
 - sequence*, 248, 249, 258
 - types*, 243, 255
 - usage*, 253
 - virtual (global data declarations)*, 253
- Production client (PROD), 67
- Production support, 39
- Program
 - RMVKON00*, 611
 - zdemo_prg*, 623
- Program attributes, 143, 570
 - maintain*, 146
 - types*, 145
- Program constructor, 262
- Program execution, 574
- Program groups, 578
- Program line elements, 767
- Program types, 569, 570
- Programming concepts, 107
- Projection views, 426, 429
 - create*, 429
- Promote optimistic lock, 467
- Pseudocomment, 989
- Publish and subscribe interfaces, 967
- Pure Java system, 54
- Push buttons, 527, 533

Q

Quality assurance, 973
 Quality assurance client (QTST), 67
 Quantity fields, 405

R

R_ER, 380
 Radio buttons, 219, 527, 531
 groups, 219
 RAISE, 290, 371
 EVENT, 324
 EXCEPTION, 376
 Range field, 223
 Range tables, 221
 HIGH, 222
 LOW, 222
 OPTION, 222
 SIGN, 222
 values, 226
 Ranges, 224
 operator, 225
 READ, 165, 173, 179
 CURRENT LINE, 596
 DATASET, 500
 LINE, 596
 Receiver, 842
 Receiver determination, 842
 Recording routine, 416
 REDEFINITION, 338
 Refactoring assistant, 346
 Reference data types, 127, 439, 446
 Reference objects, 310
 Reference variables, 680
 assignment, 684
 upcast and downcast, 684
 Referenced data types, 443
 Relational database, 472
 Relational database management system
 (RDBMS), 51, 61, 187, 189
 example, 61
 foreign keys, 51, 191
 primary key, 190
 relational database design, 189
 table, 190
 Remote-enabled function modules, 289
 Remove<subject>(), 824
 Replacement objects, 438
 Report output
 types, 626
 Report transactions, 571
 Reporting events, 262
 END-OF-SELECTION, 267
 GET <table> LATE, 266
 GET table, 265
 INITIALIZATION, 263
 START-OF-SELECTION, 264
 Reporting programs, 559
 Reports, 576
 background execution, 620
 forms, 723
 programs, 44
 run in background, 619
 scheduling, 619
 Repository, 68
 object, 40, 74
 packages, 68
 Repository Browser, 91
 Representational State Transfer (REST), 895
 REQUEST_LOC, 467
 Request-response cycle, 519
 Requests, 70
 RESTful APIs, 895
 RETCODE, 376
 RETURN, 249, 252
 Returning parameters, 297
 REUSE_ALV_BLOCK_LIST_APPEND, 637
 REUSE_ALV_BLOCK_LIST_INIT, 636
 REUSE_ALV*, 626
 RFC, 58, 261, 284
 destination, 857
 trace, 975, 1000, 1006
 RFC/BOR interface, 898
 ROLLBACK WORK, 486, 488
 Row type, 167
 Runtime analysis, 474, 975, 1009
 Runtime Type Creation (RTTC), 685, 687
 dynamic, 689
 Runtime Type Information (RTTI), 685, 686
 query, 687
 Runtime Type Services (RTTS), 664, 685

S

Sales and Distribution (SD), 935

SAP

- data structure*, 40
- Functional Modules*, 36
- history*, 31
- introduction*, 36
- modules*, 36
- systems*, 39
- technical modules*, 37
- users*, 37

SAP Basis, 75

SAP Easy Access, 84

- Favorites*, 80
- SAP Menu*, 79
- User Menu*, 79

SAP ERP, 29, 946

SAP ERP Controlling (CO), 37

SAP ERP Financial Accounting (FI), 37

SAP Gateway, 895

- activate/register an OData service*, 902
- READ*, 905
- Service Builder*, 896

SAP GUI, 50, 52, 77, 78, 245

- architecture*, 53
- for HTML (Web GUI)*, 53
- for the Java environment*, 52
- for the Windows environment*, 52
- set status*, 647
- status*, 645

SAP HANA, 51, 61, 189, 197, 426, 718

- ABAP CDS views*, 198
- aggregate functions*, 477
- code pushdown*, 197
- parallelism*, 197

SAP Interactive Forms by Adobe, 723, 777

- address node*, 789
- alternative node*, 792
- context and layout*, 785
- Context tab*, 787
- currency/quantity fields*, 784
- data node*, 792
- development*, 777
- download as PDF*, 799
- driver program*, 797, 798
- form interface*, 778

SAP Interactive Forms by Adobe (Cont.)

- global definitions*, 783
- graphic node*, 788
- import form data*, 800
- layout*, 794
- Layout tab*, 793
- loop*, 791
- object palette*, 794
- single-record node*, 793
- structure node*, 790

SAP List Viewer (ALV), 624, 625

SAP liveCache, 61

SAP LUW, 485, 487, 508

- bundle with function modules*, 489
- bundle with RFC*, 492
- bundle with subroutines*, 491
- bundling techniques*, 488
- dialog steps*, 488

SAP NetWeaver, 54, 313

SAP NetWeaver 7.5, 36, 858

- debugging*, 700
- global temporary tables*, 400
- new SQL*, 483

SAP Notes, 922

SAP script editor, 737, 739, 762

- printing*, 740

SAP Software Change Registration (SSCR), 924

SAP start service, 55, 59

SAP Support Portal, 924

SAP system

- architecture*, 49
- enqueue server*, 59
- environment*, 78
- gateway*, 58
- instances*, 55
- layers*, 63
- logon*, 78
- message server*, 59
- SAP Web Dispatcher*, 59
- session*, 84
- user context*, 60

SAP Web Dispatcher, 55, 59

SAPscripts, 723, 725

- align and print*, 744
- billing document*, 725
- change header*, 730
- composer*, 725

- SAPscripts (Cont.)
 - create form layout*, 729
 - create standard text*, 741
 - create window*, 733
 - disadvantages*, 749
 - driver program*, 727, 747, 774
 - elements*, 743
 - field entries*, 737
 - formatting*, 729, 737
 - graphics administration*, 736
 - insert graphic*, 739
 - layout*, 725
 - layout designer*, 730
 - maintain tab positions*, 745
 - maintain window details*, 736
 - paragraph and character formatting*, 734
 - print logo*, 736
 - printing*, 743
 - process forms with function modules*, 741
 - return_code*, 748
 - subroutines*, 727
 - us_screen*, 748
 - windows*, 727
- SAPUI5, 1000
- Schema, 188
- Screen, 79, 515, 520
 - application toolbar*, 81
 - attributes*, 521, 523
 - command field*, 82
 - create*, 520
 - events*, 271, 510, 512
 - exits*, 938, 942
 - groups*, 523
 - menu bar*, 80
 - number*, 521
 - processor*, 511, 512
 - sequence*, 556
 - standard toolbar*, 81
 - status bar*, 82
 - title bar*, 81
 - Transaction codes*, 82
 - types*, 522
- Screen elements, 510, 513, 515, 525, 528
 - basic screen elements*, 529
- Screen fields, 520
 - modifying dynamically*, 553
- Screen flow logic, 511, 513, 517, 518, 548, 928
- Screen Painter, 77, 92, 93, 231, 509, 522, 560, 568
 - alphanumeric layout editor*, 525
 - dynpro*, 211
 - graphical layout editor*, 94, 525, 530
 - modifications*, 928
 - module pools*, 572
 - screen elements*, 513
 - subscreens*, 545
 - tab*, 525
 - tabs*, 93
- Screen structure, 554
 - components*, 554, 621
- SCREEN_OPTIONS, 218
 - AS CHECKBOX*, 219
 - AS LISTBOX VISIBLE LENGTH vlen*, 220
 - RADIOBUTTON GROUP*, 219
 - VISIBLE LENGTH vlen*, 218
- SCREEN-OPTIONS
 - NO-DISPLAY*, 218
 - OBLIGATORY*, 218
- Search helps, 134, 440, 455
 - assign*, 463
 - change*, 462
 - exit*, 464
 - parameters*, 460
- Second normal form (2NF), 195
- Secondary index, 410, 413
 - create*, 411
- Secondary keys, 172, 190, 447, 449, 472
- Secondary list, 590
- Secondary window, 752
- Security, 506
- Segment
 - add*, 852
 - create*, 849
 - definition*, 850
- Segment Editor, 848, 851
- Segment filtering, 843
- Segments, 847
- SELECT, 177, 188
- Select
 - ranges*, 224
 - single values*, 224
- SELECT clause, 476

Index

- SELECT statement, 475
 - FROM clause*, 479
 - INTO clause*, 479
 - SELECT clause*, 476
 - WHERE clause*, 480
- Selection screen events, 269
- Selection screens, 148, 212, 510, 601
 - calling programs*, 623
 - create variants*, 609
 - define*, 601, 602
 - dynamic date*, 616
 - dynamic time*, 617
 - dynamically display/hide screen elements*, 621
 - events*, 601, 604
 - fields*, 213
 - PARAMETERS*, 214
 - radio buttons*, 607
 - standard*, 602
 - standard and user-defined*, 602
 - tasks*, 212
 - user-specific variables*, 618
 - variant attributes*, 613
 - variants*, 608
- Selection texts, 230
- SELECTION-SCREEN, 212, 229
 - BEGIN OF BLOCK*, 229
 - SKIP*, 229
 - ULINE*, 229
- Selectivity analysis, 974, 990
- SELECT-OPTIONS, 185, 221, 228, 239
 - multiple selection window*, 223
- Semantic attributes, 439, 442
- Separation of concerns, 978
- Sequential data, 154
- Serialization, 843
- Service implementation, 896, 903
- Service maintenance, 896, 901
- SESSION, 813, 817
 - create manually*, 818
- Session breakpoint, 702
- Sessions, 84, 717
 - components*, 717
- SET DATASET, 501
- SET HANDLER, 323
- SET SCREEN, 557
- Setter method, 310, 318, 329
 - set_*, 318
- Setter method (Cont.)
 - set_message*, 333
- Shared lock (read lock), 466
- Simple Object Access Protocol (SOAP), 881
- Simple report
 - field catalog*, 632
- Simple reports, 627
- Single inheritance, 357
- Single-record node, 793
- Singleton design pattern, 498
- Single-transaction analysis, 975, 1014
- Size category, 407
- SKIP, 270
- Smart forms, 723, 749, 778, 779, 787
 - addresses*, 759
 - advantages over SAPscripts*, 749
 - character formatting*, 772
 - commands*, 768
 - create templates*, 760
 - create text elements*, 763
 - draft page*, 752
 - driver program*, 774, 776
 - form attributes*, 752, 753
 - form interface*, 752, 754
 - Form Painter*, 751
 - global definitions*, 753, 755
 - graphics*, 758
 - interface*, 780
 - layout*, 750
 - line type*, 764
 - loops*, 768
 - maintain elements*, 756
 - maintain global settings*, 753
 - maintenance area*, 751
 - navigation area*, 751
 - paragraph formatting*, 771
 - program lines*, 767
 - row type*, 764
 - styles*, 769, 773
 - tables*, 764
 - text*, 761
 - windows*, 752, 756
- SOAP, 882, 888
- Sorted keys, 172
- Sorted tables, 168, 170
 - secondary keys*, 173
- Special dynpros, 211, 212

- Special screens, 271
 - Splitter containers, 562
 - SQL, 187
 - SQL optimizer, 410
 - SQL trace, 474, 975, 1000, 1004
 - display*, 1005
 - use cases*, 1004
 - views*, 1005
 - SQLScript, 719
 - Standard reports, 626
 - Standard selection screens, 212
 - Standard tables, 167, 170
 - secondary key*, 172
 - Standard toolbars, 515
 - Standardized BAPIs, 823
 - START-OF-SELECTION, 253, 255, 264
 - Statement
 - UNION*, 198
 - Static attribute, 320
 - Static boxes, 444
 - Static components, 321
 - Static constructor, 321
 - Static enhancement points, 961
 - Status message, 232
 - Status records, 840
 - Status types, 550
 - Strict mode, 484
 - STRING, 120, 122
 - String literals, 139
 - Structure node, 790
 - Structure types, 151, 157
 - Structured Query Language (SQL), 187
 - Structures, 151, 152, 443
 - create global structure*, 158
 - define*, 154
 - global*, 153
 - global structures*, 158
 - local structures*, 155
 - processing data*, 203
 - types*, 153
 - usage*, 162
 - use cases*, 163
 - Subclass, 326
 - SUBMIT, 218
 - SUBMIT (doj), 691
 - Subproject, 868
 - Subroutine pools, 573, 579
 - Subroutines, 243, 256, 273, 274, 417
 - error handling*, 375
 - input parameters that pass values*, 279
 - local declaration*, 153
 - output parameters that pass values*, 280
 - parameters passed by reference*, 278
 - passing internal tables*, 280
 - passing parameters*, 277
 - SAP LUW*, 491
 - USING and CHANGING*, 275
 - Subscreen, 545, 603
 - Subscreen area, 528
 - SUM, 478
 - Superclass, 383
 - Switch Framework, 31, 936, 961
 - SY-DBCNT, 480
 - Synchronous data update, 812
 - Syntax, 110
 - chained statements*, 111
 - comment lines*, 112
 - rules*, 110
 - System fields, 177
 - SY-SUBRC, 375, 480
-
- ## T
-
- Table, 97, 266
 - cells*, 766
 - EDID4*, 846
 - EDIDC*, 846
 - EDIDS*, 847
 - fields*, 429
 - ICON*, 82
 - INDX*, 504
 - IT_SFLIGHT*, 182
 - IT_VBRP*, 185
 - line*, 765
 - MAKT*, 63, 398
 - MARA*, 63, 66, 419, 461
 - MARC*, 63, 203
 - NAST*, 749
 - PTAB*, 695
 - SAPLANE*, 419
 - SBOOK*, 267
 - SFLIGHT*, 155, 156, 419, 588
 - SPFLI*, 411, 477, 588
 - T001W*, 204

Index

- Table (Cont.)
 - T100*, 235
 - TVARVC*, 614
 - VBLOG*, 491
 - VBRK*, 185, 436, 595, 726
 - VBRP*, 161, 185, 726
- Table buffer trace, 975
- Table call statistics, 975, 997
 - screen*, 999
- Table category, 167
- Table controls, 528, 537
 - attributes*, 540
 - create*, 538
 - create with wizard*, 542, 543
 - create without wizard*, 538
- Table display, 654
- Table elements, 764
 - areas*, 764
 - table line*, 765
- Table fields, 398
- Table key, 171, 179
 - default key*, 171
 - hash key*, 172
 - secondary key*, 172, 174
 - sorted key*, 172
- Table maintenance generator, 414, 415
- Table Painter, 750
- Table types, 446
 - global*, 282
 - line types*, 446
 - primary key*, 448
- Table view maintenance, 401
- table_line*, 173
- Tabstrip controls, 528, 544
 - create*, 544
 - wizard*, 545
- Tags, 362
- Tasks, 70
- Technical consultants, 37
- Templates, 760
 - create*, 760
- Termination message, 232
- Test classes, 980
 - define*, 980, 982
 - fixtures*, 981
 - implementation*, 984
 - properties*, 981
- Testing, 973
 - results*, 987
- Text
 - elements*, 761
 - modules*, 761
 - node*, 792
- Text field literals, 139
- Text fields, 527
 - create*, 530
- Text literals, 233
- Text symbols, 141, 233
 - change*, 233
- Third normal form (3NF), 196
- Three-tier architecture, 30, 49
 - application layer*, 51
 - buffer*, 63
 - database layer*, 51
 - presentation layer*, 50
- TOP-OF-PAGE, 583, 584, 650
- Total, 478
- Transaction, 82
 - /*xxxx*, 86
 - /h*, 86
 - /i*, 86
 - /IWFND/GW_CLIENT*, 903, 906
 - /IWFND/MAINT_SERVICE*, 901
 - /N*, 85
 - /n*, 86
 - /nend*, 86
 - /nex*, 86
 - /ns000*, 86
 - /nxxxx*, 86
 - /o*, 85, 86
 - /oxxxx*, 86
 - ABAPDOCU*, 382
 - assign*, 558
 - BAPI*, 820
 - BD51*, 863
 - BMV0*, 807
 - CMOD*, 939, 942
 - code*, 518
 - command field*, 86
 - create*, 558
 - custom namespace*, 559
 - DB05*, 974, 990
 - FIBF*, 967
 - FILE*, 501

Transaction (Cont.)

FK02, 971
 LSMW, 868
 ME21N, 83, 943
 ME21n, 577
 ME22N, 83
 ME23N, 83
 MM01, 83, 806, 808, 872
 MM02, 83
 MM03, 83
 MRKO, 575, 611, 612
 NACE, 747, 799
navigating and opening, 83
opening, 85
 RZ10, 58
 S_MEMORY_INSPECT, 995
 S_MEMORY_INSPECTOR, 975, 996
 SA38, 575, 619
 SAMC, 914
 SAP Easy Access screen, 85
 SAT, 474, 975, 1009
 SAUNIT_CLIENT_SETUP, 980
 SCC4, 68
 SCII, 988
 SE01, 70
 SE03, 70
 SE09, 70
 SE10, 70
 SE11, 86, 130, 156, 158, 282, 395, 574
 SE13, 406
 SE18, 948
 SE19, 948, 956
 SE21, 72
 SE24, 92, 294, 310, 360, 383, 980
 SE37, 91, 243, 284, 573, 627
 SE38, 88, 143, 571, 701
 SE41, 80, 513, 645
 SE51, 513
 SE71, 725, 727, 729
 SE78, 736
 SE80, 69, 88, 91, 92, 514, 948
 SE91, 235
 SE93, 69, 82, 514, 518
 SE95, 931, 932
 SEGW, 896, 897
 SHDB, 813, 814
 SICF, 882
 SLDB, 145

Transaction (Cont.)

SM35, 818
 SM37, 991
 SM50, 57, 58, 974, 992, 994
 SM59, 857
 SM66, 974, 992, 994
 SMARTFORMS, 750, 753, 769
 SMOD, 938, 943
 SNOTE, 922
 SO10, 741
 SOAMANAGER, 885, 890
 SPACKAGE, 72
 SPAU, 923
 SPDD, 923
 ST05, 474, 975, 1000, 1001, 1014
 ST10, 975, 997, 998
 ST12, 1014, 1016
 ST22, 1017
 STMS, 70
 STVARV, 614, 616
 SU3, 618
 SWO1, 826, 834
 VA01, 577
 VF01, 83, 518
 VF02, 83
 VF03, 83, 511
 WE20, 859
 WE21, 859
 WE30, 851
 WE31, 848, 849
 WE41, 861
 WE42, 862, 864
 WE81, 849, 854
 WE82, 849, 855
 Transaction Recorder, 813
 create program from recording, 816
 update modes, 814
 Transactional RFC (tRFC), 859
 TRANSFER, 500
 Transient data, 107
 Transient program, 695
 Translations, 238
 Transparent tables, 397
 Transport Organizer, 70
 Transportable package, 68
 TRANSPORTING, 181
 trigger_event, 324

Index

Troubleshooting, 719
TRUNCATE DATASET, 501
TRY, 376
Two-dimensional arrays, 164
TYPE, 114, 216
 c, 125
 concept, 107, 114
 d, 125
 f, 120
 i, 120
 n, 121
 p, 121
 vs. LIKE, 217
Type classes, 686
Type conversions, 122, 124
 with invalid content, 123
 with valid content, 123
Type groups, 450
 data types, 451
Type objects, 686
Type pools, 574
TYPE RANGE OF, 221
TYPE REF TO, 312
TYPE_OPTIONS, 214, 215
 LIKE (name), 217
 LIKE dobj, 216
 TYPE data_type [DECIMALS dec], 216
Typed formal parameters, 275, 276
TYPE-POOLS, 451
TYPES, 156

U

ULINE, 270, 583
UNASSIGN, 678
Uncomment, 113
Undelete(), 824
Unique index, 412
Unit tests, 977
 write and implement, 980
Universal Description, Discovery, and
 Integration (UDDI), 882
Unnamed data objects, 138
Upcasting, 684
UPDATE, 188
Update anomaly, 193

Update function modules, 489
 attributes, 490
Update modules, 284, 289
Update work process, 490
User actions, 649
User context, 60
User exits, 933, 935
 SD, 935
User interaction, 211
User interface, 49
User-defined elementary data types, 115, 125
User-defined selection screens, 212, 602
 define, 603
User-specific values, 618
USING, 277
 input parameters, 279

V

VALUE, 279
Value ranges, 451, 454
Value tables, 420
VALUE_OPTIONS, 221
Variable window, 728
Variables, 140
Variants, 608
 attributes, 611, 613
 create, 609
 dynamic date, 616
 dynamic time, 617
 for selection screen fields, 615
 system variant, 611
 user-specific variables, 618
Views, 97, 396, 426
 ABAP CDS views, 434
 database, 427
 help, 433
 maintenance views, 431
 projection, 429
 replacement objects, 438
 type, 427
Virtual processing blocks, 253
Visibility, 152
Visibility section, 313
 private, 327
 protected, 327

Visibility section (Cont.)
 public, 326
Visibility sections, 326

W

Warning message, 232
Watchpoints, 703, 720
 create, 707
 view, 707
Web Dynpro ABAP, 56, 78, 780
Web services, 804, 880
 consume, 888, 894
 create, 884
 create ABAP program to consume, 893
 maintain port information, 890
Web Services Description Language
 (WSDL), 880
WebSockets, 913
WHERE clause, 475, 480
Windows, 756

WITH, 623
Work area, 164, 530
Work process, 53, 56, 511, 519
 types, 58
WRITE, 125, 270, 599
WRITE_FORM, 741

X

XI Message interface, 882
XML, 911
 array, 911
 if_ixml_element, 364
XML Path Language (XPath), 908
XML schema-based interface, 780
XML transformations, 804
XSL transformation
 deserialization, 910
 serialization, 909
XSTRING, 120, 122

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: editorial@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

Supplements (sample code, exercise materials, lists, and so on) are provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: <http://www.sap-press.com/3921>. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at SAP PRESS, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website <http://www.sap-press.com> provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at <http://www.sap-press.com>.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

© 2017 by Rheinwerk Publishing, Inc., Boston (MA)

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All of the screenshots and graphics reproduced in this book are subject to copyright © SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany. SAP, the SAP logo, ABAP, Ariba, ASAP, Duet, hybris, SAP Adaptive Server Enterprise, SAP Advantage Database Server, SAP Afaria, SAP ArchiveLink, SAP Business ByDesign, SAP Business Explorer (SAP BEx), SAP BusinessObjects, SAP BusinessObjects Web Intelligence, SAP Business One, SAP BusinessObjects Explorer, SAP Business Workflow, SAP Crystal Reports, SAP d-code, SAP EarlyWatch, SAP Fiori, SAP Ganges, SAP Global Trade Services (SAP GTS), SAP GoingLive, SAP HANA, SAP Jam, SAP Lumira, SAP MaxAttention, SAP MaxDB, SAP NetWeaver, SAP PartnerEdge, SAPPHIRE NOW, SAP PowerBuilder, SAP PowerDesigner, SAP R/2, SAP R/3, SAP Replication Server, SAP SI, SAP SQL Anywhere, SAP Strategic Enterprise Management (SAP SEM), SAP StreamWork, SuccessFactors, Sybase, TwoGo by SAP, and The Best-Run Businesses Run SAP are registered or unregistered trademarks of SAP SE, Walldorf, Germany.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.