Horst Keller, Sascha Krüger

ABAP® Objects
ABAP Programming in SAP NetWeaver™

Galileo Press
Bonn • Boston
Contents at a Glance

1 Introduction .................................................. 23
2 A Practical Introduction to ABAP ................. 53
3 Basic Principles of ABAP ............................. 141
4 Classes and Objects ................................. 177
5 Basic ABAP Language Elements ..................... 225
6 Advanced Concepts in ABAP Objects .......... 341
7 Classic ABAP—Events and Procedures ........ 449
8 Error Handling .............................................. 479
9 GUI Programming with ABAP ..................... 513
10 Working with Persistent Data ...................... 705
11 Dynamic Programming ................................. 795
12 External Interfaces ................................. 841
13 Testing and Analysis Tools ......................... 939
# Contents

Foreword ........................................................................................................ 19

## 1 Introduction .................................................................................................. 23

1.1 What Is ABAP? ......................................................................................... 23
   1.1.1 The Evolution of ABAP ................................................................... 23
   1.1.2 Scope of ABAP ................................................................................. 25
   1.1.3 ABAP Development Environment ............................................. 26
   1.1.4 ABAP Programming Model ....................................................... 26
   1.1.5 ABAP and SAP NetWeaver ......................................................... 29
   1.1.6 ABAP or Java? ............................................................................... 30
   1.1.7 ABAP and Java! ............................................................................. 31

1.2 The Objective of This Book ...................................................................... 40
   1.2.1 Target Audience ............................................................................ 40
   1.2.2 Structure of this Book ..................................................................... 41
   1.2.3 Observing Programming Guidelines ...................................... 46
   1.2.4 Syntax Conventions ........................................................................ 47

1.3 How Can I Use This Book on a Practical Level? ................................... 48
   1.3.1 Creating the Examples .................................................................. 48
   1.3.2 Goal of the Examples ..................................................................... 49
   1.3.3 Using the Examples ......................................................................... 49
   1.3.4 Releases Described ........................................................................ 50
   1.3.5 Database Tables Used ..................................................................... 50

## 2 A Practical Introduction to ABAP ............................................................... 53

2.1 Functionality of the Sample Application .............................................. 54
2.2 Getting Started with the ABAP Workbench ........................................ 54
   2.2.1 Entry through SAP Easy Access ................................................. 55
   2.2.2 The Object Navigator .................................................................... 57
2.3 Packages .................................................................................................... 60
   2.3.1 Package for Local Development Objects .................................. 61
   2.3.2 Packages for Transportable Development Objects ..................... 62
   2.3.3 Creating a Package ......................................................................... 63
   2.3.4 Calling the Transport Organizer ............................................... 67
2.4 Database Tables ........................................................................................ 68
   2.4.1 Creating a Customer Table ......................................................... 68
   2.4.2 Creating a Data Element ............................................................. 72
## Contents

### 3.1.2 The Application Server ................................... 142

### 3.1.3 The Application Server ABAP .......................... 143

### 3.1.4 The ABAP Runtime Environment ........................ 150

### 3.1.5 The Text Environment .................................... 151

### 3.2 ABAP Program Organization and Properties ............ 152

#### 3.2.1 ABAP Program Design .................................... 152

#### 3.2.2 ABAP Program Execution ............................... 155

#### 3.2.3 ABAP Program Calls ....................................... 156

#### 3.2.4 ABAP Program Types ..................................... 159

#### 3.2.5 Other Program Attributes ............................... 162

#### 3.2.6 Processing Blocks ........................................... 164

### 3.3 Source Code Organization .......................................... 167

#### 3.3.1 Include Programs ........................................... 167

#### 3.3.2 Macros ........................................................ 170

### 3.4 Software and Memory Organization of AS ABAP ......... 171

#### 3.4.1 AS ABAP as a System ..................................... 171

#### 3.4.2 Application Server .......................................... 171

#### 3.4.3 User Session ................................................ 174

#### 3.4.4 Main Session .................................................. 174

#### 3.4.5 Internal Session .......................................... 175

### 4 Classes and Objects ......................................................... 177

#### 4.1 Object Orientation ..................................................... 177

#### 4.2 Object-Oriented Programming in ABAP ...................... 180

#### 4.3 Classes .............................................................. 182

##### 4.3.1 Global and Local Classes ................................ 182

##### 4.3.2 Creating Classes ............................................. 183

#### 4.4 Attributes and Methods .............................................. 191

##### 4.4.1 Instance Components and Static Components .......... 191

##### 4.4.2 Attributes ...................................................... 192

##### 4.4.3 Methods ........................................................ 194

##### 4.4.4 Using Static Components ............................... 197

##### 4.4.5 Editor Mode of the Class Builder .................... 199

#### 4.5 Data Types as Components of Classes ......................... 200

#### 4.6 Objects and Object References ............................... 202

##### 4.6.1 Creating and Referencing Objects .................. 202

##### 4.6.2 The Self-Reference “me” ................................ 205

##### 4.6.3 Assigning References ............................... 205

##### 4.6.4 Multiple Instantiation ............................... 207
## Contents

4.6.5 Object Creation in a Factory Method .......... 209
4.6.6 Garbage Collection ........................................ 212

4.7 Constructors .............................................................. 213
4.7.1 Instance Constructor ..................................... 214
4.7.2 Static Constructor .......................................... 216
4.7.3 Destructors ................................................... 219

4.8 Local Declarations of a Class Pool ..................... 219
4.8.1 Local Types in Class Pools ......................... 220
4.8.2 Local Classes in Class Pools ....................... 220

4.9 Using ABAP Objects on the AS ABAP ............... 221
4.10 Summary and Perspective ................................. 224

5 Basic ABAP Language Elements ............................... 225
5.1 Data Types and Data Objects ..................................... 225
5.1.1 Data Objects ................................................. 225
5.1.2 Data Types .................................................... 229
5.1.3 Elementary Data Types and Data Objects .......... 236
5.1.4 Structured Data Types and Data Objects .......... 244
5.1.5 Table Types and Internal Tables ..................... 248
5.1.6 Reference Types and Reference Variables .......... 249
5.1.7 Data Types in the ABAP Dictionary ................ 250
5.1.8 Flat and Deep Data Types ......................... 261
5.1.9 Generic Data Types ....................................... 263
5.1.10 Further Details in Data Objects ................. 265

5.2 Operations and Expressions ....................................... 273
5.2.1 Assignments .................................................. 273
5.2.2 Type Conversions .......................................... 274
5.2.3 Special Assignments ...................................... 282
5.2.4 Calculations ................................................... 286
5.2.5 Logical Expressions ........................................ 292

5.3 Control Structures ...................................................... 298
5.3.1 Conditional Branches ..................................... 298
5.3.2 Loops ............................................................ 301

5.4 Processing Character and Byte Strings ................ 303
5.4.1 Operations with Character Strings .......... 305
5.4.2 Find and Replace ........................................... 306
5.4.3 Subfield Access ............................................ 313
5.4.4 Functions for Character String Processing .... 315
5.4.5 Relational Operators for Character String Processing ............... 316
5.5 Internal Tables ............................................................ 318
5.5.1 Attributes of Internal Tables ........................... 319
5.5.2 Working with Internal Tables ......................... 326

6 Advanced Concepts in ABAP Objects ...................... 341
6.1 Method Interfaces and Method Calls ...................... 345
  6.1.1 Parameter Interfaces of Methods .................. 345
  6.1.2 Method Calls ................................................. 355
6.2 Inheritance ................................................................. 359
  6.2.1 Basic Principles .............................................. 359
  6.2.2 Creating Subclasses ........................................ 362
  6.2.3 Visibility Sections and Namespaces in
       Inheritance ..................................................... 364
  6.2.4 Method Redefinition ....................................... 366
  6.2.5 Abstract Classes and Methods ..................... 370
  6.2.6 Final Classes and Methods ............................. 372
  6.2.7 Static Attributes in Inheritance .................... 373
  6.2.8 Constructors in Inheritance ......................... 374
  6.2.9 Instantiation in Inheritance ......................... 380
6.3 Standalone Interfaces ................................................. 381
  6.3.1 Basic Principles .............................................. 382
  6.3.2 Creating Interfaces ......................................... 384
  6.3.3 Implementing Interfaces in Classes ............... 386
  6.3.4 Access to Interfaces of Objects .................... 389
  6.3.5 Access to Static Interface Components .......... 394
  6.3.6 Composing Interfaces ..................................... 394
  6.3.7 Alias Names for Interface Components .......... 397
  6.3.8 Interfaces and Inheritance .............................. 400
6.4 Object References and Polymorphism ......................... 402
  6.4.1 Static and Dynamic Type ................................ 402
  6.4.2 Assignments Between Reference Variables .......... 405
  6.4.3 Polymorphism ................................................ 413
6.5 Events and Event Handling ......................................... 422
  6.5.1 Declaring Events ............................................ 424
  6.5.2 Triggering Events ........................................... 426
  6.5.3 Event Handlers ............................................... 428
  6.5.4 Registering Event Handlers ......................... 431
6.6 Shared Objects ........................................................... 433
  6.6.1 Basics—Areas and Co. .................................... 435
  6.6.2 Accessing Shared Objects ............................. 436
  6.6.3 Creating an Area ........................................... 437
## Contents

6.6.4 Locking ......................................................... 440  
6.6.5 Working with Shared Objects ......................... 441  
6.6.6 Managing Shared Objects ............................... 447  

7 Classic ABAP—Events and Procedures .................. 449  
7.1 Event-Oriented Program Execution .................... 451  
7.1.1 Executable Programs ..................................... 451  
7.1.2 Dialog Transactions ....................................... 457  
7.1.3 Comparison Between Different Types  
of Classic Program Execution ............................ 459  
7.2 Procedural Modularization ................................. 460  
7.2.1 Function Modules ......................................... 461  
7.2.2 Subroutines ................................................... 474  

8 Error Handling .................................................... 479  
8.1 Robust Programs .............................................. 479  
8.1.1 Defensive Programming ................................ 479  
8.1.2 Exception Situations .................................... 480  
8.2 Exception Handling .......................................... 481  
8.2.1 Class-Based Exception Handling .................... 481  
8.2.2 Classic Exception Handling ............................ 500  
8.2.3 Messages in Exception Handling .................... 503  
8.2.4 Combining Class-Based Exception  
Handling and Earlier Concepts ............................ 505  
8.2.5 Runtime Errors ............................................ 508  
8.3 Assertions ....................................................... 508  
8.3.1 Advantages of Assertions ............................... 509  
8.3.2 Using Assertions ............................................ 509  

9 GUI Programming with ABAP ............................. 513  
9.1 General Dynpros .............................................. 515  
9.1.1 Screen ........................................................ 515  
9.1.2 Dynpro Flow Logic ....................................... 517  
9.1.3 Dynpros and ABAP programs ......................... 518  
9.1.4 Dynpro Sequences and Dynpro Calls ............... 519  
9.1.5 Creating Dynpros ......................................... 524  
9.1.6 Dynpro Fields .............................................. 530  
9.1.7 Function Codes and Functions ....................... 534  
9.1.8 Context Menus .............................................. 539
# Contents

10.1.2 Open SQL ..................................................... 710  
10.1.3 Consistent Data Storage ................................. 741  
10.1.4 Special Sections Relating to  
   Database Accesses ......................................... 751  
10.2 Database Access with Object Services ....................... 757  
   10.2.1 Creating Persistent Classes ....................... 757  
   10.2.2 Managing Persistent Objects ...................... 760  
   10.2.3 GUID Object Identity .............................. 770  
   10.2.4 Transaction Service ............................... 771  
10.3 File Interfaces .................................................. 775  
   10.3.1 Files of the Application Server ............... 776  
   10.3.2 Files of the Presentation Server ............. 781  
10.4 Data Clusters ................................................... 784  
   10.4.1 Storing Data Clusters ............................ 785  
   10.4.2 Reading Data Clusters ........................... 786  
   10.4.3 Deleting Data Clusters ......................... 787  
   10.4.4 Example for Data Clusters ...................... 787  
10.5 Authorization Checks .......................................... 789  
   10.5.1 Authorization Objects and Authorizations .... 790  
   10.5.2 Authorization Check .............................. 791  
11 Dynamic Programming ............................................ 795  
   11.1 Field Symbols and Data References .................... 796  
      11.1.1 Field Symbols ........................................ 797  
      11.1.2 Data References ................................... 809  
   11.2 Run Time Type Services (RTTS) ......................... 819  
      11.2.1 Run Time Type Information (RTTI) ............ 820  
      11.2.2 Run Time Type Creation (RTTC) ............... 824  
   11.3 Dynamic Token Specifications .......................... 829  
      11.3.1 Dynamic Specifications of Operands .......... 830  
      11.3.2 Dynamic Specifications of Clauses .......... 830  
      11.3.3 Special Dynamic Specifications of Clauses .... 831  
   11.4 Dynamic Procedure Call .................................. 832  
      11.4.1 Dynamic Method Call ............................ 832  
      11.4.2 Dynamic Function Module Call ............... 835  
   11.5 Program Generation ........................................ 836  
      11.5.1 Transient Program Generation ............... 837  
      11.5.2 Persistent Program Generation .............. 840
Foreword

This book is the sequel to *ABAP Objects: An Introduction to Programming SAP Applications* from the SAP PRESS series. Instead of producing a reworked second edition of the Introduction, we have written a new book that is based, in part, on the manuscript for the previous book.

The earlier book was the first ABAP book in the SAP PRESS series and was intended to serve both as an introduction to ABAP Objects as well as a general overview of the ABAP language and SAP Basis. Since then, however, SAP PRESS has published dedicated introduction and practice books, as well as a comprehensive ABAP reference book, which is complemented by an ABAP quick reference guide to provide a quick overview. This has allowed us to take a new direction in the current book. This book is much less of a reference guide than its predecessor. Instead, it is intended to function as the programming handbook in the series of ABAP books that have previously appeared at SAP PRESS, grouped between introductory practical books and ABAP reference guides.

In this book, we are therefore offering our readers a compendium of modern ABAP programming and of the key possibilities of the ABAP Application Server in SAP NetWeaver. Modern ABAP programming means programming with ABAP Objects. Contrary to the previous book, ABAP Objects are no longer treated as an addition to classical ABAP, but rather as the underlying programming model. Consistent with all books on object-oriented programming languages, the presentation of the ABAP language in Chapter 4 begins this time with “Classes and Objects.” From the start, we have integrated the description of the Class Builder into the description of classes and objects. All remaining language elements and tools have been presented in the same way that they are used in ABAP Objects to implement classes. The classical concepts of ABAP are mentioned where they are still used. We no longer discuss obsolete concepts; and if we do, we only touch on them very briefly.
Whereas in the previous book, we dealt mainly with elementary ABAP language topics, in this book we have also included—in addition to the many new developments that the ABAP language has seen in the meantime—additional topics that are essential for the programming of the ABAP Application Server in SAP NetWeaver. In fact, we devote an entire chapter to the error handling reaching from exception classes to assertions, offer an introduction to Web Dynpro for ABAP, provide a separate chapter on dynamic programming including Run Time Type Creation, and a chapter on the external communication and data interfaces from RFC over ICF to XML, and also provide an overview of all possible tools to use for testing quality assurance.

Because of the large number of new topics, the scope of this book has now passed the magical 1000-page milestone. Therefore, we will at least try to keep the foreword brief, albeit without neglecting to extend our thanks to all of the people who have helped, directly or indirectly, to produce this book.

First we must mention our colleagues in the department “SAP NetWeaver Foundation ABAP.” While this organizational unit had a different name in all of the other books that have appeared to date, it is essentially still the “ABAP Language” group, which develops the ABAP language and the ABAP tools, and which now also encompasses the groups “ABAP Workbench” and “ABAP Connectivity.” This group’s work is the foundation of everything that is described in this book, and we do not exaggerate when we say that the output of this group is the basis of any ABAP developments internationally. In appreciation of all of this team’s members, we would here again like to thank the Vice President Andreas Blumenthal, who has supported this book from the very beginning and provided the necessary resources to make it become a reality.

We would specifically like to thank the following colleagues who have made special contributions to producing this book: Kai Baumgarten (information and corrections on Shared Objects), Thomas Becker (information on qRFC), Joachim Bender and Michael Schmitt (proofreading of the section on Web Services), Dirk Feeken and Manfred Lutz (publication of the AS ABAP Trial Version on DVD), Eva Pflug (help in setting up the AS ABAP trial version as a translation system, to ensure that the examples also work when users log
on in English), Susanne Rothaug and Volker Wichers (support with testing the ABAP Web Services on another J2EE Server), Klaus-Dieter Scherer (help and information on ALV print lists), Stefan Seemann (hooked the MaxDB that failed when we tried to install a parallel J2EE Server backup to the AS ABAP trial version), Markus Tolksdorf (information and corrections on JCo), and Doris Vielsack (information and corrections on dynpros).

As a further new feature, this issue of the ABAP Objects book is also based on texts from authors who are responsible for one particular contribution: Stefan Bresch (object services), Rupert Hieble (XML), Anne Lanfermann (Web Services), Frank Müller (RFC and ICF), and Stefanie Rohland (Web Dynpro). We would like to thank these authors for their readiness to assist with this project, in addition to their normal responsibilities. The authors’ bios are provided at the end of this book.

We would like to thank the publishers at Galileo Press for their collaboration, which was, as always, excellent. Alexandra Müller and Florian Zimniak did an outstanding job correcting and editing the manuscript, even going so far as to find formal errors in the code. For the production, we would like to thank Iris Warkus (Galileo Press) and Dirk Hemke (SatzPro), most especially for the fact that right from the first typesetting for this book, we found nothing of note to grumble about. For the English edition, the authors want to express their gratitude to Nancy Etscovitz from Wellesley Information Services, who did a terrific job in editing the translation, and to Snezhina Gileva from SAP Labs Sofia for proof reading the final manuscript.

Sascha Krüger would especially like to thank his wife Katja for so many things too numerous to mention, both big and small, such as keeping him free from any “distractions,” loads of understanding, constant encouragement and motivation, lots of free space, more than a little coffee, and so much more. In this way, she ultimately played a large part in the production of his share of the manuscript.

Horst Keller would like to thank his wife Ute, as always, who again supported the creation of this book with her considerable patience and understanding. Despite the promises made after every previous book—that next time things would be easier—this book, in particular, again proved that such promises cannot always be kept, and con-
sequently much joint free time during the first half of 2006 had to be sacrificed. The fact that Ute never once questioned this project, but just looked forward with Horst to meeting the deadline, proved to be invaluable.

Walldorf, February 2007

Horst Keller
Sascha Krüger
A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, “Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world.” The civil engineer interrupted and said, “But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth from out of the chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong; mine is the oldest profession in the world.” The computer scientist leaned back in her chair, smiled, and then said confidently, “Ah, but who do you think created the chaos?”

—Grady Booch, Object-Oriented Analysis and Design with Applications

6 Advanced Concepts in ABAP Objects

The above quotation is from a book entitled Object-Oriented Analysis and Design with Applications by Grady Booch (Addison-Wesley 1995), where it is used to introduce a chapter discussing “the inherent complexity of software.” One advantage of the object-oriented approach is its ability to handle complexity. In Chapter 4, you were introduced to classes and objects as a basis for object orientation, and to attributes and methods as underlying components of these classes and objects. We can sum up what you have already learned as follows:

- Objects constitute the key concept in object-oriented programming. An object is a self-contained unit whose status is determined by the values of its attributes, whose behavior is determined by its methods, and whose identity is defined by its address in the memory. An object is accessed by reference variables,
which refer to this address. An object in a program that performs a certain task should reflect a real object of the task 1:1 as far as possible. With objects, a clear distinction can be made between the public interface and the private and protected components, which are not externally visible. One object can interact with another by accessing its attributes directly in a method, calling methods, or triggering an event (see Section 6.5.2).

- Classes consist of source code containing the definition of possible objects. An object is always an instance of a class, which is addressed by at least one reference variable. All components and properties of its objects are declared in a class. The basis for encapsulation in ABAP Objects is always the class, rather than the object.\(^1\) Classes are either global for all programs or local in a single program. They can be specialized by inheritance (see Section 6.2), and can incorporate standalone interfaces as a public interface (see Section 6.3).

- Attributes describe the status of an object. Technically speaking, attributes (instance attributes) are the local variables of an object, which cannot normally be changed directly from the outside. A class may also contain static attributes, which are jointly used by all objects of the class. Static attributes may be variables or constants.

- Methods allow objects to perform operations. A method (instance method) always works in a specific object. In other words, it reads and changes the status of this object, and interacts with other objects by calling their methods or by triggering events. A method has a parameter interface (see Section 6.1.1), and can pass on exceptions (see Section 8.2). A class may also contain static methods, which only access static attributes, and can only trigger static events.

You may have already realized how powerful these components of ABAP Objects can be when used to program application programs; however, there is more to ABAP Objects than just these basic elements. In this chapter, you’ll become familiar with additional concepts that are essential for advanced object-oriented design.

\(^1\) The private components of an object of a class are visible to another object of the same class.
Advanced Concepts in ABAP Objects

- **Method Interfaces and Method Calls**
  Chapter 4 introduced methods in their fundamental role as the operational components of classes. Section 6.1.1 examines the parameter interface of methods in more detail, and focuses in particular on the various options with method calls.

- **Specialization by Inheritance**
  ABAP Objects supports simple inheritance, whereby a class can be declared as the direct subclass of exactly one superclass. All classes of ABAP Objects are part of an inheritance hierarchy tree originating in one common superclass. In addition to its own components, a subclass also contains the components of its superclass. The implementation of superclass methods can be overwritten in subclasses. The concept of inheritance is discussed in Section 6.2.

- **Standalone Interfaces**
  The public visibility section of a class is its external interface. ABAP Objects allows you to create standalone interfaces, which can be used by classes as part of their interface, or even as their complete interface. Objects belonging to various classes that use the same interface can be handled by outside users in the same way. An standalone interface may also comprise several other interfaces. The interface concept is described in Section 6.3.

- **Object Reference Variables and Polymorphism**
  Objects in a program can only be accessed by object references in object reference variables. The type of the object reference variables determines exactly what a program can do with an object. There are both class reference variables and interface reference variables. The latter enable exclusive access to the interface components of a class. The concepts of inheritance and independent interfaces allow you to assign object references between reference variables of different types according to certain rules. This opens up the possibilities of polymorphism, whereby the same reference variable can be used to access objects belonging to different classes with different behavior. This is discussed in Section 6.4.

- **Events and Event Handling**
  A method of an object is normally executed after a direct call. In this case, the calling object and the called object are closely coupled. Events are used to decouple the caller from the called method. In ABAP Objects, events, like attributes and methods, are component type of classes. An object can trigger an event in a
method, and methods of other objects can handle this event. This corresponds to an indirect method call because the calling method does not need to know anything about the possible event handlers. The event concept is described in Section 6.5.

**Shared Objects**

Objects as instances of classes exist in the memory area of a program, and are deleted at the latest when the program is exited. As a result, cross-program access to objects is not generally possible. However, ABAP Objects enables cross-program access with shared objects, which are objects in the shared memory of an application server. The concept of shared objects is discussed in Section 6.6.

The basic concepts of ABAP Objects, which were introduced in Chapter 4 (i.e., classes with attributes and methods, objects, and object references), are used in almost all object-oriented programming languages. The advanced concepts introduced in this chapter comprise, on the one hand, a selection of tried and tested advanced techniques adopted by ABAP Objects based on the standards of well-known object-oriented programming languages like Java or C++, and, on the other hand, specialized techniques that are unique to ABAP Objects. When this language was designed, special care was taken to ensure that the focus on business applications was not lost.

**ASAP principle**

Certain concepts of object-oriented programming, such as multiple inheritance, which is used in C++, for example, would have served only to increase the complexity of the language, without offering any additional benefits for SAP applications. In accordance with the ASAP principle, of “As Simple As Possible,” ABAP Objects was made as easy to understand as possible, and only well-established object-oriented concepts were used. Following the example of Java, the interface concept was introduced in place of multiple inheritance. The correct application of inheritance and interfaces represents the crowning achievement of object-oriented programming, and provides a range of options for managing complexity.²

The range of options for defining a parameter interface for methods is, in contrast, specific to ABAP. Similarly, the concept of fully inte-

² However, we do not wish to conceal the fact that the incorrect use of concepts like inheritance may cause major problems. Meticulous object-oriented modeling is essential, particularly when advanced concepts of object orientation are used to manage complex applications.
grating events into the language scope of ABAP Objects as independent components of classes is not a feature of all object-oriented programming languages.

6.1 Method Interfaces and Method Calls

We have defined and called methods on many occasions in the previous chapters. The next two sections discuss the finer points of methods in ABAP Objects.

6.1.1 Parameter Interfaces of Methods

The parameter interface of a method is defined by the additions to the METHODS and CLASS-METHODS statements when the method is declared, or by the selection of Parameters in the Class Builder. No further details of the parameter interface are required in the implementation section between METHOD ENDMETHOD. However, you can display the interface during implementation of global classes.

The parameter interface of a method comprises formal parameters and exceptions. The declaration of exceptions is discussed in Section 8.2. Formal parameters are keyword parameters, to which an actual parameter can or must be assigned when the method is called. Within a method, formal parameters can be used via their names in operand positions. The possible usage kinds depend on the parameter properties. The following properties can be defined for a formal parameter:

- The parameter type
- Kind of parameter passing
- Parameter typing
- Supply type of the parameter

In principle, a parameter interface can contain any number of parameters; however, a small number is recommended. An ideal parameter interface contains only a small number of input parameters or none at all, and a return value.

At this point, we should point out that methods in ABAP Objects cannot be overloaded. In other words, you cannot use the same method
names with different parameter interfaces, even when you redefine methods in subclasses.

**Parameter Type**

You can define the following parameters:

- **Input Parameters**
  
  Input parameters are specified after the `IMPORTING` addition to the `METHODS` or `CLASS-METHODS` statement, or are declared by selecting `Importing` in the `Type` column on the `Parameters` tab page in the Class Builder. When a method is called, the value of the assigned actual parameter is assigned to the input parameter. Input parameters for which pass by reference is defined cannot be overwritten in the method. Input parameters for which pass by value is defined are not passed to the actual parameter when the procedure is exited.

- **Output Parameters**
  
  Output parameters are specified after the `EXPORTING` addition to the `METHODS` or `CLASS-METHODS` statement, or are declared by selecting `Exporting` in the `Type` column on the `Parameters` tab page in the Class Builder. When a method is called, the value of the assigned actual parameter is not assigned to an output parameter for which pass by value is defined. Output parameters can be overwritten in the method. If the procedure is exited without errors using `ENDMETHOD` or `RETURN`, the output parameter is passed to the actual parameter.

- **Input/Output Parameters**
  
  Input/output parameters are specified after the `CHANGING` addition to the `METHODS` or `CLASS-METHODS` statement, or are declared by selecting `Changing` in the `Type` column on the `Parameters` tab page in the Class Builder. When a method is called, the value of the assigned actual parameter is assigned to the input/output parameter, and, if the method is exited without errors using `ENDMETHOD` or `RETURN`, the input/output parameter is passed to the actual parameter. Input/output parameters can be overwritten in the method.

- **Return Value**
  
  A method can have only one return value, for which pass by value must be declared. This return value can be declared after the
RETURNING addition to the METHODS or CLASS-METHODS statement, or by selecting Returning in the Type column on the Parameters tab page in the Class Builder. A return value is handled in the same way that an output parameter is handled in the method; however, a method with a return value is a functional method, which, in addition to the return value, can have only input parameters. A functional method can be used in operand positions. The return value is then used in these positions.\(^3\)

When you declare a parameter, you must always select the type that matches the behavior of that parameter exactly. A parameter that is received but not changed by the method is an input parameter. A parameter that is output but is not received is an output parameter or a return value. A parameter that is received, changed, and output is an input/output parameter.

This may appear to be stating the obvious, but, as you will see, parameters do not have to behave in accordance with their type.

**Kind of Parameter Passing**

You can define the way a formal parameter is passed either as pass by reference or as pass by value for each individual parameter, with the exception of the return value, for which pass by value is set by default.

The syntax for pass by reference is shown below using the example of an input parameter `ipara`:

```
METHODS meth IMPORTING ipara ...
```

Equally, you can also use:

```
METHODS meth IMPORTING REFERENCE(ipara) ...
```

The syntax for pass by value is shown below using the example of the return value `return`:

```
METHODS meth RETURNING VALUE(return) ...
```

---

\(^3\) Functional methods (as opposed to function modules) are the natural extension of integrated functions (see Section 5.2.4) by self-defined functions in the same way as self-defined data types extend the built-in ABAP types.
In the Class Builder, you define the kind of parameter passing by selecting the **Pass by value** check box on the **Parameters** tab page or leaving this blank. Therefore, pass by reference is the standard transfer type, which is used unless a different type is specified, both in the syntax and in the Class Builder. What is the difference between these transfer types?

- **Pass by Reference**
  
  With pass by reference, a reference to the actual parameter is passed to the method for each formal parameter for which an actual parameter is specified when you call the method, regardless of the parameter type. The method thus uses the actual parameter itself, and changes to formal parameters have a direct effect on the actual parameter.

- **Pass by Value**
  
  With pass by value, a local data object is created as a copy of the actual parameter for each formal parameter when the method is called. In the case of input parameters and input/output parameters, the value of the actual parameter is assigned to this data object. The value of the formal parameter is only assigned to output parameters, input/output parameters, and return values if the method is exited without errors using **ENDMETHOD** or **RETURN**.

The kind of parameter passing is a technical property, which defines the behavior of a formal parameter. Only with pass by value does the actual behavior always correspond to the behavior defined by the parameter type. The following points apply to pass by reference:

- Output parameters are not necessarily initial at the start of the method (output parameters behave like input/output parameters).
- Changes to output parameters and input/output parameters are effective, even if the method terminates with an exception.
- Input parameters that are passed by reference cannot be explicitly changed in the method. Their values may change, however, if they are linked to global actual parameters and if these parameters are changed during the method is executed.

Therefore, a method should always be programmed in such a way that the behavior of its parameters corresponds to the semantics defined by the parameter type:
Do not execute read access to an output parameter that is passed by reference because its initial value is not defined.

If you add lines to an internal table or extend a string that is defined as an output parameter that is passed by reference, you must initialize the parameter before the first access.

Give due consideration to the value you set for output parameters or input/output parameters that are passed by reference before an exception is triggered to ensure that a calling program can execute adequate exception handling.

A number of precautionary methods are thus required for pass by reference, which do not apply to pass by value. So why is pass by reference even necessary? The answer is performance.

In ABAP, pass by reference always performs better than pass by value, because no data object has to be created when a procedure is called, and no data transport takes place. For performance reasons, pass by reference is usually preferable to pass by value, unless explicit or implicit write access to an input parameter is required, or you want to ensure that an output parameter or an input/output parameter is only returned if the procedure is completed without any errors. If possible, these cases should be limited to the transfer of parameters smaller than approximately 100 bytes.\(^4\)

The example in Listing 6.1 is of a small and probably unexpected situation, which may occur if pass by reference is used without due consideration.

Listing 6.1 Transfer Type of Formal Parameters

REPORT z_parameter_passing.

CLASS demo DEFINITION CREATE PRIVATE.

PUBLIC SECTION.
  CLASS-METHODS main.

PRIVATE SECTION.
  METHODS: meth1 IMPORTING value(idx) TYPE i,
            meth2 IMPORTING reference(idx) TYPE i.

\(^4\) With strings and internal tables, the disadvantage in terms of performance of pass by value compared with pass by reference can even be compensated for by the integrated Copy-on-Write semantics (the concept of sharing, see Section 5.1.7). This is the case for input parameters in particular, provided that they are not changed.
DATA msg TYPE string.
ENDCLASS.

CLASS demo IMPLEMENTATION.
  METHOD main.
    DATA oref TYPE REF TO demo.
    CREATE OBJECT oref.
    DO 2 TIMES.
      oref->meth1( sy-index ).
      oref->meth2( sy-index ).
    ENDDO.
  ENDMETHOD.
  METHOD meth1.
    DO 3 TIMES.
      msg = idx.
      CONCATENATE 'meth1: ' msg INTO msg.
      MESSAGE msg TYPE 'I'.
    ENDDO.
  ENDMETHOD.
  METHOD meth2.
    DO 3 TIMES.
      msg = idx.
      CONCATENATE 'meth2: ' msg INTO msg.
      MESSAGE msg TYPE 'I'.
    ENDDO.
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  demo=>main( ).

In the main method, two methods are called with an identical implementation in a DO loop. The first method, meth1, outputs the content of sy-index, which is passed by value, three times as expected, in other words, "1", "1", "1" during the first call, and "2", "2", "2" during the second call. The second method, meth2, outputs "1", "2", "3" during both calls. The DO loop in meth1 and meth2 sets the global system field sy-index and thus also the formal parameter idx (passed by reference) in meth2.

The method with pass by value is therefore more robust. However, this example also shows that global parameters like system fields—changes to which are not subject to the direct control of a method—should not be simply passed to methods in this way from the calling
program. The expected result is also returned by `meth2` if a local auxiliary variable is implemented in `main`, to which `sy-index` is assigned and which is then passed to `meth2`.

**Typing**

You must type each formal parameter of a method. Typing simply means that you assign a type to a formal parameter. As with data declaration, the syntax used for this purpose is a `TYPE` or `LIKE` addition, which you must specify after each formal parameter, for example:

```
METHODS meth EXPORTING opara TYPE dtype ...
```

With local classes, any visible type can be specified here. In the Class Builder, fill the **Typing** (Type, Type Ref To or Like) and Reference type columns accordingly on the **Parameters** tab page in order to specify the type. Since the type you specify must also be accessible to all users of the method, you can only specify built-in ABAP types, global types from the ABAP Dictionary, or types from the public visibility section of a global class for public methods. With protected methods, additional types from the protected visibility section of the class can also be used, while types from the private visibility section and local types from the class pool can be used for private method types only.

The main difference between typing and data declaration is that a formal parameter is assigned its actual data type only when it is linked to an actual parameter when a method is called. All technical properties of the actual parameter are then passed to the formal parameter.

In order for an actual parameter to be passed, its data type must match the typing of the formal parameter. To be more precise, its technical type properties must be compatible with the data type used for the typing. The technical properties of an elementary type are the built-in ABAP type (c, d, f, i, n, p, t, string, x, xstring), the length (for c, n, p, x), and the number of decimal places (for p). The technical property of a structured type is its structure, based on substructures and elementary components (the component names are irrelevant). The technical properties of an internal table are the table type (STANDARD, HASHED, SORTED), line type, and table key.

**Checking typing**
The typing of a formal parameter may be complete or generic. For complete typing, use TYPE to refer to a complete data type, or LIKE to refer to a data object. For generic typing, you can use the built-in generic types (any, any table, c, clike, csequence, data, hashed table, index table, n, numeric, object, simple, sorted table, standard table, table, x, xsequence—see Section 5.1.9). Internal table types are also generic if the table key is not fully defined.

Formal parameters that have complete typing can always be regarded as local data objects of this type, with all type properties known inside the method. Generic types differ in terms of static and dynamic access. The type properties used for typing are only used for static access. With dynamic access, the type properties of the assigned actual parameter are used. These properties may differ from the typing in terms of the non-technical properties, such as component names.

In addition to checking the data type of an assigned actual parameter, the typing defines how the formal parameter can be used as an operand of statements in the method. With one exception, formal parameters can be used in all operand positions that are not excluded by the typing. For example, a generic formal parameter with the typing any can be assigned to any formal parameter that has the same typing. In that case, an exception occurs at runtime if types for which no conversion rules exist (see Section 5.2.2) are assigned. Internal tables constitute the exception to this rule. In this case, table accesses are only permitted to formal parameters that have a corresponding typing.

The example provided in Listing 6.2 shows various typings and their effects on how formal parameters are used in the methods.

Listing 6.2 Typing of Formal Parameters

REPORT z_parameter_typing.

CLASS demo DEFINITION.
  PUBLIC SECTION.
    METHODS: meth1 IMPORTING ipar TYPE any,
             meth2 IMPORTING ipar TYPE any table,
             meth3 IMPORTING ipar TYPE index table.
  ENDCLASS.

5 With dynamic access to a component during an operation on an internal table, for example.
CLASS demo IMPLEMENTATION.
  METHOD meth1.
    DATA num TYPE string.
    num = ipar.
    "READ TABLE ipar INDEX 1"
      TRANSPORTING NO FIELDS.
    "READ TABLE ipar WITH KEY table_line = '...'
      TRANSPORTING NO FIELDS.
  ENDMETHOD.
  METHOD meth2.
    DATA num TYPE string.
    'num = ipar.
    "READ TABLE ipar INDEX 1"
      TRANSPORTING NO FIELDS.
    READ TABLE ipar WITH KEY table_line = '...'
      TRANSPORTING NO FIELDS.
  ENDMETHOD.
  METHOD meth3.
    DATA num TYPE string.
    'num = ipar.
    READ TABLE ipar WITH KEY table_line = '...'
      TRANSPORTING NO FIELDS.
    READ TABLE ipar INDEX 1
      TRANSPORTING NO FIELDS.
  ENDMETHOD.
ENDCLASS.

Three conceivable uses of the input parameter are specified in the methods, while the statements that result in syntax errors for the respective typing are commented out:

- The ipar input parameter of the meth1 method is typed as completely generic. It can be assigned to the num local variables; however, no read operations can be executed for internal tables. When the method is called, any data objects can be passed to the formal parameter. But, if an internal table is passed, an exception occurs during the assignment to num.

- The ipar input parameter of the meth2 method is typed with an internal table that is generic in terms of table type, line type, and table key. It cannot be assigned to the num local variable. Only key access can be executed for internal tables because only these accesses are permitted for all table types. When the method is called, any internal tables can be passed to the formal parameter.
The ipar input parameter of the meth3 method is typed with an index table that is generic in terms of line type and table key. It cannot be assigned to the num local variable. However, all accesses can be executed for internal tables because key and index accesses are possible for index tables. When the method is called, only index tables (and no hash tables) can be passed to the formal parameter.

Formal parameters should be as appropriately typed as possible. The typing must comply with both the implementation requirements and the expectations of the calling program. If you want or need to use a generic type, you should always be as specific as possible. Use generic types like csequence, numeric, simple, and xsequence instead of any. For example, csequence is usually an appropriate typing for text processing. The typings standard table, sorted table, index table, or hashed table are similarly preferable to any table.

The more generic the typing you use, the more careful you must be when using the formal parameter in the implementation to avoid exceptions. Accordingly, you should avoid assigning formal parameters with a typing that is completely generic if you do not want to first check the type at runtime (see Section 11.2) or handle possible exceptions (see Section 8.2).

Unless generic typing is required, you should always use complete typing. Only formal parameters with complete typing always behave in the same way and can be tested locally. You must be particularly careful to ensure that you don’t use generic typing by mistake when you actually intend to use complete typing. This frequently occurs with internal tables with a generic key.

Supply Type

For every formal parameter that awaits a value—input parameters and input/output parameters—by standard, an actual parameter must be specified when the method is called. The assignment of actual parameters to output parameters and return values is always optional.

For input parameters and input/output parameters, this rule can be avoided by declaring the parameter as optional. The syntax is shown below, using the example of an input/output parameter:
No actual parameters have to be specified when the method is called for a formal parameter that is declared as optional. An optional formal parameter for which no actual parameter is specified is initialized in accordance with its type. With the addition DEFAULT, the value and type of an appropriately specified replacement parameter dobj are copied.

In the Class Builder, you can make a formal parameter optional by selecting the Optional column, or by entering a value in the Default value column.

We recommend that you make all formal parameters optional, with the exception of those for which a different entry is actually required each time the method is called. Otherwise, you force your callers to specify unnecessary actual parameters, for which type-specific auxiliary variables often have to be created.

Ensure that the predefined initialization of optional parameters is sufficient or, if you must initialize such a parameter explicitly, for example, in dependence of other parameters. With the special predicate

\[ \ldots \text{IS SUPPLIED} \ldots \]

you can even use a logical expression to react differently in the method, depending on whether an actual parameter is assigned to an optional parameter.

### 6.1.2 Method Calls

This section discusses the options for calling methods statically. A dynamic method call is also possible (see Section 11.4). When a method is called, actual parameters must be passed to all non-optional formal parameters (in other words, all input parameters and input/output parameters that are not defined as optional). Actual parameters can be connected to optional formal parameters. The actual parameters must match the typing of the formal parameters.
The following sections describe static method with increasing complexity of the method interface.

**Static Method Calls**

The simplest method has no interface parameters. Accordingly, the method call is also simple. The statement is as follows:

**No parameters**  

\[ \text{meth( )}. \]

With \text{meth}, you specify the method as it can be addressed as a component of a class or an object in the current location, that is, directly with its name \text{meth} in a method of the same class, or with \text{oref}\rightarrow\text{meth} or \text{class}\rightarrow\text{meth} everywhere the method is visible.

If the method has one non-optional input parameter, the statement is as follows:

**One input parameter**  

\[ \text{meth( dobj ).} \]

The \text{dobj} data object is passed to the input parameter as an actual parameter. If the method has several non-optional input parameters, the statement is as follows:

**Several input parameters**  

\[ \text{meth( \( i1 = \text{dobj1} \) \( i2 = \text{dobj2} \) ... ).} \]

A data object is explicitly assigned to each input parameter. If actual parameters are to be assigned to any formal parameters, the syntax is as follows:

**Any parameter**  

\[ \text{meth( EXPORTING \( i1 = \text{dobj1} \) \( i2 = \text{dobj2} \) ... IMPORTING \( a1 = \text{dobj1} \) \( a2 = \text{dobj2} \) ... CHANGING \( c1 = \text{dobj1} \) \( c2 = \text{dobj2} \) ... ).} \]

With \text{EXPORTING}, you supply the input parameters defined with \text{IMPORTING}. With \text{IMPORTING}, you receive values from output parameters defined with \text{EXPORTING}. With \text{CHANGING}, you assign the actual parameters to the input/output parameters defined with \text{CHANGING}. The equal sign is not an assignment operator in this case. Instead, its function is to bind actual parameters to formal parameters. This syntax includes the previous short forms and can be used instead.

Finally, you can add a \text{CALL METHOD} to all of the previous syntax forms, for example:

**CALL METHOD**  

\[ \text{CALL METHOD meth( \( i1 = \text{dobj1} \) \( i2 = \text{dobj2} \) ... ).} \]
However, this specification is merely unnecessary syntactical noise and can be omitted (as of Release 6.10).\(^6\)

**Functional Method Call**

You may notice that we haven’t mentioned the `RETURNING` parameter of a functional method\(^7\) in our discussion of method calls. This is because functional methods are intended to be used in operand positions. Nevertheless, there is also a separate statement for calling a functional method:

\[
\text{meth( EXPORTING } i1 = \text{dobj1 } i2 = \text{dobj2 } \\
\text{ RECEIVING } r = \text{dobj }).
\]

Here, `RECEIVING` receives the return value in `dobj`; however, this statement is seldom if ever used in practice. The functional equivalent for the above call is as follows:

\[
\text{dobj = meth( } i1 = \text{dobj1 } i2 = \text{dobj2 } 
\]

The call of the functional method can be specified in an operand position, which, in this case, is the source field of an assignment, without specifying `RECEIVING`. When the statement is executed, the method is called and the return value is used as an operand. In the example shown above, it is assigned to `dobj`. The actual parameters are assigned to input parameters using the three syntax forms described above for no input parameters, one input parameter, or several input parameters.

\[
\ldots \text{meth( ) } \ldots \\
\ldots \text{meth( dobj ) } \ldots \\
\ldots \text{meth( } i1 = \text{dobj1 } i2 = \text{dobj2 } \ldots \) \ldots
\]

Functional methods can be used in the same places as built-in functions (see Section 5.2.4). A functional method called with `meth( a )` hides a built-in function with the same name:

- As the source field of an assignment
- As an operand in an arithmetic expression
- As an operand in a logical expression

\(^6\) The `CALL METHOD` language element is only required for the dynamic method calls still.

\(^7\) Remember that a function method can have any number of input parameters and only one return value that is passed by value.
Advanced Concepts in ABAP Objects

- As an operand in the CASE statement
- As an operand in the WHEN statement
- As an operand in the WHERE condition for internal tables

If a functional method called in an operand position sends a class-based exception, this can be handled within a TRY control structure. As of the next release of SAP NetWeaver, you will be able to use functional methods as well as built-in functions and complete arithmetic expressions in almost all operand positions where it is useful to do so. You will be able to use them, in particular, as actual parameters for input parameters of methods, which will allow you to nest method calls.

In Listing 6.3, we have implemented two functional methods get_area and get_volume, to calculate the circular area and volume of a cylinder in a class called cylinder.

Listing 6.3 Functional Methods

```
REPORT z_functional_method.
SELECTION-SCREEN BEGIN OF SCREEN 100.
PARAMETERS: p_radius TYPE i,
            p_height TYPE i.
SELECTION-SCREEN END OF SCREEN 100.
CLASS demo DEFINITION.
  PUBLIC SECTION.
  CLASS-METHODS main.
ENDCLASS.
CLASS cylinder DEFINITION.
  PUBLIC SECTION.
    METHODS: constructor IMPORTING i_radius TYPE numeric
               i_height TYPE numeric,
              get_area RETURNING value(r_area) TYPE f.
              get_volume RETURNING value(r_volume) TYPE f.
  PRIVATE SECTION.
    CONSTANTS pi TYPE f VALUE '3.14159265'.
    DATA: radius TYPE f.
          height TYPE f.
ENDCLASS.
CLASS cylinder IMPLEMENTATION.
  METHOD constructor.
```

8 Classical exceptions cannot be handled in this case.
class demo implementation.

method main.
  data: oref type ref to cylinder,
        volume type string.
  call selection-screen 100 starting at 10 10.
  if sy-subrc = 0.
    create object oref exporting i_radius = p_radius
                  i_height = p_height.
    volume = oref->get_volume().
    concatenate 'volume type ' volume into volume.
    message volume type 'i'.
  endif.
endmethod.
endclass.

start-of-selection.
  demo=>main( ).

the main method of the demo class uses a function call to get_volume on the right side of an assignment, and assigns the result to the volume string. the get_volume method calls get_area in an arithmetic expression. the calculation type of this expression is f.

6.2 Inheritance

in object orientation, inheritance refers to the specialization of classes by deriving subclasses from superclasses.

6.2.1 Basic Principles

classes provide a construction plan for objects. suppose you create two classes called "car" and "truck". you want to implement methods for both classes, which control the objects or return information
about their location and speed. Even at this stage, you can foresee that some parts of the classes will have to be written twice. The inheritance mechanism of an object-oriented programming language provides options that help you to reuse the same or similar parts of a class, and to create a hierarchy of classes.

Superclasses and subclasses
If we examine the two classes (i.e., “Car” and “Truck”) in more detail, it becomes clear that both classes comprise types of vehicles. If you want to create a third class called “Dump truck” it will comprise a specific type of truck. To create a hierarchy relationship between these classes, classes can be derived from each other using inheritance. In our example, “Car” and “Truck” are derived from the “Vehicle” class, while “Dump truck” are derived from the “Truck” class. Derived or more specific classes are referred to as subclasses, while more general classes are called superclasses.

Simple inheritance
The concept of simple inheritance is implemented in ABAP Objects. According to this concept, each class can have several subclasses but only one superclass. In simple inheritance, inheritance relationships are represented by an inheritance tree. Every class in an object-oriented programming language in which simple inheritance is implemented has a unique position as a node in an inheritance tree. This also applies to all the classes we have dealt with up to now, although we have not yet spoken of them in terms of inheritance. For each class, a unique path can be traced back through their superclasses in the inheritance tree until you reach exactly one root node. This root node is the superclass of all classes in the inheritance tree.

Root class
Figure 6.1 illustrates this relationship. The root node of the inheritance tree in ABAP Objects is the predefined, empty, and abstract class `object`.

Derivation
Inheritance simply means that a subclass inherits all components (attributes, methods, events, etc.) of its superclass and can use them like its own components. In each subclass, new elements can be added or methods can be redefined in order to specialize, without this having any impact on the superclass. Elements can only be added in subclasses. It would go against the inheritance concept to remove elements in a subclass.

9 Other programming languages, such as C++, allow a class to be derived from several classes. This mechanism, which is referred to as multiple inheritance, is not implemented in ABAP Objects.
In accordance with this concept, the direct subclasses of the empty root class do not inherit any components from its superclass. Instead, they can add new components. This situation applies to all our sample classes up to now. All classes in ABAP Objects that do not explicitly inherit from another class are implicit direct subclasses of object.

When subclasses of explicitly defined classes are created, these inherit the components of their superclasses and can add new components. Classes become increasingly specialized the further away you move from the root in the inheritance tree. As you move towards the root node, on the other hand, the classes become more generalized.

If you look at a class that is located near the bottom of the inheritance tree, you will notice that the inherited components of the class originate in all classes along the path between this class and the root class, which is the superclass of all classes. In other words, the definition of a subclass is composed of the definitions of all of its superclasses right up to object. The relationship between a subclass and its superclasses should always be expressed as “is a”; for example, “a cargo plane is a plane is a means of transportation is an object.” If this is fulfilled, subclasses can always be handled the same way as superclasses (see polymorphism in Section 6.4).
6.2.2 Creating Subclasses

A superclass has no knowledge of any subclasses it may have. Only a subclass is aware that it is the heir of another class. Therefore, an inheritance relationship can only be defined when a subclass is declared. The syntax for deriving a subclass (subclass) from a superclass (superclass) is as follows:

\[
\text{INHERITING FROM} \quad \text{CLASS} \quad \text{subclass} \quad \text{DEFINITION INHERITING FROM} \quad \text{superclass}.
\]

It therefore involves a simple addition to the \text{CLASS DEFINITION} statement. Any non-final class that is visible at this point can be specified as superclass. To create a subclass in the Class Builder, select \text{Superclass} on the \text{Properties} tab page. Then enter any non-final, global class as a superclass in the \text{Inherits from} field. The \text{Undo inheritance} and \text{Change inheritance} options allow you to change the inheritance relationship (see Figure 6.2).

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{inheritance.png}
\caption{Inheritance in the Class Builder}
\end{figure}

To display the components in a subclass that were inherited from the superclass, select the menu option \text{Utilities \cdot Settings}, and select the \text{Display Inherited Components Also} option.

For each class that does not have an explicit \text{INHERITING FROM} addition, the system implicitly adds the \text{INHERITING FROM object} addition, which means that any class without an \text{INHERITING} addition is automatically a direct subclass of the \text{object} root class.

Listing 6.4 shows the implementation of our example based on vehicles. In this implementation, two classes (\text{car} and \text{truck}) are derived from the \text{vehicle} class.

\begin{flushleft}
\textbf{Listing 6.4} Simple Example of Inheritance
\end{flushleft}

\text{REPORT} \ z\_inheritance. \\
\text{CLASS} \ demo \text{ DEFINITION.} \\
\text{PUBLIC SECTION.}
CLASS-METHODS main.
ENDCLASS.

CLASS vehicle DEFINITION.
PUBLIC SECTION.
  METHODS: accelerate IMPORTING delta TYPE i.
            show_speed.
  ENDCLASS.

CLASS car DEFINITION INHERITING FROM vehicle.
ENDCLASS.

CLASS truck DEFINITION INHERITING FROM vehicle.
PUBLIC SECTION.
  METHODS: load IMPORTING freight TYPE string.
            unload.
  ENDCLASS.

CLASS vehicle IMPLEMENTATION.
METHOD accelerate.
  me->speed = me->speed + delta.
ENDMETHOD.

METHOD show_speed.
  DATA output TYPE string.
  output = me->speed.
  MESSAGE output TYPE 'I'.
ENDMETHOD.
ENDCLASS.

CLASS truck IMPLEMENTATION.
METHOD load.
  me->freight = freight.
ENDMETHOD.

METHOD unload.
  CLEAR me->freight.
ENDMETHOD.
ENDCLASS.

CLASS demo IMPLEMENTATION.
METHOD main.
  DATA: car_ref TYPE REF TO car,
        truck_ref TYPE REF TO truck.
  CREATE OBJECT: car_ref,
                 truck_ref.
  car_ref->accelerate( 130 ).
  car_ref->show_speed( ).
The vehicle class contains a protected attribute (speed) and two public methods (accelerate and show_speed). Note that we have explicitly specified that vehicle inherits from object. Normally, we do not specify the INHERITING addition for such classes. The car and truck classes are both derived from vehicle. Therefore, they inherit the attribute and methods of the vehicle class. Since speed is declared in the PROTECTED SECTION, it is also visible in the subclasses. The truck class is specialized with an additional attribute for freight and additional methods for loading and unloading (load and unload). In this example, the car class receives no additional components. This means that its objects are the same as those of the vehicle class. Since no methods have been added, car does not require an implementation part.

In the main method of the demo class, we use the reference variables car_ref und truck_ref to generate one object each for the two subclasses and call their methods. The accelerate and show_speed methods can be used in both subclasses; however, the load and unload methods can be used only in truck.

### 6.2.3 Visibility Sections and Namespaces in Inheritance

There are three different visibility sections in a class, in which the components of the class are declared (see Section 4.3.2). A subclass inherits all components of its superclasses without changing their visibility. For that reason, only the public and protected components of its superclasses are visible in a subclass. In contrast, the private components are contained in the subclass but are invisible.\(^{10}\) The visibility sections of a subclass therefore contain the following components:

\(^{10}\) Note, however, that the methods inherited from the superclass use the private attributes of the superclass, unless these inherited methods are redefined in the subclass.
PUBLIC
The public visibility section of a subclass contains all public components of all superclasses, plus its own additional public components. These components can be accessed externally using component selectors.

PROTECTED
The protected visibility section of a subclass contains all protected components of all superclasses, plus its own additional protected components. These components cannot be accessed externally using component selectors. From an external point of view, “protected” is the same as “private.”

PRIVATE
The private visibility section of a subclass contains only the subclass’s own private components. These components can only be accessed in the method implementations of the subclass.

Since all visible components in a class must have unique names, all public and protected components of all classes along an inheritance path in the inheritance tree belong to the same namespace and have unique names. Private components, which are only visible within a class and cannot be used in subclasses, must only have unique names within their own class.

The implications of this are as follows: A superclass is not aware of any subclasses it may have. If you create a non-final class in a class library and release it for use, you can never know, as a developer, which subclasses your class will eventually have other than those you define yourself. If you then subsequently add new components to the public or protected section of your class, and any of its subclasses happen to have a component of its own with the same name, this becomes syntactically incorrect. Therefore, it is only secure to add private components. In global classes, not only the external interface but also the interface with any possible subclasses must remain stable.

Therefore, to limit the subclasses of a class to at least the same package, non-final classes should preferably be organized in packages for which the Package Check as Server property is activated (see Section 2.3.3).
6.2.4 Method Redefinition

A subclass inherits all public and protected methods additionally to its own components. When a method is called in the subclass, it is executed in the same way it was implemented in the superclass, and even uses the private components of the superclass. However, since the main purpose of inheritance is to specialize classes, the behavior of the method of a superclass may be too general for the more specific purpose of the subclass. In some cases, the implementation of superclass must be enhanced in the subclass, while in other instances, the implementation must be completely changed. However, the semantics of the method must remain stable for the external user, because all this user ever sees is the constant interface (including the documentation) and not the implementation itself.

Instance methods can be redefined in subclasses to specialize the behavior of subclass objects. Static methods cannot be redefined. Redefining a method means creating a new implementation of the method in a subclass without changing the interface. The method is still declared in the superclass. Previous implementations of the method in preceding superclasses remain unchanged. When a method is redefined in a subclass, an additional implementation is created, which hides the previous implementation when the subclass and further subclasses are used.

Every reference that refers to an object of the subclass uses the redefined method. This is always the case, regardless of the type of the reference variables (for more details, see Section 6.4). This applies in particular to the self reference me. Therefore, if a superclass method (meth1) contains the call of a method (meth2) belonging to the same class, which is redefined in a subclass, the call of the meth1 method in an instance of the superclass results in the execution of the original method (meth2), while the call of the meth1 method in an instance of the subclass results in the execution of the redefined method (meth2).

11 The private methods are also inherited in principle, but are not visible in the subclass.

12 Some other object-oriented programming languages permit the overloading of functions or methods. This means that a separate, changed parameter interface can be defined for an overwritten or redefined method. ABAP Objects does not currently support this mechanism.
Like the methods belonging to the subclass, a redefined method accesses the private attributes of the subclass.

The syntax for redefining an instance method in a subclass is as follows:

METHODS meth REDEFINITION.

This statement must be specified in the declaration part of the subclass in the same visibility section as the actual declaration of the method in the superclass. The definition of the interface is not repeated.

In the Class Builder, you redefine an inherited method by displaying it on the Methods tab. To do so, you must use the Settings function of the Class Builder to select the Display Inherited Components Also entry. Then, you must highlight the method and select the Redefine function (see Figure 6.3).

A new implementation must be created for each redefined method in the redefining subclass. In global classes, the Class Builder does this as part of the Redefine process, and you can navigate to the implementation in the same way as with normal methods. In local classes, you must enter the implementation yourself in the implementation part as for normal methods.

In the implementation of a redefined method, you can use the pseudo reference super-> to access the original method of the direct superclass. This overrides the hiding of the redefined method. You must always use this pseudo reference if you want to first copy the functionality of the superclass and then enhance it.

We can now apply method redefinition to our example from Listing 6.4. Listing 6.5 shows how this differs from Listing 6.4.
Listing 6.5  Method Redefinition

REPORT z_method_redefinition.

...  
CLASS car DEFINITION INHERITING FROM vehicle.
   PUBLIC SECTION.
      METHODS show_speed REDEFINITION.
   ENDCLASS.

CLASS truck DEFINITION INHERITING FROM vehicle.
   PUBLIC SECTION.
      METHODS: accelerate REDEFINITION,
               show_speed REDEFINITION,
               load IMPORTING freight TYPE string,
               unload.
   PROTECTED SECTION.
      DATA freight TYPE string.
   PRIVATE SECTION.
      CONSTANTS max_speed TYPE i VALUE '80'.
   ENDCLASS.

...  
CLASS car IMPLEMENTATION.
   METHOD show_speed.
      DATA output TYPE string.
      output = me->speed.
      CONCATENATE 'Car, speed: ' output INTO output.
      MESSAGE output TYPE 'I'.
   ENDMETHOD.

ENDCLASS.

CLASS truck IMPLEMENTATION.
   METHOD accelerate.
      super->accelerate( delta ).
      IF me->speed > truck=>max_speed.
         me->speed = truck=>max_speed.
      ENDIF.
   ENDMETHOD.

METHOD show_speed.
   DATA output TYPE string.
   output = me->speed.
   CONCATENATE 'Truck with ' me->freight ' speed: ' output INTO output.

368
Inheritance

MESSAGE output TYPE 'I'.
ENDMETHOD.

CLASS demo IMPLEMENTATION.
METHOD main.
  DATA: car_ref TYPE REF TO car,
        truck_ref TYPE REF TO truck.
  CREATE OBJECT: car_ref,
                 truck_ref.
  car_ref->accelerate( 130 ).
  car_ref->show_speed( ).
  truck_ref->load( 'Beer' ).
  truck_ref->accelerate( 110 ).
  truck_ref->show_speed( ).
  truck_ref->unload( ).
ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
demo=>main( ).

We specialize the accelerate method in the truck class and the show_speed method in both subclasses:

▶ In the truck class, we introduced a maximum speed max_speed that cannot be exceeded in accelerate. In the new implementation, the speed is therefore set by calling super->accelerate via the previous implementation, and then checked and adapted, if necessary.

▶ The show_speed method is extended by specific outputs in both subclasses. The previous implementation is not used for this purpose.

All redefined methods keep their original semantics in spite of the new implementation. You will notice that this requires some programming discipline because we can also implement the methods in a completely different way (for more information, see Section 6.4.3).

A test tool that might help you check the stability of applications is ABAP Unit (see Section 13.3).
6.2.5 Abstract Classes and Methods

If you want to use a class just as a template for subclasses and don’t need any objects of this class, you can define the class as an abstract class. The syntax for defining an abstract class is:

```
ABSTRACT CLASS class DEFINITION ABSTRACT.
  ...
ENDCLASS.
```

To create an abstract class in the Class Builder, select Abstract in the Instantiation input field on the Properties tab (see Figure 6.4).

![Figure 6.4 Abstract Global Class](image)

Objects cannot be created from an abstract class using CREATE OBJECT. Instead, abstract classes are used as a template for subclasses. From an abstract class, actual subclasses can be derived from which objects can then be created.

Single instance methods can be identified as abstract as well. The syntax is:

```
METHODS meth ABSTRACT.
```

In the Class Builder, you can identify a method as Abstract in its Detail view (see Figure 6.5).

![Figure 6.5 Abstract Method](image)

Implementation

An abstract method cannot be implemented in its own class, but only in a concrete subclass. Therefore, abstract methods can only be created in abstract classes. Otherwise, it would be possible to create an object with an addressable method but without its implementation. To implement an abstract method in a subclass, you use the method definition mechanism discussed in Section 6.2.4. The only difference to a real redefinition is that you cannot use the super-> pseudo reference in the method.
In an abstract class, both concrete and abstract methods can be declared. Concrete methods are declared and implemented as usual. With the exception of instance constructors, concrete methods can even call abstract methods, because names and interfaces are completely known. The behavior of the abstract method, however, is defined during the implementation in a subclass and can therefore vary in different subclasses.

In our example in Listing 6.5, the vehicle superclass is rather rudimentary and is not used for creating any objects. To prevent this also syntactically, the class can be defined as abstract, as shown in Listing 6.6. Listing 6.6 only demonstrates the differences in comparison with Listing 6.5.

Listing 6.6 Abstract Class and Method

REPORT z_abstract_class.

...  

CLASS vehicle DEFINITION ABSTRACT.

PUBLIC SECTION.

METHODS: accelerate IMPORTING delta TYPE i,

show_speed ABSTRACT.

PROTECTED SECTION.

Figure 6.5 Abstract Method
The `vehicle` class only determines the common elements of the subclasses. Because the two subclasses in Listing 6.5 redefine the `show_speed` method anyway, we declared it in Listing 6.6 as abstract as well. It is therefore no longer implemented in the `vehicle` class.

**Design**

The use of abstract classes and methods can be an important means of object-oriented design. Abstract classes provide a common interface and a partially implemented functionality to their subclasses, but cannot perform any relevant operations on their attributes themselves. In a payroll system, for example, you can imagine a class that already implements many tasks like bank transfers, but only includes the actual payroll function in an abstract manner. It is then the task of various subclasses to perform the correct payroll calculation for different work contracts.

**Interfaces**

Because ABAP Objects does not support multiple inheritance, the usage of abstraction via abstract classes is always restricted to the subclasses of a specific node of the inheritance tree. Interfaces are another means of solving similar tasks, irrespective of the position in the inheritance hierarchy. They are discussed in Section 6.3.

**6.2.6 Final Classes and Methods**

Just as abstract classes and methods require a definition of subclasses in order to work with the classes, there can be adverse situations where you want to protect a whole class or a single method from uncontrolled specialization. For this purpose, you can declare a class or an instance method as final. This can make sense particularly if you want to make changes to a class at a later stage without causing any subclasses to become syntactically or semantically incorrect (see the namespace of components in inheritance in Section 6.2.3). If you follow the defensive procedure for programming the AS ABAP using
ABAP Objects, which was introduced in Section 4.9, the declaration of final classes is always recommended.

The syntax for defining a final class is:

```plaintext
CLASS class DEFINITION FINAL.

ENDCLASS.
```

In the Class Builder, you create a final class by selecting the Final checkbox on the Properties tab (see Figure 6.4). You cannot derive any more subclasses from a final class. A final class therefore terminates a path of the inheritance hierarchy. All instance methods of a final class are automatically final.

In a non-final class, individual instance methods can be declared as final. The syntax is:

```plaintext
METHODS meth FINAL.
```

In the Class Builder, you can identify an instance method as Final in its Detail view (see Figure 6.5). A final method cannot be redefined in subclasses. A final method cannot be abstract at the same time. A class can be final and abstract at the same time, but only its static components are usable in this case. Although you can declare instance components in such a class, it is not recommended.

### 6.2.7 Static Attributes in Inheritance

To use a static component of a class, instances of the class are not required. If instances exist, they share the static components. How does inheritance affect static components, and static attributes in particular?

Like all components, a static attribute exists exactly once within a path of the inheritance tree. A subclass can access the contents of the public and protected static attributes of all superclasses. Alternatively, a superclass shares its public and protected static attributes with all subclasses. In inheritance, a static attribute is therefore not assigned to a single class, but to a path of the inheritance tree. It can be accessed from outside via the class component selector (=>) using all class names involved, or from inside in all affected classes where a static attribute is visible. Changes to the value are visible in all relevant classes. Listing 6.7 shows a simple example.
Listing 6.7 Static Attributes in Inheritance

REPORT z_static_attributes.

CLASS demo DEFINITION.
  PUBLIC SECTION.
  CLASS-METHODS main.
ENDCLASS.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
  CLASS-DATA a1 TYPE string.
ENDCLASS.

CLASS c2 DEFINITION INHERITING FROM c1.
.
ENDCLASS.

CLASS demo IMPLEMENTATION.
  METHOD main.
    c2=>a1 = 'ABAP Objects'.
    MESSAGE c1=>a1 TYPE 'I'.
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  demo=>main( ).

Static constructor

When addressing a static attribute belonging to a path of an inheritance tree, you always address the class in which the attribute is declared, irrespective of the class name used in the class component selector. This is important for calling the static constructor (see Section 6.2.8). A static constructor is executed when a class is addressed for the first time. If a static attribute is addressed via the class name of a subclass but declared in a superclass, only the static constructor of the superclass is executed.

Static methods

Static methods cannot be redefined in ABAP Objects, because static components should occur exactly once (i.e., not more or less) in a path so that they can be shared by all subclasses.

6.2.8 Constructors in Inheritance

Constructors are used for initializing the attributes of a class (see Section 4.7). While instance constructors can set the instance attributes of every single object during the instancing process, the static constructors are responsible for the static attributes of the class before the class is first accessed. Because a subclass inherits all attributes of

374
its superclasses in inheritance, this automatically begs the question “How can the constructors ensure that the inherited attributes are initialized as well when the subclass is used?”

**Instance Constructors**

Every class has a predefined instance constructor named `constructor`. Instance constructors thus deviate from the rule that there are only unique component names along a path of the inheritance tree. Consequently, the instance constructors of the individual classes of an inheritance tree must be completely independent of one another. To avoid naming conflicts, the following rules apply:

- Instance constructors of superclasses cannot be redefined in subclasses.
- Instance constructors cannot be explicitly called via the `constructor( )` statement.

After an object has been created with the `CREATE OBJECT` command, the instance constructor is automatically invoked. Because a subclass contains all superclass attributes that are visible to it, the contents of which can be set by instance constructors of these classes, the instance constructor of a subclass must ensure that the instance constructors of all superclasses are executed as well. For this purpose, the instance constructor of every subclass must contain a call `super->constructor( ... )`.

of the instance constructor of the direct superclass, even if the constructor is not explicitly declared. The only exceptions to this rule are the direct subclasses of the root node, `object`. The `super->constructor( ... )` statement is the only exception from the rule that constructors cannot be explicitly called.

In superclasses in which the instance constructor is not explicitly declared and implemented, the implicitly existing implementation of the instance constructor is run. It automatically ensures that the instance constructor of the next higher superclass is called.

Before an instance constructor is run, you must supply its non-optional input parameters. These are searched for as follows:

13 In particular, the private attributes of superclasses can only be initialized in the superclasses' own constructors.
Advanced Concepts in ABAP Objects

- **Provision in CREATE OBJECT**
  Starting with the class of the created object, the first explicitly defined instance constructor is searched for in the corresponding path of the inheritance tree. This is the instance constructor of the class itself, or the first explicitly defined instance constructor of a superclass.

- **Provision in super->constructor( ... )**
  Starting with the direct superclass, the first explicitly defined instance constructor is searched for in the corresponding path of the inheritance tree.

In CREATE OBJECT or in super->constructor( ... ), respectively, the interface of the first explicitly defined instance constructor is provided with values like a normal method:

- If there are no input parameters, no parameters are transferred.
- Optional input parameters can be provided with values.
- Non-optional input parameters must be provided with values.

If there is no explicitly defined instance constructor in the path of the inheritance tree up to the object root class, no parameters will be transferred.

Inheritance tree

For both CREATE OBJECT and super->constructor( ... ), the first explicit instance constructor must therefore be regarded and, if one exists, its interface must be provided with a value. When working with subclasses, you therefore need to know the entire path very well because when creating a subclass object that resides at the lower end of the inheritance tree, a situation can occur whereby parameters must be transferred to the constructor of a superclass positioned much closer to the root node.

The instance constructor of a subclass is split into two parts by the super->constructor( ... ) call required by the syntax. In the statements before the call, the constructor behaves like a static method. Before the call, it does not have access to the instance attributes of its class, that is, instance attributes cannot be addressed until after the call.

3-phase model

The execution of a subclass instance constructor can therefore be divided into three phases that are presented in the comment lines of Listing 6.8.
LISTING 6.8 Three-Phase Model of an Instance Constructor

METHOD constructor.
  " Phase 1: Access to static attributes only
  ..."  
  " Phase 2: Execution of super class constructor(s)
  CALL METHOD super->constructor EXPORTING ...
  " Phase 3: Access to instance attributes only
  ..."
ENDMETHOD.

In the individual phases, the instance constructor can execute the following tasks:

- **Phase 1**
  Here you can prepare the call of the superclass instance constructor, for example, you can determine the actual parameters for its interface.

- **Phase 2**
  In this phase, the instance constructor of the superclass is executed, which is again divided into three phases, if implemented.

- **Phase 3**
  The attributes of all superclasses are now correctly initialized. Using these values, the necessary initializations for the own instance attributes can be performed.

Therefore, during the instantiation of a subclass, a nested call of the instance constructors from the subclass to the superclasses takes place, where the instance attributes of the highest superclass can be addressed only as of the deepest nesting level. When returning to the constructors of the subclasses underneath, their instance attributes can also be addressed successively.

The methods of subclasses are not visible in constructors. If an instance constructor calls an instance method of the same class via the implicit self-reference me, the method is called in the way in which it is implemented in the class of the instance constructor, and not the possibly redefined method of the subclass to be instantiated. This is an exception to the rule that whenever instance methods are called, the implementation is called in the class of the instance to which the reference is pointing.

Listing 6.9 shows the behavior of instance constructors in inheritance using a simple example.
Listing 6.9 Instance Constructors in Inheritance

REPORT z_constructor_inheritance.

CLASS demo DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS main.
  ENDCLASS.

CLASS vessel DEFINITION.
  PUBLIC SECTION.
    METHODS constructor IMPORTING i_name TYPE string.
  PROTECTED SECTION.
    DATA name TYPE string.
  ENDCLASS.

CLASS ship DEFINITION INHERITING FROM vessel.
  ...
  ENDCLASS.

CLASS motorship DEFINITION INHERITING FROM ship.
  PUBLIC SECTION.
    METHODS constructor IMPORTING i_name TYPE string
                    i_fuelamount TYPE i.
  PRIVATE SECTION.
    DATA fuelamount TYPE i.
  ENDCLASS.

CLASS vessel IMPLEMENTATION.
  METHOD constructor.
    name = i_name.
  ENDMETHOD.

CLASS motorship IMPLEMENTATION.
  METHOD constructor.
    super->constructor(i_name).
    fuelamount = i_fuelamount.
  ENDMETHOD.

CLASS demo IMPLEMENTATION.
  METHOD main.
    DATA: o_vessel TYPE REF TO vessel,
         o_ship TYPE REF TO ship,
         o_motorship TYPE REF TO motorship.
    CREATE OBJECT:
      o_vessel EXPORTING i_name = 'Vincent',
      o_ship EXPORTING i_name = 'Mia',
      o_motorship EXPORTING i_name = 'Jules'
                     i_fuelamount = 12000.

This example shows three consecutive classes of the inheritance hierarchy. The vessel class has an instance constructor with an input parameter. From vessel, we derive the ship class that does not explicitly declare and implement the instance constructor. From ship, we derive motorship. This class again has an explicit instance constructor with two input parameters. We create an object from every class and provide the parameter interface of the constructors with actual parameters. The constructors are called as follows:

- The object created using o_vessel is initialized at CREATE OBJECT in the explicit instance constructor of vessel, where an attribute is set using the passed actual parameter.
- The object created using o_ship is also initialized at CREATE OBJECT via the instance constructor of vessel, because it is called by the implicit instance constructor of ship. Its parameter interface needs to be provided with actual parameters.
- The object created using o_motorship is initialized in the explicit instance constructor of motorship. In this constructor, the instance constructor of the direct superclass must be called via super->constructor. The implicit instance constructor of ship calls the explicit instance constructor of vessel. Its parameter interface needs to be provided with actual parameters.

You can best understand the behavior of the program if you run it line by line in the ABAP Debugger.

**Static Constructors**

Every class has a static constructor named class_constructor. With regard to the namespace along an inheritance tree, the same rules that apply to the instance constructor also apply to the static constructor.

When a subclass is addressed for the first time in a program, its static constructor is run. Before that, however, the preceding static constructors of the entire inheritance tree must have been run. Because
a static constructor should be called only once during the execution of a program, when a subclass is addressed for the first time, the next higher superclass is searched whose static constructor has not yet run. Then this static constructor is executed first, followed by the constructors of all subclasses up to and including the addressed subclass. In contrast to instance constructors, a static constructor does not have to explicitly call the static constructor of its superclass. Instead, the runtime environment automatically ensures that the static constructors are called in the correct order. In a subclass, you can always assume that the static attributes of the superclasses have been correctly initialized.

6.2.9 Instantiation in Inheritance

A subclass includes the object descriptions of all superclasses. The instantiation of a subclass therefore means the instantiation of all superclasses in a single object, where the initialization of the superclass attributes is ensured by calling the superclass constructors, as described in Section 4.3.2.

The additions CREATE PUBLIC|PROTECTED|PRIVATE of the CLASS statement or the corresponding Class Builder settings, respectively, control for each class who can create an instance of the class or call its instance constructor (see Section 4.3.2). In inheritance, this results in three scenarios whose behavior is defined in ABAP Objects as follows:

- **Superclass with Public Instantiation**
  The instance constructor of the superclass is publicly visible. If the instantiatiability of a subclass is not explicitly specified, it inherits the public instantiation of the superclass. The instantiatiability of a subclass can be explicitly specified in one of the three ways. A subclass can control the visibility of its own instance constructor independently of the superclass.

- **Superclass with Protected Instantiation**
  The instance constructor of the superclass is visible in subclasses. If the instantiatiability of a subclass is not explicitly specified it inherits the protected instantiation of the superclass. The instantiatiability of a subclass can be explicitly specified in one of the three ways. A subclass can control the visibility of its own instance constructor independently of the superclass and can thus also
Superclass with Private Instantiation
The instance constructor of the superclass is visible only in the superclass. There are two different scenarios here:

- The subclass is not a friend of the superclass.
  Because only the superclass itself can call its instance constructor, the subclass cannot be instantiated. Therefore, the subclass has an implicit addition, `CREATE NONE`. The instantiatability of the subclass cannot be explicitly specified because this would mean a publication of the superclass constructor in the specified section.

- The subclass is a friend of the superclass.
  If the instantiatability of the subclass has not been explicitly specified, it inherits the private instantiation of the superclass. The instantiatability of a subclass can be explicitly specified in one of the three ways. As a friend, a subclass can publish the private constructor of the superclass in the specified section.

If a superclass with private instantiation has been defined in a path of the inheritance tree, no subclass can be instantiated by external users, and a subclass cannot even instantiate itself because it does not have access to the instance constructor of the superclass! The obvious thing to do would be to make a class defined for private instantiation a final class in order to prevent subclasses from being derived.

Exceptions from this rule only exist if a privately instantiatable superclass offers its friendship to its subclasses. This is not often the case, though, because a superclass usually does not know its subclasses. However, a superclass can offer its friendship to an interface as well, which can then be implemented by its subclasses (see Section 6.3.3). As always, when offering friendship, you should proceed very carefully in this case as well, for example, by restricting the usage of the friendly interface to the current package.

6.3 Standalone Interfaces

In ABAP Objects, interfaces of classes can be defined independently from a class as standalone interfaces.
6.3.1 Basic Principles

Point of contact

The only part of a class that is relevant to an external user is its public interface that is made up of the components of its public visibility section. All other components are irrelevant to the user. This aspect becomes clear particularly when using abstract methods in abstract classes (see Section 6.2.5). Basically, such classes are used to define nothing but interfaces that can only be used with objects of subclasses.

No multiple inheritance

Because ABAP Objects does not support multiple inheritance, the usage of abstract classes for defining interfaces is restricted to their subclasses. However, it is also desirable to be able to define generally valid interfaces that can equally be used in several classes.

Decoupling

Such generally valid interfaces can be provided via standalone interfaces. Standalone interfaces are independently definable interfaces without implementation that can be integrated and implemented in classes. Standalone interfaces are used to achieve a looser coupling between a class and a user, because they provide an additional access layer (protocol). Two scenarios are possible:

▶ A class entirely or partially provides its public interface to the user via one or several standalone interfaces and thus decouples the user from the actual class definition. Every standalone interface describes an independent aspect of the class and only provides this aspect and nothing else to a user. This can positively affect the maintainability of a class.

▶ A user has an exact idea of how an object should be used and defines an standalone interface containing all wanted components. Every class that is to fulfill this task integrates this interface and provides the functionality.

BAdi

A very nice application example of this decoupling is given by the enhancebility of delivered ABAP application programs in customer systems using Business Add-Ins (BAdis). BAdis are based on standalone interfaces that are declared in the original system. The actual functionality of a BAdi is provided only in follow-up systems by implementing the standalone interface in classes.14

---

14 The comprehensive topic of enhancing and modifying ABAP applications of AS ABAP will not yet be discussed in this edition.
Because standalone interfaces are just interfaces without implementation, you cannot create any objects from them—similar to abstract classes. Instead, they are integrated and implemented in classes. If a class implements a standalone interface, it can be addressed via this interface. There are specific interface reference variables for this purpose. These can point to objects of all classes that contain the respective standalone interface. Because any classes can integrate the same interface, their objects can be addressed via the same interface reference variable.

Figure 6.6 illustrates the role of interfaces in a graphical way. In our representation of objects with a core that is separated from the external user by a shell, standalone interfaces can be imagined as empty shells that can be used by classes instead of their own shells or as parts of their own shells.\(^{15}\) For example, if a class wants to provide services like outputting its attributes in a list or serialization, it can implement the corresponding standalone interfaces. Users who are only interested in these different aspects of objects access these via interface reference variables. In the following sections, we will discuss the language elements shown in Figure 6.6 in detail.

\(^{15}\) Compared to Figure 6.1, you can clearly see that the integration of standalone interfaces in classes can also be regarded as a multiple inheritance of interfaces to classes. Because standalone interfaces don’t have their own method implementations, there are no conceptual problems like those that occur in multiple inheritance of classes.
6.3.2 Creating Interfaces

With regard to their declaration, interfaces in ABAP Objects play the same role as classes. Just like classes, interfaces are object types that reside in the namespace of all types. While a class describes all aspects of a class, an interface only describes a partial aspect. As mentioned above, standalone interfaces can be regarded as special abstract classes without implementation that can be used in multiple classes.

Accordingly, the declaration of a standalone interface hardly varies from the declaration of a class. As with classes, we distinguish global and local interfaces in the same way that we do global and local classes. Therefore, the same rules apply regarding their usability. Global interfaces can be used in any program if the package assignment of the program permits it. Local interfaces can only be used in the same program.

The syntax for declaring a local interface is:

```
INTERFACE intf.
  DATA .
  CLASS-DATA .
  METHODS .
  CLASS-METHODS .
  .
ENDINTERFACE.
```

Basically, the declaration of an interface corresponds to the declaration part of a class, where instead of `CLASS—ENDCLASS`, you simply use `INTERFACE—ENDINTERFACE`. Interfaces can contain exactly the same components as classes. Unlike classes, however, interfaces don’t need to be divided into different visibility sections because interface components are always integrated in the public visibility section of classes.

To create a global interface, use the Class Builder just as you would for global classes. In the Object Navigator, select `Create • Class Library • Interface`. In Transaction SE24, after selecting `Create`, select the `Interface` object type instead of `Class`.\(^{16}\)

\(^{16}\) If you observe the naming convention IF..., bzw. ZIF..., an interface is created automatically.
Figure 6.7 shows the Class Builder for a global interface ZIF_DRIVE_OBJECT. You see the familiar user interface that you know from working with classes. When creating components, you need to specify the same input as you do for classes, except for the assignment to a visibility section. In the shown example, we created the same methods ACCELERATE and SHOW_SPEED as in ZCL_VEHICLE presented in Figure 4.7 in Chapter 4. The shown interface can therefore serve as an interface to objects that can be driven.

The Class Builder generates the corresponding ABAP statements in a program of the interface pool type, the source code of which can also be edited directly via Goto • Interface Section (see Figure 6.8). As in class pools, the addition PUBLIC identifies the interface as a global interface that can be used in all programs. Apart from the declaration of the global interface, an interface pool cannot contain any local type declarations except for the publication of type groups.17

The essential difference between interfaces and classes is that there is no implementation part for an interface. Therefore, it is not necessary to add DEFINITION to INTERFACE. The methods of an interface are all abstract. They are fully declared, including their parameter interface, but not implemented in the interface. Like the subclasses that

17 In interface pools, declarations like these would not be of any use. They are possible in class pools, but can only be used in the private section of the global class. This section does not exist for interfaces.
implement the abstract methods of their abstract superclasses, all classes that want to use an interface must implement its methods.\textsuperscript{18}

![Source Code of an Interface Pool](image)

**Figure 6.8** Source Code of an Interface Pool

### 6.3.3 Implementing Interfaces in Classes

Every class can implement one or more interfaces. The essential requirement for implementing an interface is that the interface is known to the implementing class. Therefore, it must be declared globally in the class library or locally in the same program. Additionally, the usage of the interface must be permitted by the package assignment.

**INTERFACES**

The syntax for implementing interfaces is:

```plaintext
CLASS class DEFINITION.
  PUBLIC SECTION.
    INTERFACES: intf1, intf2 ...
    ...
    ...
  ENDCLASS.
```

Interfaces are therefore integrated using the `INTERFACES` statement in the public visibility section of a class. Only global interfaces can be integrated in the public visibility section of a global class. You can do this on the **Interfaces** tab of Class Builder.

\textsuperscript{18} Strictly speaking, however, this similarity applies only to instance methods. In interfaces, you can also define static methods without implementation. This is not possible in abstract classes because static methods cannot be redefined.
In Figure 6.9, we copied the ZCL_VEHICLE class shown in Figure 4.7 to a new class ZCL_VEHICLE_WITH_INTF, deleted its method, and specified the interface ZIF_DRIVE_OBJECT shown in Figure 6.7. In the Abstract and Final columns, you can specify that all methods of the interface should be either abstract or final in the class. In the \texttt{INTERFACES} statement, this is expressed by the optional addition \texttt{ALL METHODS ABSTRACT|FINAL}.

Implementing an interface extends the public interface of the class by the interface components. Every \texttt{comp} component of an implemented \texttt{intf} interface becomes a full component of the class and is identified within the class via the name

\ldots \texttt{intf-comp} \ldots

Interface components are inherited to subclasses like class-specific public components. A class can have its own component of the same name like an interface component, or various implemented interfaces can contain components of the same name. All reside in one namespace and are distinguished in the class by different \texttt{intf-} prefixes. The tilde sign (~) is the interface component selector.

Figure 6.10 shows how the methods of the interface ZIF_DRIVE_OBJECT are presented in ZCL_VEHICLE_WITH_INTF. In the detailed view (see Figure 6.5), you can specify for every single method if it is to be abstract or final. The \texttt{INTERFACES} statement has the optional additions for this purpose, \texttt{ABSTRACT METHODS} and \texttt{FINAL METHODS}.
Otherwise, however, an interface method can no longer be changed in a class. The same applies to interface attributes. The only property that can be changed when integrating it in a class is the initial value (addition `DATA VALUES to INTERFACES`).

A class must implement all concrete (non-abstract) methods of all integrated interfaces in its implementation part. In the Class Builder, this is achieved via the usual procedure, by selecting `Code` for every interface method. In the ZCL_VEHICLE_WITH_INTF class, we basically used the method implementations of ZCL_VEHICLE (see Listing 6.10).

### Listing 6.10 Implementation of Interface Methods

```abap
CLASS zcl_vehicle_with_intf IMPLEMENTATION.
  METHOD zif_drive_object~accelerate.
    speed = speed + delta.
  ENDMETHOD.
  METHOD zif_drive_object~show_speed.
    DATA output TYPE string.
    output = speed.
    CONCATENATE 'Vehicle speed: ' output INTO output.
    MESSAGE output TYPE 'I'.
  ENDMETHOD.
ENDCLASS.
```

If a class does not declare its own components in its public visibility section, but only integrates standalone interfaces, the entire public
interface of the class is defined via standalone interfaces; and stand-
dalone interfaces and its public interface are indeed the same for this
class. This applies to our sample class ZCL_VEHICLE_WITH_INTF.
The interface to the outside world that had so far been built of the
class’s own components is now completely outsourced to the ZIF_
DRIVE_OBJECT interface.

Listing 6.11 summarizes what we have just described using the
example of a local interface. The public interface of the vehicle class
from Listing 4.5 is outsourced to a local standalone interface; how-
ever, the local vehicle class could just as easily implement the global
interface ZIF_DRIVE_OBJECT instead of a local interface drive_
object.

Listing 6.11 Declaration and Implementation of a Local Interface

REPORT z_vehicle_with_intf.

INTERFACE drive_object.
  METHODS: accelerate IMPORTING delta TYPE i,
    show_speed.
ENDINTERFACE.

CLASS vehicle DEFINITION.
  PUBLIC SECTION.
    INTERFACES drive_object.
  PRIVATE SECTION.
    DATA speed TYPE i.
ENDCLASS.

CLASS vehicle IMPLEMENTATION.
  METHOD drive_object~accelerate.
    speed = speed + delta.
ENDMETHOD
  METHOD drive_object~show_speed.
    DATA output TYPE string.
    CONCATENATE 'Vehicle speed: ' output INTO output.
    output = speed.
    MESSAGE output TYPE 'I'.
ENDMETHOD.
ENDCLASS.

6.3.4 Access to Interfaces of Objects

Objects are always accessed via object reference variables. Until
now, we worked with object reference variables that were declared
with a reference to a class:
Class reference variable

```plaintext
DATA cref TYPE REF TO class.
```

By using these reference variables, you can address all those components of an object's class that are visible at the current position. This kind of object reference variable is therefore referred to as a class reference variable.

As you saw in the previous section, the interface components of an interface implemented in a class are handled as full components. You might therefore be tempted to address the interface components of an object as follows:

```plaintext
... cref->intf~comp ...
```

In point of fact, this works. You can try this with our ZCL_VEHICLE_WITH_INTF class; however, this kind of access is not recommended. The external user of a class should be able to access its components without having to worry about the technical composition of the interface. Standalone interfaces and the class-specific components both define different sets of components. They should be used directly, but not in mixed forms as shown above. In short, the interface component selector should only be used within classes (and interfaces, see Section 6.3.6).

To access the interface components of objects, ABAP Objects includes interface reference variables. These are object reference variables that are declared with a reference to an interface:

```plaintext
DATA ref TYPE REF TO intf.
```

An interface reference variable can point to the objects of all classes implementing the intf interface. Using such a reference variable, all components of the interface of an object can be addressed directly via

```plaintext
... iref->comp ...
```

In contrast to `cref->intf~comp`, the interface reference variable `iref->comp` expresses that components of a class are accessed that are hierarchically on the same level but reside in a different part of the interface. An interface reference variable enables you to address those components of an object that were added to the object's class via the implementation of the intf interface that was used to declare the class. Other components—class-specific components or components of other interfaces—cannot be addressed via an interface reference variable (not even dynamically, see Sections 11.1.1 and 11.4.1).
Figure 6.11 shows how class and interface reference variables point to the same object, where the interface reference variable only knows its own interface components, and the class reference variable should only be used to address the non-interface components of the class.

The code in Figure 6.11 already shows how interface reference variables can point to objects. You can simply assign a class reference variable pointing to an object to an interface reference variable. Usually, this is an up cast (see Section 6.4.2 for more information).

This can be accomplished even more comfortably if you’re only interested in the interface components of a class. For example, you are naturally only interested in the interface components of a class if the entire public interface of a class is defined via a standalone interface. In these situations, creating the objects of the class via an interface reference variable will suffice:

```
CREATE OBJECT iref TYPE class EXPORTING ...
```

Via the TYPE addition, you specify the class of the object to be created and provide the instance constructor with EXPORTING, if necessary. However, you don’t need a class reference variable to create the object. The only prerequisite is that the class class (or one of its superclasses) contain the intf interface.

A user of object reference variables usually works with objects without having to deal with the details of their implementation. In con-
trast to the work with class reference variables, a user of an interface reference variable normally doesn’t even need to know from which class the object it is working with originates.

The example shown in Listing 6.12 demonstrates the usage of interface reference variables. The methods main and output of the demo class exclusively work with such object reference variables that were all created with a reference to our sample interface ZIF_DRIVE_OBJECT. For this purpose, an internal table type is declared in demo the line type of which is such a reference type. In addition to our global sample class CL_VEHICLE_WITH_INTF, we have also created a local class electron that contains the standalone interface as well, but specifically implements the methods by storing the speed in units of the speed of light (c=300.000).

From each of the two classes, an object is created and accelerated, and the object reference is appended to an internal table. Then this table is transferred to the output method where the show_speed interface method is executed line by line.

Listing 6.12 Standalone Interface Reference Variables

REPORT z_drive_many_objects.
CLASS demo DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS main.
  PRIVATE SECTION.
    TYPES iref_tab_type TYPE TABLE OF
       REF TO zif_drive_object.
    CLASS-METHODS output IMPORTING iref_tab
      TYPE iref_tab_type.
ENDCLASS.
CLASS electron DEFINITION.
  PUBLIC SECTION.
    INTERFACES zif_drive_object.
  PRIVATE SECTION.
    CONSTANTS c TYPE i VALUE 300000.
    DATA speed_over_c TYPE p DECIMALS 3.
ENDCLASS.
CLASS electron IMPLEMENTATION.
  METHOD zif_drive_object~accelerate.
    me->speed_over_c = me->speed_over_c + delta / c.
ENDMETHOD.
  METHOD zif_drive_object~show_speed.
DATA output TYPE string.
output = me->speed_over_c.
CONCATENATE 'Electron speed/c: ' output INTO output.
MESSAGE output TYPE '1'.
ENDMETHOD.
ENDCLASS.

CLASS demo IMPLEMENTATION.

METHOD main.
DATA: iref_tab TYPE iref_tab_type,
     iref LIKE LINE OF iref_tab.
CREATE OBJECT iref TYPE zcl_vehicle_with_intf.
iref->accelerate(100).
APPEND iref TO iref_tab.
CREATE OBJECT iref TYPE electron.
iref->accelerate(250000).
APPEND iref TO iref_tab.
demo=>output(iref_tab).
ENDMETHOD.

METHOD output.
DATA iref LIKE LINE OF iref_tab.
LOOP AT iref_tab INTO iref.
  iref->show_speed( ).
ENDLOOP.
ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
demo=>main( ).

Although the example is similar to the one shown in Listing 4.8, it has a completely new quality. As before, the internal table is a collection of pointers to objects. Because these pointers are interface reference objects, however, the classes and thus the behavior of the objects managed by an internal table can vary.

You should pay special attention to the output method. This method is an example of the user mentioned above who works with objects without knowing their classes. The output method receives a table with reference variables and knows that it can call a show_speed method there. The actual implementation is irrelevant to it. This matches the concept of polymorphism that is illustrated in Figure 6.14 exactly and will be further discussed in the corresponding section. For the moment, it will suffice just to note that syntactically identical method calls in a loop lead to different output.
6.3.5 Access to Static Interface Components

Because interfaces can contain the same components as classes, static components are possible as well. You cannot access the static components of an interface using the name of the interface and the class component selector. The only exceptions are constants declared via \texttt{CONSTANTS}:

\[ \ldots \text{ intf=>const} \ldots \]

The static components belong to the static components of every implementing class. This means that static attributes have different values depending on the class and that static methods can be differently implemented in every class. To access the static components of interfaces, independently of the instance, you would have to use the name of an implementing class and the interface component selector:

\[ \ldots \text{ class=>intf-comp} \ldots \]

However, this should be the exception for the reasons mentioned in Section 6.3.4. Instead, implementing classes should declare aliases (see Section 6.3.7) for the static components of interfaces and therefore make them addressable via the class name like their own static components. Naturally, you can always use interface reference variables for accessing static components after you created objects from the implementing classes.

6.3.6 Composing Interfaces

In Figure 6.9, it is apparent that the Class Builder provides the \texttt{Interfaces} tab as well for an interface as it does for a class. Accordingly, the \texttt{INTERFACES} statement cannot only be used in classes but also in the declaration of an interface:

\begin{verbatim}
INTERFACE intf1.
  INTERFACES: intf2, intf3, ...
\end{verbatim}

This mechanism allows you to compose several interfaces into one interface. The composition of interfaces can be useful when modeling complex applications.
The set of components of an interface $\text{intf}_1$ that integrates additional interfaces (i.e., $\text{intf}_2$, $\text{intf}_3$, ...) are composed of its own components and the components of the integrated interfaces. The components all reside on the same level. An interface containing at least one other interface is called composite or nested interface. An interface integrated in another interface is called a component interface. A component interface can be composed itself. Let’s now look at the nesting of interfaces shown in Listing 6.13.

Listing 6.13 Composite Interfaces

```plaintext
INTERFACE $\text{intf}_1$.
...
ENDINTERFACE.

INTERFACE $\text{intf}_2$.
INTERFACES: $\text{intf}_1$ ...
...
ENDINTERFACE.

INTERFACE $\text{intf}_3$.
INTERFACES: $\text{intf}_1$, $\text{intf}_2$ ...
...
ENDINTERFACE.
```

The composite interface $\text{intf}_3$ has a component $\text{intf}_2$ that is composed itself. Although it seems like the nesting of several interfaces caused a component hierarchy, this is not the case. All component interfaces of a composite interface are on the same level. A nesting of names like $\text{intf}_3$-$\text{intf}_2$-$\text{intf}_1$ is not possible.

In the example above, the component interface $\text{intf}_1$ of the composite interface $\text{intf}_2$ becomes a component interface of $\text{intf}_3$. A composite interface contains each component interface exactly once. Although $\text{intf}_1$ is integrated in $\text{intf}_3$ both directly as a component interface of $\text{intf}_3$ and indirectly via $\text{intf}_2$, it only occurs once. In $\text{intf}_3$, it can only be addressed under the name $\text{intf}_1$, even if it was not integrated directly.

If a composite interface is implemented in a class, all interface components of the interface behave as if their interface had been implemented only once. The interface components of the individual component interfaces extend the public interface of the class by its original name. Because every interface is included exactly once in a composite interface, naming conflicts cannot occur. The way an
implemented interface is composed is irrelevant when it is implemented in a class. Next, let’s look at the example shown in Listing 6.14:

Listing 6.14  Implementation of Composite Interfaces

INTERFACE intf1.
  METHODS meth.
ENDINTERFACE.

INTERFACE intf2.
  INTERFACES intf1.
  METHODS meth.
ENDINTERFACE.

INTERFACE intf3.
  INTERFACES intf1.
  METHODS meth.
ENDINTERFACE.

INTERFACE intf4.
  INTERFACES: intf2, intf3.
ENDINTERFACE.

CLASS class DEFINITION.
  PUBLIC SECTION.
    INTERFACES intf4.
ENDCLASS.

CLASS class IMPLEMENTATION.
  METHOD intf1-meth. ... ENDMETHOD.
  METHOD intf2-meth. ... ENDMETHOD.
  METHOD intf3-meth. ... ENDMETHOD.
ENDCLASS.

A method *meth* of the same name is declared in three individual interfaces and thus implemented in three different ways using the interface component selector. The composition of the interfaces does not play any role. The *intf1-meth* method is implemented only once, although it occurs in two interfaces, *intf2* and *intf3*. The name *intf4* does not show up in the implementation part of the class at all.

If you list one or more of the other interfaces—*intf1*, *intf2*, or *intf3*—in addition to *intf4* in the declaration part of the class mentioned above, the components and the implementation part of the class do not change at all, because the compiler always ensures for a class as well as in composite interfaces that every component exists only once.
If the class of an object implements a composite interface, the object is accessed in the same way as if the class implemented every interface individually. This means that interface components should be accessed using interface reference variables of the type of the appropriate component interface. This can always be achieved using the corresponding assignments to interface reference variables (up casts, see Section 6.4.2). The interface component selector should not be used for this purpose; however, it can be used in a composite interface to make the components of component interfaces as accessible as native components via aliasing.

6.3.7 Alias Names for Interface Components

The complete name of a component that is added via an interface to a class or another interface is \textit{intf-comp}. For this name, you can define an alias name at the level at which the interface is integrated using the \texttt{INTERFACES} statement:

\begin{verbatim}
ALIASES name FOR intf-comp.
\end{verbatim}

Alias names can be assigned when interfaces are implemented in the declaration part of a class or when interfaces are composed in the declaration of an interface. In the Class Builder, you can enter alias names for classes and for interfaces in the \textit{Aliases} tab (see Figure 6.12).

**Figure 6.12** Alias Names

**Alias Names in Classes**

In classes, alias names belong to the namespace of the components of a class and must be assigned to a visibility section just like the other

397
components. The visibility of an alias name from outside the class depends on its visibility section and not on the visibility section of the assigned interface component.

In Listing 6.15, we modify the example of Listing 6.12 by using alias names. For this purpose, in the local class electron, we declare an alias name accelerate for the zif_drive_object-accelerate interface method as we did for ZCL_VEHICLE_WITH_INTF (see Figure 6.12). Listing 6.15 shows only the differences between it and Listing 6.12.

Listing 6.15  Alias Names in Classes

REPORT z_drive_viaAliases.
...
CLASS electron DEFINITION.
  PUBLIC SECTION.
    INTERFACES zif_drive_object.
    ALIASES accelerate FOR zif_drive_object-accelerate.
  PRIVATE SECTION.
  ...
ENDCLASS.
CLASS electron IMPLEMENTATION.
  METHOD accelerate.
    me->speed_over_c = me->speed_over_c + delta / c.
  ENDMETHOD.
  ...
ENDCLASS.
CLASS demo IMPLEMENTATION.
  METHOD main.
    DATA: vehicle TYPE REF TO zcl_vehicle_with_intf,
          electron TYPE REF TO electron,
          iref_tab TYPE iref_tab_type.
    CREATE OBJECT vehicle.
      vehicle->accelerate( 100 ).
    APPEND vehicle TO iref_tab.
    CREATE OBJECT electron.
      electron->accelerate( 250000 ).
    APPEND electron TO iref_tab.
    demo=>output( iref_tab ).
  ENDMETHOD.
  ...
ENDCLASS.
START-OF-SELECTION.
  demo=>main( ).
The interface method can now be implemented in the class via its alias name and called by a user like a direct method of the class. Here, we change the main method in which the classes for the object creation must be known, anyway so that the objects are created via class reference variables. Because of the alias name, the class reference variables can be used to call the interface method \texttt{accelerate} without using the interface component selector. Nothing is changed in the \texttt{output} method, which does not need to know the classes.

Using alias names, a class can publish its interface components as class-specific components, so to speak. In particular, alias names can be used in classes to further on address class-specific components that are outsourced to standalone interfaces in the course of a development cycle using their old name. Then, the users of the class don't need to be adapted to the new names.

In our sample class \texttt{ZCL\_VEHICLE\_WITH\_INTF}, we converted the methods from \texttt{ZCL\_VEHICLE} to interface methods. Just imagine if we had made this change directly in \texttt{ZCL\_VEHICLE}! All users would have become syntactically incorrect. By introducing alias names simultaneously, however, the class would have remained addressable.

\textbf{Alias Names in Composite Interfaces}

Because names cannot be concatenated in composite interfaces, alias names provide the only means of addressing those components that would otherwise not be available in the composite interface. Let's look at the example shown in Listing 6.16.

\begin{verbatim}
Listing 6.16  Alias Names in Interfaces
INTERFACE intf1.
   METHODS meth1.
ENDINTERFACE.

INTERFACE intf2.
   INTERFACES intf1.
   ALIASES meth1 FOR intf1-meth1.
ENDINTERFACE.

INTERFACE intf3.
   INTERFACES intf2.
   ALIASES meth1 FOR intf2-meth1.
ENDINTERFACE.
\end{verbatim}
The intf3 interface can use the alias name meth1 in intf2 to address the meth1 component of the intf1 interface in its own ALIASES statement. Without alias names in intf2, this would not be possible because the name intf2~intf1~m1 is not permitted. Now the user of intf3 can access the component meth1 in intf1 without having to know anything about the composition of the interface:

```
DATA i_ref TYPE REF TO intf3.
...
i_ref->meth1( ... ).
```

Without alias names in intf3, the access would look as follows:

```
i_ref->intf1~meth1( ... ).
```

The user would have to know that intf3 is composed of intf2, which is composed of intf1. For global interfaces, in particular, the user should not have to look at the composition of an interface in the Class Builder before he can use a method of the interface. Of course, it is not necessary that the alias names always match the original names.

### 6.3.8 Interfaces and Inheritance

To conclude the description of interfaces, we will discuss the relationship of standalone interfaces to inheritance and compare both concepts in a summary.

The concepts of standalone interfaces and inheritance are independent of each other and totally compatible. Any number of interfaces can be implemented in the classes of an inheritance tree, but every interface can be implemented only once per inheritance tree path. Thus, every interface component has a unique name intf~comp throughout the inheritance tree and is contained in all subclasses of the class implementing the interface. After their implementation, interface methods are full components of a class and can be redefined in subclasses. Although interface methods cannot be identified as abstract or final in the interface declaration, every class can specify these settings when implementing the interface.

**Coupling**

The usage of inheritance always makes sense when different classes have a generalization/specialization relationship. For example, if we
regard two classes “cargo plane” and “passenger plane”, both classes contain components that can be declared in a common “plane” superclass. The big advantage of inheritance is that the subclasses take on and reuse all properties already programmed in the superclass. At the same time, this causes a very tight coupling between superclasses and subclasses. A subclass strongly depends on its superclass, because it often largely consists of the superclass components. A subclass must know its superclass exactly. This became particularly clear, for example, in the discussion of instance constructors in inheritance (see Section 6.2.8). Every change to non-private components of a superclass changes all of its subclasses. Conversely, subclasses can also affect the design of superclasses due to specific requests. If you use inheritance for defining classes, you should ideally have access to all classes involved because only all of the classes in a path of the inheritance tree make a reasonable whole. On the other hand, it is dangerous to just link to some superclass by defining a subclass if the superclass does not belong to the same package, or was explicitly shared as a superclass in the package interface.19

The implementation of interfaces is always recommended when interfaces or protocols are to be described without having to use a specific type of implementation. An additional layer is introduced between user and class that decouples the user from an explicit class and therefore makes it much more independent. Interfaces allow the user to handle the most different classes, which don’t need to be related to each other. In object-oriented modeling, interfaces provide an abstraction that is independent of classes. Irrespective of the actual implementation, the services required by a user can be described. Additionally, interfaces also implement an aspect of multiple inheritance, because several interfaces can be implemented in a class. If a programming language permits a real multiple inheritance, this multiple inheritance is usually used in the sense of interfaces as well. This means that only abstract classes with exclusively abstract methods are suitable as different superclasses of a single subclass. Otherwise, the question would arise regarding which method implementation is actually used in a subclass if it is already implemented

19 A complete package concept that allows you to predefine and check such specifications in the package interface will only be implemented in the next SAP NetWeaver release.
6.4 Object References and Polymorphism

Object references are the linchpin when dealing with objects. They are used for creating and addressing objects. As the contents of object reference variables, they can be assigned to other variables or passed to procedures.

Object reference variables are divided into class reference variables and interface reference variables. When using interface reference variables, we already observed that the type of a reference variable does not have to match the type of the referenced object. In this section, we will have a closer look at this fact and at the resulting polymorphic behavior of method calls.

6.4.1 Static and Dynamic Type

In this section, we define two important terms for reference variables, that is, their static type and dynamic type.

**Static type**

The static type of a reference variable `oref` is the type that is specified after

```
... oref TYPE REF TO class|intf ...
```

in the declaration. As with all data objects, the static type is fixed during the entire runtime of a program. For object reference variables, the object types `class` for class reference variables and `intf` for interface reference variables are possible as static types.21

---

20 This is the "diamond" problem of multiple inheritance. A method that is declared in a superclass is redefined in two subclasses, which, in turn, make up the superclass of another subclass. Which implementation is used in this subclass? For interfaces, this problem does not occur, because in the implementation of composite interfaces every interface method exists only once.

21 Accordingly, data types are possible as static types for data reference variables that can point to data objects (see Section 11.1.2).

---
Index

- (structure component selector) 245
#EC (extended program check) 945
$TMP package 61
& (literal operator) 269
(F1) help, dynpro 548
(F4) help, dynpro 549
* (Comment) 85
:: (statement chain) 94
< (relational operator) 293
<= (relational operator) 293
= (assignment operator) 273
= (relational operator) 293
=> (class component selector) 129, 198
-> (object component selector) 129, 204, 250
>(relational operator) 293
->* (dereferencing operator) 250, 811
>= (relational operator) 293
?= (casting operator) 409, 817
~ (column selector) 722, 728
~ (interface component selector) 387
4GL
fourth-generation language 24, 25

A

A

message type 670
ABAP
Advanced Business Application Programming 23
classic 449
Generic Report Generation Processor 23
programming models 450
requirements 30
ABAP Debugger
classical 951
configuring 954
new 951
tool 950
use 959
user interface 952
using 97

ABAP Dictionary
dynpro field 531
storage 69
tool 69, 254
ABAP Editor
configuring 84
direct entry 57
ABAP glossary
ABAP keyword documentation 138
ABAP keyword documentation
opening 94
using 136
ABAP memory
data cluster 785
main mode 174
ABAP Objects
ABAP 24
object orientation 180
use 26, 221
Using 110
ABAP processor
AS ABAP 147
ABAP program
activating 88
call 156
copying 91
creation 82
design 152
execute 155
executing 89
load 155, 175
modularization 183
testing 939
type 159
ABAP runtime analysis
tool 980
ABAP Runtime environment
AS ABAP 150
ABAP runtime environment
AS ABAP 143
virtual machine 88
ABAP scripting
BSP 671
ABAP syntax
comments and statements 84
Index

ATRA
  transaction  981
Attribute
  creation  192
declaring  116
object orientation  178, 342
XML  912
AUTHORITY_CHECK_TCODE
  authorization check  792
AUTHORITY-CHECK
  ABAP statement  792
Authorization
  authorization check  790
checking  791
Authorization check
  AS ABAP  789
Authorization group
  authorization check  792
Authorization object
  authorization check  790
Authorization profile
  authorization check  791
Automation Controller
  CFW  588
Automation queue
  CFW  595
AVG
  aggregate function  715

B
BACK
  ABAP Statement  646
Background job
  scheduling  451
Background process
  tRFC  860
Background processing
  executable program  451, 641
  scheduling  644
BACKGROUND TASK
  CALL FUNCTION  859
Background task
  background request  641
BADI
  interface  382
Base list
  classical list  647
displaying  648
BEGIN OF
  TYPES/DATA  244
BEGIN OF BLOCK
  SELECTION-SCREEN  628
BEGIN OF LINE
  SELECTION-SCREEN  628
BEGIN OF TABBED BLOCK
  SELECTION-SCREEN  628
BETWEEN
  predicate  296
  WHERE clause  722
bgRFC
  Background RFC  862
Bit expression
  calculation expression  286
Block
  selection screen  628
Boolean operator
  logical expression  297
Branch
  conditional  298
  control structure  298
BREAK-POINT
  ABAP statement  957
Breakpoint
  ABAP Debugger  956
  Setting  96
Browser control
  CFW  592
  example  608
BSP
  Business Server Pages  145, 515, 671
Built-in type
  ABAP Dictionary  253
  ABAP program  236
Bundling
  SAP LUW  744
Business key
  object identity  759
Business Server Page
  AS ABAP  145
Button
  screen element  527
  Web Dynpro ABAP  691
BY
  SORT  335
BYPASSING BUFFER
  Open SQL  753
Byte code
  ABAP program 88
Byte field
  data object 237, 241
Byte string
  data object 242
  processing 303
BYTE-CA
  relational operator 296
BYTE-CN
  relational operator 296
BYTE-CO
  relational operator 296
BYTE-CS
  relational operator 296
Byte-like
  data type 237
BYTE-NA
  relational operator 296
BYTE-NS
  relational operator 296

C
  ABAP type 236, 239
CA
  relational operator 296, 316
Calculation expression
  ABAP syntax 286
Calculation type
  calculation expression 288
CALL FUNCTION
  ABAP statement 472
  dynamic 835
Call hierarchy
  exception handling 487
CALL METHOD
  ABAP statement 356
  dynamic 832
CALL SCREEN
  ABAP statement 520, 522
  CALL SELECTION-SCREEN
  ABAP statement 630
Call sequence
  main session 174
CALL SUBSCREEN
  dynpro statement 581
CALL TRANSACTION
  ABAP statement 521
CALL TRANSFORMATION
  ABAP program 929
  ABAP statement 920
  iXML Library 914
CALL TRANSACTION
  ABAP statement 158
Callback routine
  aRFC 855
  context menu 539
CASE
  ABAP statement 299
  control structure 299
Case distinction
  control structure 299
CASTING
  ASSIGN 807
  Casting
    field symbol 806
    casting operator
      down cast 409
CATCH
  ABAP statement 484
CATCH block
  TRY control structure 485
CATCH SYSTEM-EXCEPTIONS
  ABAP statement 500
Catchable runtime error
  use 501
CATT
  tool 986
  ceil
    numeric function 287
CFW
  Control Framework 588
CHAIN
  dynpro statement 544, 548
Change and Transport System
  CTS 62
CHANGING
  Actual parameter 356
  FORM 474
  METHOD 195
  METHODS 346
  PERFORM 475
Character string
  find/replace 306
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation 305</td>
</tr>
<tr>
<td>processing 303</td>
</tr>
<tr>
<td>Character-like data type 237</td>
</tr>
<tr>
<td>charlen description function 287, 315</td>
</tr>
<tr>
<td>CHECK ABAp statement 302</td>
</tr>
<tr>
<td>Check Indicator authorization check 791</td>
</tr>
<tr>
<td>Check table dynpro 546</td>
</tr>
<tr>
<td>foreign key 81</td>
</tr>
<tr>
<td>input help 550</td>
</tr>
<tr>
<td>Check variant Code Inspector 946</td>
</tr>
<tr>
<td>CHECKBOX PARAMETERS 619</td>
</tr>
<tr>
<td>Checkbox screen element 527</td>
</tr>
<tr>
<td>selection screen 619</td>
</tr>
<tr>
<td>Checking typing 351</td>
</tr>
<tr>
<td>Checkpoint ABAP program 957</td>
</tr>
<tr>
<td>Checkpoint group assertion 510</td>
</tr>
<tr>
<td>breakpoint 957</td>
</tr>
<tr>
<td>CL_ABAP_CLASSDESCR RTTI 823</td>
</tr>
<tr>
<td>CL_ABAP_MATCHER regular expression 312</td>
</tr>
<tr>
<td>CL_ABAP_MEMORY_AREA shared objects 441</td>
</tr>
<tr>
<td>CL_ABAP_REGEX regular expression 312</td>
</tr>
<tr>
<td>CL_ABAP_STRUCTDESCR RTTI 823</td>
</tr>
<tr>
<td>CL_ABAP_TYPEDESCR RTTI 820</td>
</tr>
<tr>
<td>CL_AUNIT_ASSERT ABAP Unit 968</td>
</tr>
<tr>
<td>CL_CTMENU context menu 539</td>
</tr>
<tr>
<td>CL_GUI_ALV_GRID CFW 593</td>
</tr>
<tr>
<td>CL_GUI_CFW CFW 589</td>
</tr>
<tr>
<td>CL_GUI_CONTROL CFW 589</td>
</tr>
<tr>
<td>CL_GUI_CUSTOM_CONTAINER CFW 590</td>
</tr>
<tr>
<td>CL_GUI_DIALOGBOX_CONTAINER CFW 590</td>
</tr>
<tr>
<td>CL_GUI_DOCKING_CONTAINER CFW 590</td>
</tr>
<tr>
<td>CL_GUI_FRONTEND_SERVICES system class 781</td>
</tr>
<tr>
<td>CL_GUI_HTML_VIEWER CFW 592</td>
</tr>
<tr>
<td>CL_GUI_OBJECT CFW 589</td>
</tr>
<tr>
<td>CL_GUI_PICTURE CFW 591</td>
</tr>
<tr>
<td>CL_GUI_SIMPLE_TREE CFW 592</td>
</tr>
<tr>
<td>CL_GUI_SPLITTER_CONTAINER CFW 590</td>
</tr>
<tr>
<td>CL_GUI_TEXTEDIT CFW 592</td>
</tr>
<tr>
<td>CL_GUI_TOOLBAR CFW 591</td>
</tr>
<tr>
<td>CL_HTTP.Utility ICF 890</td>
</tr>
<tr>
<td>CL_SALV_EVENTS_TABLE ALV 660</td>
</tr>
<tr>
<td>CL_SALV_FUNCTIONS ALV 660</td>
</tr>
<tr>
<td>CL_SALV_HIERSEQ_TABLE ALV 599</td>
</tr>
<tr>
<td>CL_SALV_PRINT ALV 664</td>
</tr>
<tr>
<td>CL_SALV_TABLE ALV 599</td>
</tr>
<tr>
<td>CL_SALV_TREE ALV 599</td>
</tr>
<tr>
<td>CL_SHM_AREA shared objects 441</td>
</tr>
<tr>
<td>CLASS ABAP statement 184</td>
</tr>
<tr>
<td>Class ABAP Objects 180</td>
</tr>
<tr>
<td>abstract 370</td>
</tr>
<tr>
<td>concrete 370</td>
</tr>
<tr>
<td>creation 112, 183</td>
</tr>
<tr>
<td>final 373</td>
</tr>
<tr>
<td>global 182</td>
</tr>
<tr>
<td>local 182</td>
</tr>
<tr>
<td>object orientation 342</td>
</tr>
</tbody>
</table>
Persistence Service 770
Communication
asynchronous 843
synchronous 843
Communication scenarios
RFC 849
COMMUNICATION_FAILURE
RFC 855
Communications technology
AS ABAP 841
Comparison
byte-like 295
character-like 295
conversion 293
logical expression 293
numeric 294
WHERE clause 722
Compatibility mode
Transaction Service 774
Compatible
data object 273
Compilation unit
ABAP program 152
complete typing
Usage 354
Complex
data type 231
Component
structure 244
Component controller
web dynpro component 682
Component interface
interface 395
Component specification
dynamic 830
COMPUTE
ABAP statement 288
Computing time
analyze 980
CONCATENATE
ABAP statement 305
Concrete
subclass 370
CONDENSE
ABAP statement 305
Consolidation system
CTS 62
Constant
class 194
data object 266
CONSTANTS
ABAP statement 194, 266
Constructor
class 214
creation 114
inheritance 375
constructor
instance constructor 214
Container
class 190
Container control
CFW 590
Context binding
Web Dynpro ABAP 688
Context mapping
Web Dynpro ABAP 686
Context menu
defining 539
SAP GUI 539
CONTINUE
ABAP statement 302
Control event
example 610
Control Framework
CFW 588
Control level change
control structure 332
Control level processing 332
Control structure
processing block 298
Controller
MVC 672, 681
CONTROLS
ABAP statement 579, 583
Conversion
assignment 274
byte-like type 276
date and time 277
numeric type 276
text-type type 275
Conversion error
type conversion 278
Conversion routine
ABAP Dictionary 285
Conversion rule
comparison 294
elementary data type 275
internal table 281
structure 280
  type conversion 275
CONVERT DATE
  ABAP statement 292
CONVERT TEXT
  ABAP statement 305
CONVERT TIME STAMP
  ABAP statement 292
Convertible
data object 274
Cookie
  access 886
Copy-on-write
dynamic data object 974
  internal table 282
CORRESPONDING FIELDS
  INTO clause 718
cos
  floating point function 287
cosh
  floating point function 287
COUNT
  aggregate function 715
Coupling
  inheritance 400
Coverage Analyzer
calling 985
tool 984
CP
  relational operator 296, 316
CREATE DATA
  ABAP statement 271, 812
CREATE OBJECT
  ABAP statement 203, 391
CREATE PRIVATE
  CLASS 189
CREATE PROTECTED
  CLASS 189
CREATE PUBLIC
  CLASS 189
CREATE_PERSISTENT
  Persistence Service 765
  Cross-transaction application buffer
data cluster 785
CS
  relational operator 296, 316
csequence
  generic type 264
CTS
  SAP Change and Transport System 62
cursor
data type 735
  Cursor variable
database cursor 735
Custom container
  CFW 590
Custom controls
  screen element 527
Customer system
  namespace 63
CX_DYNAMIC_CHECK
  exception category 492
  exception class 482
CX_NO_CHECK
  exception category 493
  exception class 482
CX_ROOT
  exception class 482
CX_STATIC_CHECK
  exception category 492
  exception class 482
CX_SY_NO_HANDLER
  exception class 490

D
  ABAP type 236, 240
DATA
  ABAP statement 192, 226
Data
  ABAP program 225
  attribute 227
  character-like 229
  encapsulation 228
  local 226
  numerical 229
  program-global 227
data
  generic type 264
Data Browser
tool 98
DATA BUFFER
  data cluster 785
Data cluster
  AS ABAP 784
deleting 787
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>reading</td>
<td>786</td>
</tr>
<tr>
<td>storing</td>
<td>785</td>
</tr>
<tr>
<td>Data declaration</td>
<td>page 94</td>
</tr>
<tr>
<td>Data element</td>
<td>page 74</td>
</tr>
<tr>
<td>activation</td>
<td>page 74</td>
</tr>
<tr>
<td>creation</td>
<td>page 72, 256</td>
</tr>
<tr>
<td>database table</td>
<td>page 71</td>
</tr>
<tr>
<td>Data element documentation</td>
<td>page 548</td>
</tr>
<tr>
<td>field help</td>
<td>page 548</td>
</tr>
<tr>
<td>Data element maintenance</td>
<td>page 73</td>
</tr>
<tr>
<td>ABAP Dictionary</td>
<td>page 255</td>
</tr>
<tr>
<td>Data elements</td>
<td>page 255</td>
</tr>
<tr>
<td>ABAP Dictionary</td>
<td>page 73</td>
</tr>
<tr>
<td>Data encapsulation</td>
<td>page 27</td>
</tr>
<tr>
<td>Data object</td>
<td>page 27</td>
</tr>
<tr>
<td>ABAP program</td>
<td>page 226</td>
</tr>
<tr>
<td>anonymous</td>
<td>page 271, 812</td>
</tr>
<tr>
<td>assigning</td>
<td>page 273</td>
</tr>
<tr>
<td>context</td>
<td>page 226</td>
</tr>
<tr>
<td>converting</td>
<td>page 274</td>
</tr>
<tr>
<td>declaring</td>
<td>page 226</td>
</tr>
<tr>
<td>dynamic</td>
<td>page 241</td>
</tr>
<tr>
<td>elementary</td>
<td>page 236</td>
</tr>
<tr>
<td>named</td>
<td>page 265</td>
</tr>
<tr>
<td>names</td>
<td>page 226</td>
</tr>
<tr>
<td>numeric</td>
<td>page 287</td>
</tr>
<tr>
<td>operations</td>
<td>page 273</td>
</tr>
<tr>
<td>predefined</td>
<td>page 271</td>
</tr>
<tr>
<td>static</td>
<td>page 237</td>
</tr>
<tr>
<td>Data processing</td>
<td>page 29</td>
</tr>
<tr>
<td>business</td>
<td>page 29</td>
</tr>
<tr>
<td>Data reference</td>
<td>page 817</td>
</tr>
<tr>
<td>assign</td>
<td>page 817</td>
</tr>
<tr>
<td>dynamic programming</td>
<td>page 796</td>
</tr>
<tr>
<td>Data reference variable</td>
<td>page 809</td>
</tr>
<tr>
<td>declare</td>
<td>page 809</td>
</tr>
<tr>
<td>declaring</td>
<td>page 250</td>
</tr>
<tr>
<td>Data root</td>
<td>page 817</td>
</tr>
<tr>
<td>Simple Transformation</td>
<td>page 930</td>
</tr>
<tr>
<td>Data storage</td>
<td>page 741</td>
</tr>
<tr>
<td>consistency</td>
<td>page 741</td>
</tr>
<tr>
<td>persistent</td>
<td>page 705</td>
</tr>
<tr>
<td>Data transmission</td>
<td>page 842</td>
</tr>
<tr>
<td>AS ABAP</td>
<td>page 842</td>
</tr>
<tr>
<td>Data type</td>
<td>page 232</td>
</tr>
<tr>
<td>ABAP Dictionary</td>
<td>page 233, 250</td>
</tr>
<tr>
<td>bound</td>
<td>page 232</td>
</tr>
<tr>
<td>class</td>
<td>page 200</td>
</tr>
<tr>
<td>class/interface</td>
<td>page 233</td>
</tr>
<tr>
<td>data object</td>
<td>page 229</td>
</tr>
<tr>
<td>defining</td>
<td>page 232</td>
</tr>
<tr>
<td>domain</td>
<td>page 75</td>
</tr>
<tr>
<td>elementary</td>
<td>page 236</td>
</tr>
<tr>
<td>generic</td>
<td>page 237</td>
</tr>
<tr>
<td>global</td>
<td>page 233, 251</td>
</tr>
<tr>
<td>independent</td>
<td>page 232</td>
</tr>
<tr>
<td>Local</td>
<td>page 116</td>
</tr>
<tr>
<td>program-local</td>
<td>page 234</td>
</tr>
<tr>
<td>type group</td>
<td>page 233</td>
</tr>
<tr>
<td>type hierarchy</td>
<td>page 229</td>
</tr>
<tr>
<td>use</td>
<td>page 234</td>
</tr>
<tr>
<td>Database</td>
<td>page 734</td>
</tr>
<tr>
<td>AS ABAP</td>
<td>page 148</td>
</tr>
<tr>
<td>commit</td>
<td>page 743</td>
</tr>
<tr>
<td>lock</td>
<td>page 748</td>
</tr>
<tr>
<td>LUW</td>
<td>page 742</td>
</tr>
<tr>
<td>rollback</td>
<td>page 744</td>
</tr>
<tr>
<td>Database cursor</td>
<td>page 734</td>
</tr>
<tr>
<td>opening</td>
<td>page 734</td>
</tr>
<tr>
<td>Database interface</td>
<td>page 734</td>
</tr>
<tr>
<td>AS ABAP</td>
<td>page 148, 706</td>
</tr>
<tr>
<td>Database logon</td>
<td>page 172</td>
</tr>
<tr>
<td>Database table</td>
<td>page 723</td>
</tr>
<tr>
<td>ABAP Dictionary</td>
<td>page 707</td>
</tr>
<tr>
<td>activating</td>
<td>page 77</td>
</tr>
<tr>
<td>change contents</td>
<td>page 738</td>
</tr>
<tr>
<td>change row</td>
<td>page 739</td>
</tr>
<tr>
<td>create</td>
<td>page 68</td>
</tr>
<tr>
<td>data cluster</td>
<td>page 785</td>
</tr>
<tr>
<td>data type</td>
<td>page 252</td>
</tr>
<tr>
<td>delete row</td>
<td>page 741</td>
</tr>
<tr>
<td>insert or add row</td>
<td>page 740</td>
</tr>
<tr>
<td>insert rows</td>
<td>page 738</td>
</tr>
<tr>
<td>reading</td>
<td>page 712</td>
</tr>
<tr>
<td>relational</td>
<td>page 706</td>
</tr>
<tr>
<td>repository object</td>
<td>page 68</td>
</tr>
<tr>
<td>structure</td>
<td>page 708</td>
</tr>
<tr>
<td>technical settings</td>
<td>page 76</td>
</tr>
<tr>
<td>Database view</td>
<td>page 727</td>
</tr>
<tr>
<td>ABAP Dictionary</td>
<td>page 709</td>
</tr>
<tr>
<td>reading</td>
<td>page 727</td>
</tr>
<tr>
<td>Date field</td>
<td>page 727</td>
</tr>
<tr>
<td>calculating</td>
<td>page 290</td>
</tr>
<tr>
<td>comparing</td>
<td>page 294</td>
</tr>
<tr>
<td>data object</td>
<td>page 237, 240</td>
</tr>
</tbody>
</table>
validity 291
DCL
   Data Control Language 707
DDL
   Data Definition Language 707
Debuggee
   ABAP Debugger 951
Debugger
   ABAP Debugger 951
Debugger Breakpoint
   ABAP Debugger 956
Debugger tools
   ABAP Debugger 955
Debugging session
   ABAP Debugger 955
   ending 956
   starting 955
DECIMALS
   TYPES/DATA 243
DECLARATION
   CLASS 185
Declaration part
   ABAP program 87, 152
   class 184
   top include 168
Decoupling
   interface 401
   object orientation 382
Deep
   data type 261
   structure 262
Deep data object
   memory requirements 262, 973
DEFAULT
   METHODS 355
   PARAMETERS 619
DEFERRED
   CLASS DEFINITION 211
DEFINE
   ABAP statement 170
DELETE DATASET
   ABAP statement 778
DELETE dbtab
   ABAP statement 741
DELETE FROM
   data cluster 787
DELETE itab
   ABAP statement 334
DELETE_PERSISTENT
   Persistence Service 769
Delivery class
   database table 70
DEQUEUE
   lock function module 751
Dereferencing
   data reference variable 811
Dereferencing operator
   using 250
DESCENDING
   SORT 335
DESCRIBE FIELD 820
DESCRIBE_BY_DATA
   RTTI 823
DESCRIBE_BY_NAME
   RTTI 823
Description function
   function 287
Deserialization
   ST 927
   XML-ABAP 908
   XSLT 921
Design by contract
   assertion 511
DESTINATION
   CALL FUNCTION 849, 854
   DESTINATION IN GROUP
      CALL FUNCTION 856
Destructor
   ABAP Objects 219
Details list
   classical list 647
   creating 652
Developer
   authorization 54
Developer key
   SAPNet 55
Development class
   package 62
Development environment
   ABAP 26
Development object
   ABAP Workbench 59
Development system
   CTS 62
Dialog box container
   CFW 590
Dialog interface
  AS ABAP  146
Dialog module
  creating  542
  data  227
  processing block  154, 165
  use  542
Dialog programming
  classical ABAP  457
Dialog status
  GUI status  535
Dialog step
  PA/PBO  543
Dialog transaction
  creating  520
  execution  457
  initial dynpro  520
  selection screen  631
  transaction code  157
Dialog transactions
  use  460
Dialog window
  modal  523
Dictionary type
  creation  253
div
  arithmetic operator  286
DIVIDE
  ABAP statement  286
DML
  Data Manipulation Language  707
DO
  ABAP statement  301
Docking container
  CFW  590
  example  604
Documentation
  data element  256
DOM
  Document Object Model  913, 914
DOM object
  iXML library  918
Domain
  ABAP Dictionary  256
  creation  256
  data element  74
Domain management
  ABAP Dictionary  76
Domains
  creation  75
Double-click
  classical list  651
Down Cast
  data reference variable  817
Down cast
  inheritance  410
  interface  411
  object reference variable  409
Downward compatibility
  ABAP  36, 449
  ABAP Objects  181
Dropdown
  list box  553
DTD
  document type definitions  913, 914
Dynamic Access
  class components  801
Dynamic data object
  dynamic memory object  972
Dynamic documents
  CFW  599
Dynamic type
  object reference variable  403
  polymorphism  413
  reference variable  282
  dynamic type
  data reference variable  809
Dynpro
  ABAP Objects  555
  ABAP program  518
  AS ABAP  146
  control  573
  creating  524
  dialog transaction  457
  dynamic program  515
  dynpro sequence  519
  example  557
  exiting  523
  field  530
  function group  556
  input check  545
  number  524
  process  458
  properties  525
  type  525
  usage  146
Dynpro data transport
   automatic 543
   controlling 544
Dynpro field
   data transport 531
   data type 251, 531
Dynpro flow logic
   implementing 529
   statements 530
   table control 578
Dynpro number
   selection screen 617
Dynpro processing
   message 668
Dynpro screens
   data transport 519
Dynpro sequence
   dialog transaction 524
   dynpro process 458
   nesting 523
   terminating 522
Dynpros
   layout 526

E
   message type 669
eCATT
      Extended Computer Aided Test Tool 986
Editor mode
   Class Builder 199
Element
   XML 910
Elementary
   data type 231
Elementary ABAP type
   asXML 922
   JCO 875
   RFC API 866
Elementary data object
   assigning 275
   comparing 294
   initial value 285
Elementary type
   declaring 242
ELSE
   ABAP statement 298
ELSEIF
   ABAP statement 298
Encapsulation
   class 187
   classic ABAP 461
   object orientation 179
   procedure 188
   use 223
END OF
   TYPES/DATA 244
ENDING AT
   CALL SCREEN 523
   CALL SELECTION-SCREEN 630
END-OF-PAGE
   List event 646
   END-OF-SELECTION event 453
Enhancement category
   database table 76
ENQUEUE
   lock function module 749
Enterprise service
   client proxy 905
Enterprise Services Repository
   Exchange Infrastructure 893
Enterprise SOA
   XML 908
EQ
   relational operator 293
Error
   exception situation 480
Error message
   dynpro 559
   message type 669
   error_message classical exception 670
Errors
   avoiding 479
Event
   ABAP Objects 27, 422
   ABAP runtime environment 452
   class 423
   declaring 424
   inheritance 428
   management 431
   Object orientation 343
   triggering 426
   Web Dynpro ABAP 691
Index

Event block
data 227
  function group 465
  processing block 154, 165
Event handler
declaring 428
  method 423
EVENTS
  ABAP statement 424
Example library
  ABAP keyword documentation 139
Exception
catch 484
category 492
declaring 114, 490
  function module 469
  handle 484
  handler 484
  non-class-based 500
Parameter interface 345
  propagating 487
  raising 483
  RFC API 867
  untreatable 481
Exception category
  use 493
Exception class
  advantages 482
  attributes 494
  class-based exception 481
  creation 111, 493
  exception text 495
  local 494
  methods 494
Exception group
  catchable runtime error 501
  exception class 507
Exception handler
class-based exception 485
Exception handling
  ABAP 481
  class-based 481
  class-based/classic 505
  classic 500
  cleanup tasks 489
  executing 96
  function module 472
  messages 503
Exception object
class-based exception 481
  generating 483
Exception text
  creation 496
  message 496, 505, 670
  OTR 496
  use 496
EXCEPTIONS
  CALL FUNCTION 502
  CALL METHOD 502
  METHODS 502
EXCEPTION-TABLE
  CALL FUNCTION 835
  CALL METHOD 833
EXCLUDING
  SET PF-STATUS 538
EXEC SQL
  ABAP statement 754
Executable program
  ABAP runtime enviroment 452
  call 156
  execute 451
  program type 160
  use 460
EXISTS
  WHERE clause 731
EXIT
  ABAP statement 302
Exit message
  message type 670
  exp
    floating point function 287
EXPORT
  ABAP statement 785
EXPORTING
  Actual parameter 356
  EVENTS 424
  METHOD 195
  METHODS 346
  RAISE EVENT 426
Extension information system
  Object Navigator 59

F

f
  ABAP type 236, 238
  calculation type 289
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Factory method</strong></td>
</tr>
<tr>
<td>class</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td><strong>logical expression</strong></td>
</tr>
<tr>
<td>Favorites</td>
</tr>
<tr>
<td>SAP Easy Access</td>
</tr>
<tr>
<td>Favorites menu</td>
</tr>
<tr>
<td>SAP Easy Access</td>
</tr>
<tr>
<td><strong>FETCH NEXT CURSOR</strong></td>
</tr>
<tr>
<td>ABAP statement</td>
</tr>
<tr>
<td>FIELD</td>
</tr>
<tr>
<td>Dynpro statement</td>
</tr>
<tr>
<td>Field</td>
</tr>
<tr>
<td>database table</td>
</tr>
<tr>
<td>Field help</td>
</tr>
<tr>
<td>defining</td>
</tr>
<tr>
<td>dynpro</td>
</tr>
<tr>
<td>selection screen</td>
</tr>
<tr>
<td>Web Dynpro ABAP</td>
</tr>
<tr>
<td>Field label</td>
</tr>
<tr>
<td>data element</td>
</tr>
<tr>
<td>Field symbol</td>
</tr>
<tr>
<td>dynamic programming</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>usage</td>
</tr>
<tr>
<td><strong>FIELD-SYMBOLS</strong></td>
</tr>
<tr>
<td>ABAP statement</td>
</tr>
<tr>
<td>File</td>
</tr>
<tr>
<td>application server</td>
</tr>
<tr>
<td>closing</td>
</tr>
<tr>
<td>deleting</td>
</tr>
<tr>
<td>opening</td>
</tr>
<tr>
<td>presentation server</td>
</tr>
<tr>
<td>reading</td>
</tr>
<tr>
<td>writing</td>
</tr>
<tr>
<td>File interface</td>
</tr>
<tr>
<td>AS ABAP</td>
</tr>
<tr>
<td>File name</td>
</tr>
<tr>
<td>logical</td>
</tr>
<tr>
<td>Filter condition</td>
</tr>
<tr>
<td>Query Service</td>
</tr>
<tr>
<td><strong>FINAL</strong></td>
</tr>
<tr>
<td>CLASS</td>
</tr>
<tr>
<td>METHODS</td>
</tr>
<tr>
<td>FIND</td>
</tr>
<tr>
<td>ABAP statement</td>
</tr>
<tr>
<td>Fixed value</td>
</tr>
<tr>
<td>domain</td>
</tr>
<tr>
<td>input help</td>
</tr>
<tr>
<td><strong>Fixed-Point Arithmetic</strong></td>
</tr>
<tr>
<td>program attribute</td>
</tr>
<tr>
<td>Fixture</td>
</tr>
<tr>
<td>ABAP Unit</td>
</tr>
<tr>
<td>Flat</td>
</tr>
<tr>
<td>data type</td>
</tr>
<tr>
<td>structure</td>
</tr>
<tr>
<td>Flat structure</td>
</tr>
<tr>
<td>assigning</td>
</tr>
<tr>
<td>Flight data model</td>
</tr>
<tr>
<td>dynpro</td>
</tr>
<tr>
<td>SAP</td>
</tr>
<tr>
<td>Floating point function</td>
</tr>
<tr>
<td>function</td>
</tr>
<tr>
<td>Floating point number</td>
</tr>
<tr>
<td>data object</td>
</tr>
<tr>
<td>floor</td>
</tr>
<tr>
<td>numeric function</td>
</tr>
<tr>
<td>Flow logic</td>
</tr>
<tr>
<td>dynpro</td>
</tr>
<tr>
<td>FOR ALL ENTRIES</td>
</tr>
<tr>
<td>SELECT</td>
</tr>
<tr>
<td>FOR EVENT</td>
</tr>
<tr>
<td>METHODS</td>
</tr>
<tr>
<td>Foreign key</td>
</tr>
<tr>
<td>database table</td>
</tr>
<tr>
<td>Foreign key dependency</td>
</tr>
<tr>
<td>database table</td>
</tr>
<tr>
<td>FORM</td>
</tr>
<tr>
<td>ABAP statement</td>
</tr>
<tr>
<td>Form field</td>
</tr>
<tr>
<td>URL</td>
</tr>
<tr>
<td>Formal parameter</td>
</tr>
<tr>
<td>operand position</td>
</tr>
<tr>
<td>optional</td>
</tr>
<tr>
<td>parameter interface</td>
</tr>
<tr>
<td>typing</td>
</tr>
<tr>
<td>usage</td>
</tr>
<tr>
<td><strong>FORMAT</strong></td>
</tr>
<tr>
<td>ABAP Statement</td>
</tr>
<tr>
<td>Forward navigation</td>
</tr>
<tr>
<td>ABAP Workbench</td>
</tr>
<tr>
<td>frac</td>
</tr>
<tr>
<td>numeric function</td>
</tr>
<tr>
<td>Frame</td>
</tr>
<tr>
<td>classical list</td>
</tr>
<tr>
<td>screen element</td>
</tr>
<tr>
<td>Framework</td>
</tr>
<tr>
<td>ABAP Objects</td>
</tr>
</tbody>
</table>
Index

Friend
class 190
FRIENDS
CLASS 190
Friendship
class 190
FROM clause
SELECT 713
Full-text search
ABAP keyword documentation 139
FUNCTION
ABAP statement 470
Function
built-in 287
Function Builder
tool 105, 462, 467
Function code
classical list 651
evaluating 538
exit command 546
Menu Painter 535
SAP GUI 534
selection screen 634
use 534
Function code assignment
Menu Painter 535
Function group
ABAP program 462
create 463
creation 101
global data 463
introducing 101
naming convention 465
program type 160
use 462
FUNCTION KEY
SELECTION-SCREEN 636
Function key assignments
GUI status 519
Function module
create 467
creation 105
dynamic 835
implementing 105
procedure 164, 462
release 471
source code 469
test 471
testing 108
Function pool
program type 160
Functional method
call 357
define 347
operand position 357
FUNCTION-POOL
ABAP statement 160, 466

G
Garbage Collector
ABAP runtime environment 212
GE
relational operator 293
Generalization
inheritance 361
GENERATE SUBROUTINE POOL
ABAP statement 837
Generic
data type 263
Generic data type
ABAP type hierarchy 229
generic typing
usage 354
GET
event 453
GET DATASET
ABAP statement 778
GET REFERENCE
ABAP statement 810
GET RUN TIME
ABAP statement 726
GET TIME STAMP
ABAP statement 292
GET/SET methods
persistent class 758
GET_PERSISTENT
Persistence Service 765
GET_PERSISTENT_BY_QUERY
Query Service 760
GET_PRINT_PARAMETERS
print parameters 655
Golden rule
checking 405
data reference variable 809
object reference variable 404
GROUP BY clause
SELECT 732
GT
  relational operator 293
GUI
  Graphical User Interface 513
GUI Control
  SAP GUI 587
GUI control
  dynpro 587
  event 595
  example 601
  lifetime 597
  methods 594
  processing 594
  wrapping 598
GUI status
  ABAP program 519
  checking 537
  classical list 651
  compare templates 537
  example 563
  functions 535
  selection screen 634
  setting 537
GUI title
  creating 538
GUI_DOWNLOAD
  writing a file 781
GUI_UPLOAD
  reading a file 781
GUID
  object identity 770

H

HANDLE
  ASSIGN 808
  CREATE DATA 813
HASHED TABLE
  TYPES/DATA 248, 320
Hashed table
  generic type 264
  table category 320
  use 322
HAVING clause
  SELECT 732
Header
  deep data object 973
Hello world
  ABAP program 88

Hiding
  data object 228
Host variable
  Native SQL 754
HTML
  interface 671
HTML GUI
  SAP GUI 516
HTTP body
  access 882
HTTP client
  ICF 888
HTTP communications
  AS ABAP 877
HTTP header
  access 882
HTTP request
  sending 890
HTTP request handler
  creating 880
  ICF 879
  implementing 882
  registering 880
HTTP server
  ICF 880
HTTP service
  creating 880
  testing 880
HTTP(S)
  protocol 877

I

message type 669
i
  ABAP type 236, 238
  calculation type 289
ICF
  AS ABAP 878
  Internet Communication Framework 877
  methods 884
  web service 897
ICF client
  programming 886
ICF Server
  programming 879
Index

ICM
  AS ABAP  878
  Internet Communication Manager  149, 877
ID
  XSLT program  921
IF
  ABAP statement  298
    control structure  298
IF_HTTP_CLIENT
  ICF  886
IF_HTTP_ENTITY
  ICF  882
IF_HTTP_EXTENSION
  ICF  879
IF_HTTP_HEADER_FIELDS
  ICF  882
IF_HTTP_HEADER_FIELDS_SAP
  ICF  882
IF_HTTP_RESPONSE
  ICF  882, 890
IF_OS_TRANSACTION
  Transaction Service  771
IF_SERIALIZABLE_OBJECT
  interface  412
    tag interface  925
IF_T100_MESSAGE
  message interface  496, 498, 667
IMPLEMENTATION
  CLASS  185
  Implementation
    method  195
  Implementation part
    ABAP program  87, 153
      class  184
IMPORT
  ABAP statement  786
IMPORTING
  Actual parameter  356
    METHOD  195
    METHODS  346
IN BYTE MODE
  byte string processing  304
IN CHARACTER MODE
  character string processing  304
IN PROGRAM
  PERFORM  476
IN seltab
  logical expression  627
predicate  296
WHERE clause  722
Inbound plug
  Web Dynpro ABAP  696
INCLUDE
  ABAP statement  167
  INCLUDE program
    ABAP program  167
      usage  168
    INCLUDE STRUCTURE
      ABAP statement  246
      INCLUDE TYPE
        ABAP statement  246
  Indentation
    pretty printer  86
  Independent interface reference variable
    ABAP Objects  383, 390
INDEX
  INSERT  327
  READ TABLE  329
  Index access
    internal table  321
  Index search
    ABAP keyword documentation  138
INDEX TABLE
  generic table type  324
  index table
    generic type  264
INDX-like
  database table  786
  Informational message
    message type  669
Inheritance
  ABAP Objects  27, 359
    independent interface  400
    object orientation  179
    polymorphism  414
Inheritance tree
  ABAP Objects  360
INHERITING FROM
  CLASS  362
  Initial dynpro
    dynpro sequence  520
INITIAL SIZE
  TYPES/DATA  325
  Initial value
    data object  285
INITIALIZATION
  event  452
Index

Initialization
  object  213
INNER JOIN
  SELECT  729
INPUT
  MODULE  542
Input and output parameter
    function module  468
Input check
  automatic  545
defining  547
selection screen  633
Input dialog
  Web Dynpro ABAP  689
Input format
dynpro  545
Input help
  automatic  550
defining  552
dynpro  549
hierarchy  551
selection screen  633
Web Dynpro ABAP  689
Input parameter
  formal parameter  346
  function module  468
  instance constructor  215, 375
Input stream object
tXML library  918
Input verification
  message  669
Input/output field
  screen element  527
Input/output parameter
  Formal parameter  346
INSERT dbtab
  ABAP statement  738
INSERT itab
  ABAP statement  327
INSERT REPORT
  ABAP statement  840
Inspection
  Code Inspector  948
Instance
  ABAP Objects  180
data object  230
Instance attribute
  creation  121, 192
Instance component  191
Instance constructor
  3-phase model  376
class  214
  exception  122
implementing  118, 122
inheritance  375
interface  121
Instance method
  creation  194
Instantiation
  ABAP Objects  27
  ABAP program  461
inheritance  380
Integer
  data object  236, 238
Integration broker
  web service  896
INTERFACE
  ABAP statement  384
Interface
  ABAP Objects  27, 381
class  187, 382
  composing  394
creating  384
implementing  386, 395
independent, usage  382
inheritance  400, 404
object orientation  343
polymorphism  414
user view  391
using  397
Interface component
  interface  384
static  394
Interface component selector
  interface  387
Interface method
  implementing  388
Interface parameter
  creation  114
  parameter interface  345
  parameter type  346
  transfer type  347
Interface pool
  program type  159
Interface reference variable
  ABAP Objects  383
Interface view
  web dynpro window  677
JCO.createClient  JCo 871
JCO.disconnect  JCo 871
JCO.Function  JCo 874
JCO.getClient  JCo 873
JCO.ParameterList  JCo 875
JCO.Pool  JCo class 872
JCO.PoolManager  JCo class 872
JCO.releaseClient  JCo 873
JCO.Repository  JCo 874
JCO.Server  JCo 876
JCO.Structure  JCo 875
JCO.Table  JCo 875
Job
  background request 641
  background processing 643
  JOB_CLOSE
    background processing 641
  JOB_OPEN
    background processing 641
  JOIN
    SELECT 727
  Join
    FROM clause 727
    linking 728
K
  Kernel
    AS ABAP 147
  Key access
    internal table 321
  Key attribute
    persistent class 759
  Keyword
    ABAP statement 85
Knowledge Warehouse
  using 138
L
  LDB_PROCESS
    function module 455
  LE
    relational operator 293
  LEAVE SCREEN
    ABAP statement 523
  LEAVE TO LIST-PROCESSING
    ABAP statement 648
  LEAVE TO SCREEN
    ABAP statement 523
  LEAVE TO TRANSACTION
    ABAP statement 521
  LEFT OUTER JOIN
    SELECT 729
  LENGTH
    TYPES 232
    TYPES/DATA 242
  Library
    Java 35
  LIKE
    TYPES/DATA 232
    WHERE clause 722
  Line
    classical list 646
    selection screen 628
  LINE OFF
    TYPES/DATA 328
  lines
    description function 287
  LINES OF
    APPEND 329
    INSERT itab 328
  Linked list
    example 815
  List
    ABAP Objects 658
    ALV list 659
    classical 645
    executable program 453
  List buffer
    classical list 646
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>List cursors</td>
<td>646</td>
</tr>
<tr>
<td>classical list</td>
<td>646</td>
</tr>
<tr>
<td>List dynpro</td>
<td>646</td>
</tr>
<tr>
<td>classical list</td>
<td>646</td>
</tr>
<tr>
<td>List event</td>
<td>646</td>
</tr>
<tr>
<td>event block</td>
<td>167</td>
</tr>
<tr>
<td>handling</td>
<td>652</td>
</tr>
<tr>
<td>List level</td>
<td>647</td>
</tr>
<tr>
<td>classical list</td>
<td>647</td>
</tr>
<tr>
<td>List output</td>
<td>69</td>
</tr>
<tr>
<td>creation</td>
<td>89</td>
</tr>
<tr>
<td>List processor</td>
<td>648</td>
</tr>
<tr>
<td>calling</td>
<td>648</td>
</tr>
<tr>
<td>classical list</td>
<td>646</td>
</tr>
<tr>
<td>Literal</td>
<td>268</td>
</tr>
<tr>
<td>data object</td>
<td>268</td>
</tr>
<tr>
<td>Literal XML element</td>
<td>929</td>
</tr>
<tr>
<td>Simple Transformation</td>
<td>929</td>
</tr>
<tr>
<td>LOAD-OF-PROGRAM</td>
<td>929</td>
</tr>
<tr>
<td>ABAP statement</td>
<td>166</td>
</tr>
<tr>
<td>event</td>
<td>452, 458</td>
</tr>
<tr>
<td>Local class</td>
<td>220</td>
</tr>
<tr>
<td>class pool</td>
<td>220</td>
</tr>
<tr>
<td>creation</td>
<td>126</td>
</tr>
<tr>
<td>definition</td>
<td>128</td>
</tr>
<tr>
<td>function group</td>
<td>465</td>
</tr>
<tr>
<td>implementation</td>
<td>128</td>
</tr>
<tr>
<td>LOCAL FRIENDS</td>
<td>220</td>
</tr>
<tr>
<td>CLASS</td>
<td>220</td>
</tr>
<tr>
<td>Local object</td>
<td>61</td>
</tr>
<tr>
<td>repository browser</td>
<td>61</td>
</tr>
<tr>
<td>Local type</td>
<td>220</td>
</tr>
<tr>
<td>class pool</td>
<td>220</td>
</tr>
<tr>
<td>Locale</td>
<td>151</td>
</tr>
<tr>
<td>text environment</td>
<td>151</td>
</tr>
<tr>
<td>lock</td>
<td>440</td>
</tr>
<tr>
<td>shared objects</td>
<td>440</td>
</tr>
<tr>
<td>Lock concept</td>
<td>748</td>
</tr>
<tr>
<td>AS ABAP</td>
<td>748</td>
</tr>
<tr>
<td>Lock object</td>
<td>748</td>
</tr>
<tr>
<td>SAP lock</td>
<td>748</td>
</tr>
<tr>
<td>Lock table</td>
<td>748</td>
</tr>
<tr>
<td>SAP lock</td>
<td>748</td>
</tr>
<tr>
<td>Log</td>
<td>510</td>
</tr>
<tr>
<td>assertion</td>
<td>510</td>
</tr>
<tr>
<td>log</td>
<td>287</td>
</tr>
<tr>
<td>floating point function</td>
<td>287</td>
</tr>
<tr>
<td>log10</td>
<td>287</td>
</tr>
<tr>
<td>floating point function</td>
<td>287</td>
</tr>
<tr>
<td>Logical database</td>
<td>163</td>
</tr>
<tr>
<td>program attribute</td>
<td>163</td>
</tr>
<tr>
<td>selection screen</td>
<td>616</td>
</tr>
<tr>
<td>Logical Database Builder</td>
<td>454</td>
</tr>
<tr>
<td>tool</td>
<td>454</td>
</tr>
<tr>
<td>Logical databases</td>
<td>454</td>
</tr>
<tr>
<td>use</td>
<td>454</td>
</tr>
<tr>
<td>Logical expression</td>
<td>292</td>
</tr>
<tr>
<td>ABAP syntax</td>
<td>292</td>
</tr>
<tr>
<td>IF/ELSEIF</td>
<td>299</td>
</tr>
<tr>
<td>WHERE</td>
<td>332</td>
</tr>
<tr>
<td>WHERE clause</td>
<td>721</td>
</tr>
<tr>
<td>Logical port</td>
<td>906</td>
</tr>
<tr>
<td>client proxy</td>
<td>906</td>
</tr>
<tr>
<td>Long text</td>
<td>666</td>
</tr>
<tr>
<td>LOOP</td>
<td>331</td>
</tr>
<tr>
<td>ABAP statement</td>
<td>331</td>
</tr>
<tr>
<td>Loop</td>
<td>331</td>
</tr>
<tr>
<td>conditional</td>
<td>301</td>
</tr>
<tr>
<td>control structure</td>
<td>301</td>
</tr>
<tr>
<td>internal table</td>
<td>331</td>
</tr>
<tr>
<td>unconditional</td>
<td>301</td>
</tr>
<tr>
<td>LOOP AT SCREEN</td>
<td>528</td>
</tr>
<tr>
<td>ABAP statement</td>
<td>528</td>
</tr>
<tr>
<td>LOOP WITH CONTROL</td>
<td>578, 581</td>
</tr>
<tr>
<td>dynpro statement</td>
<td>578, 581</td>
</tr>
<tr>
<td>Loops</td>
<td>129</td>
</tr>
<tr>
<td>executing</td>
<td>129</td>
</tr>
<tr>
<td>LOWER CASE</td>
<td>619</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>619</td>
</tr>
<tr>
<td>LT</td>
<td>293</td>
</tr>
<tr>
<td>relational operator</td>
<td>293</td>
</tr>
<tr>
<td>LUW</td>
<td>742</td>
</tr>
<tr>
<td>database</td>
<td>742</td>
</tr>
<tr>
<td>Logical Unit of Work</td>
<td>742</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main program</td>
<td>106, 463</td>
</tr>
<tr>
<td>Main session</td>
<td>174</td>
</tr>
<tr>
<td>Mapping</td>
<td>757</td>
</tr>
<tr>
<td>Markup element</td>
<td>910</td>
</tr>
<tr>
<td>Mathematical function</td>
<td>287</td>
</tr>
</tbody>
</table>
MAX
  aggregate function 715
  me
  data object 271
  self-reference 125, 205
Memory analysis
  ABAP Debugger 975
Memory area
  AS ABAP 173
Memory Inspector
  calling 977
tool 971
Memory leak
  causes 971
  example 979
  object 213
Memory object
  dynamic 972
Memory snapshot
  comparing 978
  create 975
  opening 977
  ranked list 978
Menu bar
  SAP GUI 516, 534
Menu Painter
tool 535
MESSAGE
  ABAP statement 667
Message
  dialog processing 668
  exception handling 503
  SAP GUI 666
  sending 667
  use 670
Message class
  message 666
Message number
  message 666
Message output
  creation 90
Message server
  AS ABAP 172
Message type
  message 668
Messages
  creating 666
tool 666
Metadata
  XML 909
  meth() 356
  Method call 356
METHOD
  ABAP statement 195
Method
  abstract 370
  call 355
  calling 129
  concrete 371
  creation 194
  declaring 113
  final 373
  functional 347
  implementing 117
  object orientation 178, 342
  parameter interface 345
  polymorphism 414
  procedure 164
  redefine 366
  source code 117
  subclass 366
Method call
  dynamic 832
  functional 357
  static 356
METHODS
  ABAP statement 194
MIME Repository
  Object Navigator 606
MIN
  aggregate function 715
  mod
  arithmetic operator 286
Model
  MVC 672, 681
Modeling
  object orientation 222
MODIFY dbtab
  ABAP statement 740
MODIFY itab
  ABAP statement 333
MODIFY SCREEN
  ABAP statement 528
Modularization
  internal 477
  procedural 460
Index

MODULE
  ABAP statement 542
  Dynpro statement 542
Module pool
  dialog programming 457
  program type 161
Module pools
  use 460
Module test
  ABAP Unit 961
  analysis 969
  definition 962
Mouse
  double-click 536
MOVE
  ABAP statement 273
  MOVE ?TO
    ABAP statement 409, 817
  MOVE-CORRESPONDING
    ABAP statement 283
MS Windows
  SAP GUI 516
Multiple inheritance
  ABAP Objects 382
  object orientation 344
Multiple instantiation
  class 207
Multiple selection
  selection criterion 626
MULTIPLY
  ABAP statement 286
MVC
  Model View Controller 145, 515, 672
  mySAP Business Suite
      product family 141

X
n
  ABAP type 236, 239
NA
  relational operator 296, 316
Namespace
  asXML 922
      class 191
      data object 228
      data type 234
      inheritance 365
      Simple Transformation 929
  XML 912
  Naming conventions
      class 185
      customer system 63
Native SQL
  AS ABAP 753
  Native SQL interface
      AS ABAP 149
Navigation link
  Web Dynpro ABAP 697
NE
  relational operator 293
NEW-PAGE PRINT ON
  ABAP statement 655
Next dynpro
  calling 521
      dynpro property 526
      dynpro sequence 519
NO INTERVALS
  SELECT-OPTIONS 624
NO-EXTENSION
  SELECT-OPTIONS 624
Non-class-based exception
  define 502
NOT
  Boolean operator 297
  WHERE clause 721
NP
  relational operator 296, 316
NS
  relational operator 296, 316
Numeric
  data type 237
  numeric
      generic type 264
  Numeric function
      function 287
  Numeric literal
  data object 268
  Numeric text field
      data object 236, 239

O
O/R mapping
  object-relational mapping 757
OASIS
  Organization for the Advancement of
      Structured Information Systems 893
Object

ABAP Objects 180
asXML 925
Creating 128
dynamic memory object 972
generic type 408
object orientation 341
real world 177
root class 360
software 178
Object component selector
ABAP syntax 204
using 250
Object ID
Object Services 757
Object list
Repository Browser 60
Object list type
Repository Browser 60
Object Navigator
ABAP Workbench 57
Object orientation
programming 177
Object reference
ABAP Objects 402
internal table 208
memory address 202
persistent 770
Object reference variable
declaring 250
golden rule 404
user view 403
Object reference variables
creation 202
Object Services
database access 757
Object set
Code Inspector 948
Object type
ABAP type hierarchy 230
Object-oriented transaction mode
Transaction Service 774
OBLIGATORY
PARAMETERS 619
Obsolete language element
ABAP 35
Offset/length specification
subfield access 313
OK field
dynpro field 533
use 538
ON
JOIN 728
ON BLOCK
AT SELECTION-SCREEN 632
ON CHAIN-INPUT
MODULE 545
ON CHAIN-REQUEST
MODULE 545
ON COMMIT
PERFORM 747
ON END OF
AT SELECTION-SCREEN 632
ON EXIT-COMMAND
AT SELECTION-SCREEN 632
ON HELP-REQUEST
AT SELECTION-SCREEN 632
ON INPUT
MODULE 544
ON para|selcrit
AT SELECTION-SCREEN 632
ON RADIOBUTTON GROUP
AT SELECTION-SCREEN 632
ON REQUEST
MODULE 544
ON ROLLBACK
PERFORM 747
ON VALUE-REQUEST
AT SELECTION-SCREEN 632
OO transaction
creation 130
dynpro 560
transaction code 158
OOA
object-oriented analysis 180
OOD
object-oriented design 180
OPEN CURSOR
ABAP statement 735
OPEN DATASET
ABAP statement 776
Open SQL
ABAP statements 710
dynamic 830
performance 711
using 95
## Index

Open SQL interface

- AS ABAP 149

Operand

- ABAP statement 85
- arithmetic expression 287
- logical expression 293
- specified dynamically 830

Operand position

- ABAP statement 226

Operator

- ABAP statement 85
- arithmetic expression 286

OPTIONAL

- METHODS 354
- OR
  - Boolean operator 297
  - WHERE clause 721

ORDER BY clause

- SELECT 734

OTR

- Online Text Repository 496

Outbound plug

- Web Dynpro ABAP 696

OUTPUT

- AT SELECTION-SCREEN 631
- MODULE 542

Output field

- selection screen 628

Output parameter

- event 425
  - Formal parameter 346
    - function parameter 348

OVERLAY

- ABAP statement 305

### P

<table>
<thead>
<tr>
<th>p</th>
<th>ABAP type 236, 238</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>ABAP Workbench 60</td>
</tr>
<tr>
<td>Package check</td>
<td>package 67</td>
</tr>
<tr>
<td>Package interface</td>
<td>package 67</td>
</tr>
<tr>
<td>Packaged</td>
<td>verification 62</td>
</tr>
<tr>
<td>Package property</td>
<td>package 64</td>
</tr>
<tr>
<td>PACKAGE SIZE</td>
<td>INTO clause 718</td>
</tr>
<tr>
<td>Package type</td>
<td>package 64</td>
</tr>
<tr>
<td>Packed number</td>
<td>data object 236, 238</td>
</tr>
<tr>
<td>PAI</td>
<td>dynpro event 458</td>
</tr>
<tr>
<td>Event 538</td>
<td>PROCESS AFTER INPUT 517</td>
</tr>
<tr>
<td>selection screen 631</td>
<td>PAI module</td>
</tr>
<tr>
<td>function group 465</td>
<td>Parallel processing</td>
</tr>
<tr>
<td>aRFC 856</td>
<td>Parameter</td>
</tr>
<tr>
<td>selection screen 618</td>
<td>Parameter interface</td>
</tr>
<tr>
<td>event 425</td>
<td>exception 490</td>
</tr>
<tr>
<td>executable program 640</td>
<td>function module 467</td>
</tr>
<tr>
<td>method 195, 196</td>
<td>subroutine 474</td>
</tr>
<tr>
<td>Parameter transaction</td>
<td>dialog transaction 457</td>
</tr>
<tr>
<td>Parameter transfer</td>
<td>performance 349</td>
</tr>
<tr>
<td>PARAMETER-TABLE</td>
<td>ABAP statement 618</td>
</tr>
<tr>
<td>CALL FUNCTION 835</td>
<td>CALL METHOD 833</td>
</tr>
<tr>
<td>Parentheses</td>
<td>calculation expression 286</td>
</tr>
<tr>
<td>logical expressions 297</td>
<td>Pass by value</td>
</tr>
<tr>
<td>formal parameter 348</td>
<td>Patterns</td>
</tr>
<tr>
<td>object orientation 180</td>
<td>PRO</td>
</tr>
<tr>
<td>dynpro event 458</td>
<td>PROCESS BEFORE OUTPUT 517</td>
</tr>
<tr>
<td>selection screen 631</td>
<td>PRO module</td>
</tr>
<tr>
<td>function group 465</td>
<td></td>
</tr>
</tbody>
</table>
Index

Processing block
  ABAP program 87
  dynpro flow logic 517
  implementation 153
  terminate 156
Processor
  ABAP runtime environment 150
Production system
  CTS 62
Productive part
  ABAP program 966
  PROGRAM
  ABAP statement 160, 161
Program call
  dynamic 830
Program check
  extended 942
Program constructor
  event block 166
Program generation
  persistent 840
  transient 837
  usage 836
Program group
  internal mode 476
Program introduction
  ABAP program 86, 87
Program properties
  definition 83
Program type
  ABAP program 159
  recommendation 162
Programming
  defensive 223, 479
  robust 479
Programming guidelines
  ABAP 46
Programming languages
  AS ABAP 147
Programming model
  object-oriented 26
  procedural 26
Protected
  inheritance 365
  visibility area 186
Protected area
  TRY block 484
Protected instantiation
  superclass 380
PROTECTED SECTION
  visibility area 187
Protocol 845
Proxy
  web service 896
Proxy object
  CFW 589
Pseudo comment
  extended program check 945
Public
  inheritance 365
  visibility area 186
Public instantiation
  superclass 380
PUBLIC SECTION
  ABAP statement 186
Publish-and-Subscribe
  event 423
PUSHBUTTON
  SELECTION-SCREEN 628
Pushbutton
  selection screen 628

Q

qRFC
  API 861
  executing 861
  queued RFC 847
  RFC API 868
  scenarios 848
qRFC manager
  qRFC 861
Quality management
  Code Inspector 950
Query Manager
  Object Services 760
Query Service
  Object Services 760

R

R/2
  system 23
R/3
  product family 141
  system 24, 141
Radio button
  screen element 527
Index

selection screen 619
RADIOBUTTON GROUP PARAMETERS 619
RAISE EVENT
ABAP statement 426
RAISE exc
ABAP statement 502
RAISE EXCEPTION
ABAP statement 483
RAISING
FORM 474
MESSAGE 503
METHOD 195
METHODS 490
RANGE
ASSIGN 805
RANGE OF
DATA 627
READ DATASET
ABAP statement 777
Read lock
shared objects 440
READ REPORT
ABAP statement 838
READ TABLE
ABAP statement 329
READ-ONLY
DATA 192, 243
RECEIVE RESULTS
ABAP statement 855
RECEIVING
Actual parameter 357
REDEFINITION
METHODS 367
REF TO
DATA 202
TYPES/DATA 249, 402, 809
REFERENCE
METHODS 347
Reference
data type 231
XML 926
REFERENCE INTO
LOOP 331
READ TABLE 330
Reference semantics
assignment 261
data reference 796
dynamic memory object 974
Reference transfer
formal parameter 348
Reference type
ABAP Dictionary 257
declaring 249
Reference variable
assigning 205, 282, 405
comparing 295
declaring 249
initial value 285
using 117, 250
REGEX
FIND 310
REPLACE 311
Registering event handlers 431
Registration
event 423
Regular expression
class 312
find/replace 308
special character 309
Relational operator
logical expression 293
Remote Function Call
AS ABAP 149
RFC 845
Remote-enabled function module
RFC 853
RFM 853
Rental car application
every 54
REPLACE
ABAP statement 306
REPORT
ABAP statement 160
Report
creation 134
Report transaction
executable program 631
Reporting
classic ABAP 451
interactive 652
process 454
programming 133
Reporting event
event block 166
Repository
development objects 59
Index

Repository browser
  Object Navigator 59
Repository information system
  Object Navigator 59
Repository object
  ABAP Workbench 59
Required field
  dynpro 545
  selection screen 619
RETURN
  ABAP statement 156, 303
  Return value
    formal parameter 346
RETURNING
  METHOD 195
  METHODS 347
RFC 849
  API 862
  asynchronous 846
  communication scenarios 849
  debugging 855
  object-oriented control 862
  programming 853
  queued 847
  Remote Function Call 845
  synchronous 845
  transactional 847
RFC API
  C routines 862
RFC client
  JCo 870
  non-SAP system 863
RFC destination
  administering 850
  HTTP connection 889
  non-SAP system 866
  specifying 850, 854
RFC interface
  AS ABAP 845
  external 850
RFC library
  RFC API 862
RFC SDK
  downloading 863
  Software Development Kit 850
RFC server
  JCO 876
  non-SAP system 865
  passing parameters 866
  RfcAccept
    RFC API 865
  RfcCall
    RFC API 864
  RfcClose
    RFC API 864
  RfcCreateTransID
    RFC API 868
  RfcDispatch
    RFC API 868
  RfcGetData
    RFC API 865
  RfcGetName
    RFC API 865
  RfcIndirectCallEx
    RFC API 868
  RfcInstallFunction
    RFC API 865
  RfcInstallTransactionControl
    RFC API 868
  RfcLibrary
    external RFC interface 862
  RfcOpen
    RFC API 864
  RfcReceive
    RFC API 864
  RfcSendData
    RFC API 865
RFM
  JCo call 876
Roll area
  internal session 175
Rollback
  database 742
ROLLBACK WORK
  ABAP statement 746
  Persistence Service 770
Root class
  inheritance tree 360
Root object
  shared objects 442
Row type
  internal table 248, 319
RTTC
  Run Time Type Creation 819
RTTI
  Run Time Type Information 819
RTTS
  ASSIGN 808
Search help
  ABAP Dictionary 550
  creating 550
  creation 78
  using 108
Search help maintenance
  ABAP Dictionary 78
Secondary index
database table 723
SELECT
  ABAP statement 712
  assignment rules 720
  loop 717
SELECT clause
  SELECT 713
SELECT loop
  nested 725
  using 118
Selection criterion
  selection screen 622
Selection screen
calling 630
  creating 617
  creation 103
  Data Browser 98
dynpro 615
  event 631
  GUI status 634
  processing 104, 631
  processor 631
  quitting 635
  use 617
Selection screen event
  event block 166
Selection screen processing
  silent 641
Selection table
  selection criterion 623
  WHERE clause 724
Selection text
  Creating 104
  Selection screen 619
SELECTION-SCREEN BEGIN OF SCREEN
  ABAP statement 617
SELECT-OPTIONS
  ABAP statement 622
Self-reference
  instance constructor 377
sender
  event parameter 425, 429
Separation of concerns
  classical list 650, 658
  concept 514
  dynpro 557
  selection screen 617
  web service 885
Serialization
  ABAP-XML 908
  ST 927
  XSLT 921
Service
  web-based 878
  Service Definition Wizard
    web service 898
  Service Wizard
    ICF 880
Services
  use 30
Session breakpoint
  ABAP Debugger 956
SET DATASET
  ABAP statement 778
SET EXTENDED CHECK
  ABAP statement 945
SET HANDLER
  ABAP statement 431
SET method
class 187
SET PF-STATUS
  ABAP statement 537
SET SCREEN
  ABAP statement 522
SET TITLEBAR
  ABAP statement 538
Shared memory
  application server 173, 433
  SHARED MEMORY ENABLED
    CLASS 435
Shared Memory-enabled
class 435
Shared Objects
  AS ABAP 433
  Object orientation 344
Shared objects
  access 436
    creating 442
    object reference 436
Structure component
  assigning 283
  integrating 246
  using 245
Structure component selector
  ABAP syntax 245
Subclass
  component 361
  create 362
  implicit 361
  inheritance 360
Subfield
  access 313
  Subfield addressing
    field symbol 800
SUBMIT
  ABAP statement 157, 451
  standard selection screen 639
Subquery
  SELECT 731
Subroutine
  function group 465
  procedure 164
  use 476
Subroutine call
  dynamic 836
Subroutine pool
  creation 126
  program type 160, 476
Subroutines
  procedure 474
SUBSCREEN
  SELECTION-SCREEN 628
Subscreen
  dynpro 581
  screen element 527
Subscreen dynpros
  tabstrip page 582
Subscreen-dynpro
  selection screen 628
SUBSTRING
  FIND/REPLACE 307
Substring
  find/replace 307
Substructure
  structure 244
SUBTRACT
  ABAP statement 286
Subtransaction
  Transaction Service 773
SUM
  aggregate function 715
  super-> 367
  super->constructor
  inheritance 375
Superclass
  inheritance 360
  private 381
SUPPRESS DIALOG
  ABAP statement 649
sy
  structure 272
  sy-dbcnt
  system field 713
  sy-dynnr
  system field 521
Symbolic name
  field symbol 797
Syntax
  checking 941
  Syntax check
    ABAP program 88
  Syntax cleansing
    ABAP Objects 28, 181
  Syntax convention
    use 47
Syntax diagram
  ABAP keyword documentation 137
Syntax error
  ABAP program 88
  syntax check 942
Syntax warning
  syntax check 942
SYST
  structure 272
System codepage
  text environment 151
System data container
  eCATT 987
System event
  GUI control 596
System field
  data object 271
System library
  ABAP 25
Index

SYSTEM_FAILURE
  RFC 855
sy-subrc
  system field 272
sy-tabix
  system field 327, 332
sy-ucomm
  system field 533

T
  ABAP type 236, 240
T100
  database table 666
Tab
  dynpro 581
Tab strips
  screen element 527
TABLE
  INSERT 327
table
  generic type 264
Table category
  internal table 248, 320
Table control
  creating 574
dynpro 574
  paging 580
wizard 576
Table controls
  screen element 527
Table definition
  ABAP Dictionary 70
Table index
  internal table 321
TABLE KEY
  READ TABLE 329
Table key
  database table 71
defining 323
  internal table 249, 321
Table maintenance
  ABAP Dictionary 70
TABLE OF
  TYPES/DATA 248
Table parameter
  function module 468
subroutine 475

Table type
  ABAP Dictionary 258
  ABAP program 248
generic 264
table_line
  pseudo component 323
TABLES
  ABAP statement 532
FORM 475
TABLEVIEW
  CONTROLS 579
TABSTRIP
  CONTROLS 583
Tabstrip control
dynpro 581
  selection screen 628
wizard 583
Tag
  XML 910
Tag Browser
  Object Navigator 929
tan
  floating point function 287
tanh
  floating point function 287
Target system
  CTS 62
Task
  transport request 67
TCP/IP
  protocol 845
Termination message
  message type 670
Test class
  ABAP Unit 963
creation 967
Test configuration
eCATT 987
Test data container
eCATT 987
Test hierarchy
  ABAP Unit 963
Test method
  ABAP Unit 963
Test methods
  creation 967
Test part
  ABAP program 966
Index

Test property
  ABAP Unit 967
Test run
  ABAP Unit 969
Test script
  eCATT 987
Test systems
  CTS 62
Test task
  ABAP Unit 963
Testing
  tools 939
Testing procedure
  static 941
Text element
  translating 268
  using 103
Text element maintenance
  tool 103, 267
Text environment
  ABAP runtime environment 151
  AS ABAP 26
Text field
  data object 236, 239
  screen element 527
Text field literal
  data object 269
Text pool
  text environment 151
Text string
  data object 241
Text symbol
  creation 103
  data object 267
  text element 267
  using 103
Textedit control
  CFW 592
  use 659
Time field
  calculating 290
  comparing 294
  data object 237, 240
  validity 291
Time stamp
  date and time 292
TIMES
  DO 301

Tips & Tricks
  runtime analysis 984
Title bar
  SAP GUI 516, 534
  TO SAP-SPOOL
  SUBMIT 641, 655
Token
  ABAP statement 85
  specified dynamically 829
Tool area
  Object Navigator 59
  Toolbar control
    CFW 591
  TOP include
    function group 101
  Top include
    function group 465
    include program 168
  TOP-OF-PAGE
    List event 646
Transaction
  execute 131, 158
  nesting 773
  program execution 130
  SAP LUW 747
Transaction code
  development object 157
  dialog transaction 457, 521
  rTFC 859
  SAP Easy Access 56
  transaction 130
Transaction manager
  Transaction Service 771
Transaction mode
  Transaction Service 774
Transaction Service
  Object Services 771
  use 771
TRANSFER
  ABAP statement 777
Transformation Editor
  tool 920, 928
  transient attribute
    persistent class 769
TRANSLATE
  ABAP statement 305
Transport
  repository object 62
Transport layer
CTS 63
package 64
Transport Organizer
CTS 63, 67
Transport request
CTS 65
TRDIR
system table 840
Tree control
CFW 592
example 606
tRFC
executing 859
RFC API 868
status 861
transactional RFC 847
use 847
TRFC_SET_QUEUE_NAME
qRFC 861
true
logical expression 292
trunc
numeric function 287
TRY
ABAP statement 484
control structure 484
TRY block
TRY control structure 484
TT:cond 936
TT:include 932
TT:loop 934
TT:parameter 932
TT:ref 932
TT:root 931
TT:switch 936
TT:template 931
TT:transform 931
TT:type 932
TT:value 933
TT:variable 932
TYPE
CREATE OBJECT 403
DATA 229, 232
METHODS 351
TYPES 232
Type class
RTTS 819
Type conversion
assignment 274
operand position 274
Type group
ABAP Dictionary 259
program type 161
Type hierarchy
ABAP 229
Type name
absolute 823
Type object
create 825
RTTC 825
RTTS 819
Type of instantiation
class 188
Type specification
dynamic 830
TYPE TABLE OF
TYPES/DATA 324
TYPE-POOL
ABAP statement 161, 260
TYPES
ABAP statement 200, 232
Typing
complete 352
define 351
function module parameters 468
generic 263, 352
subroutine parameters 474
U
UDDI
server 905
Universal Description, Discovery and Integration 893
UDDI registry
web service 905
ULINE
ABAP statement 646
SELECTION-SCREEN 628
UN/CEFACT
United Nations Center for Trade Facilitation and Electronic Business 894
UNASSIGN
ABAP statement 806
Unicode
SAP system 24
Unicode checks active
program attribute 163
Unicode fragment view
structure 280
Unicode program
ABAP program 163
Unicode programs
byte and character string processing 304
Unicode system
AS ABAP 151
UNIQUE KEY
TABLE 323
Up cast
data reference variable 817
inheritance 406
interface 407
interface reference variable 391
object reference variable 406
UPDATE dbtab
ABAP statement 739
Update function module
updating 745
Update lock
shared objects 440
UPDATE TASK
CALL FUNCTION 745
Update work process
AS ABAP 746
Updating
SAP LUW 745
URI
Uniform Resource Identifier 913
URL
access 882
Usage type
SAP NetWeaver 142
User breakpoint
ABAP Debugger 957
User dialog
creation 99
decoupling 100
User interface
ABAP 513
AS ABAP 145
User menu
SAP Easy Access 55
User session
application server 174
USER-COMMAND
SELECTION-SCREEN 635
USING
FORM 474
PERFORM 475
UTC
coordinated universal time 292
V
VALUE
CONSTANTS 194, 266
DATA 243
METHODS 347
VALUE CHECK
PARAMETERS 619
Value list
dropdown list box 554
Value range
domain 75, 80
dynpro 545
Value semantics
assignment 261
dynamic data object 973
field symbol 796
Variable
data object 266
Variant
runtime analysis 982
Variant transaction
dialog transaction 457
VIA JOB
SUBMIT 451, 641
VIA SELECTION-SCREEN
SUBMIT 630
View
creation 709
database view 709
MVC 672, 681
Web Dynpro ABAP 675
View context
web dynpro view 685
View Designer
Web Dynpro ABAP 676
View layout
web dynpro view 676, 687
Index

View navigation
Web Dynpro ABAP 695
Visibility area
class 186
inheritance 364

Web Dynpro ABAP 695

dynpro control 574
dynpro context 683
Web Dynpro controller 683
Web Dynpro controller tool 674
Web Dynpro Explorer tool 677
Web Dynpro Explorer window navigation 697
Web Dynpro Explorer tool 674
Web Dynpro view navigation 697
Web Dynpro window Web Dynpro ABAP 677

Web Dynpro ABAP example application 702
use 671
Web Dynpro application executing 679
Web Dynpro ABAP 679, 693
Web Dynpro component
Web Dynpro ABAP 673, 682
Web Dynpro component context web dynpro controller 683
Web Dynpro controller tool 674
Web Dynpro context
Web Dynpro view navigation 697
Web Dynpro window Web Dynpro ABAP 677
Web reports example 882
Web service
ABAP 891
AS ABAP 894
creating 898
Enterprise SOA 892
publishing 904
releasing 900
service provider 895
service requester 895
standardization 893
testing 902
UDDI registry 895
using 891
Web service client creating 905
Web Service Framework AS ABAP 894, 897
J2EE server 902
Web service home page
Web Service Framework 902
WHEN
ABAP statement 299
WHERE
DELETE itab 334
LOOP 332
MODIFY itab 333
WHERE clause
SELECT 721
usage 723
WHILE
ABAP statement 301
Window Editor
Web Dynpro Explorer 677
Windows
SAP GUI 516
WITH
SUBMIT 641
WITH KEY
READ TABLE 330
Wizard
dynpro control 574
dynpro context 683
web service 898
Work process
application server 172
database logon 172
database LUW 743
WRITE
ABAP statement 645
eexecutable program 647
Write lock
shared objects 440
WRITE TO
ABAP statement 284
WSADMIN
transaction 902
WSCONFIG
  transaction 900
WSDL
  Web Services Description Language 893
WSDL document
displaying 904
URL 905
WS-I
  Web Service Interoperability
    Organization 894

X
  message type 670
  ABAP type 236, 241
XI
  SAP NetWeaver Exchange Infrastructure 895
XML 908
  AS ABAP 908
  CALL TRANSFORMATION 920
document 911
  Extensible Markup Language 909
XML document
tree representation 914
  well-formed 914
  XML parser
    iXML Library 913
  XML renderer
    iXML Library 913
  xmlns
    XML namespace 913
  xsequence
generic type 264
XSLT
  DOM 920
  Extensible Stylesheet Language
    Transformations 918
  XSLT processor
    AS ABAP 147, 919
  XSLT program
calling 920
  creating 920
  program generation 919
  repository object 919
  xstring
    ABAP type 242

Z
  Z_ABAP_BOOK
    package 63