

TAW12_1

ABAP Objects and Application Areas

SAP NetWeaver

Date _____
Training Center _____
Instructors _____

Education Website _____

Participant Handbook

Course Version: 63
Course Duration: 5 Day(s)
Material Number: 50090660



An SAP course - use it to learn, reference it for work

Copyright

Copyright © 2008 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Trademarks

- Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of Microsoft Corporation.
- IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.
- ORACLE® is a registered trademark of ORACLE Corporation.
- INFORMIX®-OnLine for SAP and INFORMIX® Dynamic ServerTM are registered trademarks of Informix Software Incorporated.
- UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.
- Citrix®, the Citrix logo, ICA®, Program Neighborhood®, MetaFrame®, WinFrame®, VideoFrame®, MultiWin® and other Citrix product names referenced herein are trademarks of Citrix Systems, Inc.
- HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.
- JAVA® is a registered trademark of Sun Microsystems, Inc.
- JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.
- SAP, SAP Logo, R/2, RIVA, R/3, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

Disclaimer

THESE MATERIALS ARE PROVIDED BY SAP ON AN "AS IS" BASIS, AND SAP EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR APPLIED, INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THESE MATERIALS AND THE SERVICE, INFORMATION, TEXT, GRAPHICS, LINKS, OR ANY OTHER MATERIALS AND PRODUCTS CONTAINED HEREIN. IN NO EVENT SHALL SAP BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR PUNITIVE DAMAGES OF ANY KIND WHATSOEVER, INCLUDING WITHOUT LIMITATION LOST REVENUES OR LOST PROFITS, WHICH MAY RESULT FROM THE USE OF THESE MATERIALS OR INCLUDED SOFTWARE COMPONENTS.

About This Handbook

This handbook is intended to complement the instructor-led presentation of this course, and serve as a source of reference. It is not suitable for self-study.

Typographic Conventions

American English is the standard used in this handbook. The following typographic conventions are also used.

Type Style	Description
<i>Example text</i>	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths, and options. Also used for cross-references to other documentation both internal (in this documentation) and external (in other locations, such as SAPNet).
Example text	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
Example text	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.
Example text	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
< Example text >	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Icons in Body Text

The following icons are used in this handbook.

Icon	Meaning
	For more information, tips, or background
	Note or further explanation of previous point
	Exception or caution
	Procedures
	Indicates that the item is displayed in the instructor's presentation.

Contents

Course Overview	vii
Course Goals.....	vii
Course Objectives	viii
Unit 1: Introduction to Object-Oriented Programming	1
The Object-Oriented Programming Model	3
Analysis and Design with UML.....	15
Fundamental Object-Oriented Syntax Elements	37
Unit 2: Object-Oriented Concepts and Programming Techniques.....	101
Inheritance and Casting.....	103
Interfaces and Casting.....	145
Events.....	184
Unit 3: Object-Oriented Repository Objects	225
Global Classes and Interfaces	227
Special Object-Oriented Programming Techniques.....	266
Unit 4: Class-Based Exception Concept	291
Class-Based Exceptions.....	292
Unit 5: Shared Objects	327
Shared Objects	328
Unit 6: Dynamic Programming	361
Dynamic Programming with Field Symbols and References	362
Runtime Type Services	381
Index	405

Course Overview

This two-week training course provides a comprehensive and detailed introduction to the basic principles of programming ABAP objects. You will also learn how to make specialized changes to the SAP standard system. Furthermore, you will find out how to evaluate the different methods for modification and choose the right one for the given situation. Web Dynpro is SAP's state-of-the-art technology for creating application user interfaces (UIs). This course explains in detail how to develop ABAP Web-Dynpro-based applications.

Target Audience

This course is intended for the following audiences:

- Development consultants who are responsible for adapting and developing ABAP/ABAP Objects programs

Course Prerequisites

Required Knowledge

- TAW10 (ABAP Workbench Fundamentals)
- Included in booking: TAW11 E-Learning (ABAP Details)

Recommended Knowledge

- BC410 - Programming User Dialogs with Dynpro



Course Goals

This course will prepare you to:

- Use fundamental elements of object-oriented modeling in UML
- Create ABAP Objects programs that contain all useful object-oriented programming techniques
- Use the relevant tools to create object-oriented Repository objects
- Describe and use the application areas of ABAP Objects
- Define, raise, and handle class-based exceptions
- Query type and class attributes at runtime
- Make specialized changes to the SAP standard system
- Evaluate the different methods for modification and choose the right one for any given situation
- Explain the architecture of a Web Dynpro component
- Describe the parts of a Web Dynpro controller
- Create context elements in the Web Dynpro controller
- Explain how to implement navigation and data transfer in and between Web Dynpro components
- Define the UI of a Web Dynpro component
- Internationalize a Web Dynpro application.
- Define and send messages in a Web Dynpro component
- Define input help and semantic help for UI elements in a Web Dynpro component



Course Objectives

After completing this course, you will be able to:

- Use fundamental elements of object-oriented modeling in UML
- Create ABAP Objects programs that contain all useful object-oriented programming techniques
- Use the relevant tools to create object-oriented Repository objects
- Describe and exploit the range of applications of ABAP Objects
- Define, raise, and handle class-based exceptions
- Query type and class attributes at runtime
- Make specialized changes to the SAP standard system
- Evaluate the different methods for modification and choose the right one for any given situation

- Explain the architecture of a Web Dynpro component
- Describe the parts of a Web Dynpro controller
- Create context elements in the Web Dynpro controller
- Explain how navigation and data transfer in and between Web Dynpro components can be implemented
- Define the UI of a Web Dynpro component
- Internationalize a Web Dynpro application
- Define and send messages in a Web Dynpro component
- Define input help and semantic help for UI elements in a Web Dynpro component

SAP Software Component Information

The information in this course pertains to the following SAP Software Components and releases:

- SAP NetWeaver Application Server 7.0
- SAP Web Application Server 6.20

Unit 1

Introduction to Object-Oriented Programming

Unit Overview

This unit will introduce you to the basics of object-oriented software development. The first lesson introduces the new ways of thinking and the related concepts.

The second lesson is a compact introduction to modeling, the step in the software development process that immediately proceeds the actual programming. This will be demonstrated using the modeling standard UML. To begin with, you will only see the most basic and important elements. The lessons will successively build upon this information, introducing the object-oriented programming concepts and the related UML notations in parallel. Therefore, this course will simultaneously teach you object-oriented modeling and programming.

The contents of the first two lessons are essentially also applicable to other modern object-oriented languages. As of the third lesson, syntax elements that are specific to ABAP Objects are introduced. You will need to learn many of these syntax elements in short succession.

Most of the content of later units deals with concepts that will be entirely new to you. Syntax will play a much smaller role in these units.



Unit Objectives

After completing this unit, you will be able to:

- Explain the differences between procedural and object-oriented programming models
- List the advantages of the object-oriented programming model
- Name the most important diagram types in UML
- Create simple class diagrams
- Create simple object diagrams
- Describe sequence diagrams
- Define classes

- Generate and delete objects
- Access attributes
- Call methods

Unit Contents

Lesson: The Object-Oriented Programming Model	3
Lesson: Analysis and Design with UML	15
Exercise 1: UML Class Diagrams	29
Lesson: Fundamental Object-Oriented Syntax Elements	37
Exercise 2: Local Classes.....	65
Exercise 3: Objects	71
Exercise 4: Method Calls.....	75
Exercise 5: Constructors	81
Exercise 6: Private Methods	87

Lesson: The Object-Oriented Programming Model

Lesson Overview

Based on your existing knowledge of procedural programming with ABAP, we will explain the object-oriented approach and encourage you to use it. The main emphasis will be on explanation. At this stage, it is important to introduce you to the subject and its concepts, so that we can build on this knowledge later on.

For now, it would not make sense to try to argue a conclusive case for or against the object-oriented approach. Before you can make a qualified decision, you need to get to know all object-oriented concepts and their advantages and disadvantages.

Even if this sort of decision is made within your team or by your development manager, you should be able to contribute to the discussion.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the differences between procedural and object-oriented programming models
- List the advantages of the object-oriented programming model

Business Example

You need to explain the basics of the object-oriented programming model and its advantages to your development project manager.

Moving from the Procedural to the Object-Oriented Programming Model

As we can see from the programming language Simula 67, object-oriented programming was developed at approximately the same time as the logical and procedural programming models. In the past, COBOL and, above all, the procedural programming model as expressed in languages like C or Pascal were dominant in enterprise application development. Before ABAP, SAP originally used a macro assembler.

Even today, many developers still have more experience with procedural programming than object-oriented programming. Therefore, this introduction to object-oriented programming also uses references to the procedural model in its explanations.

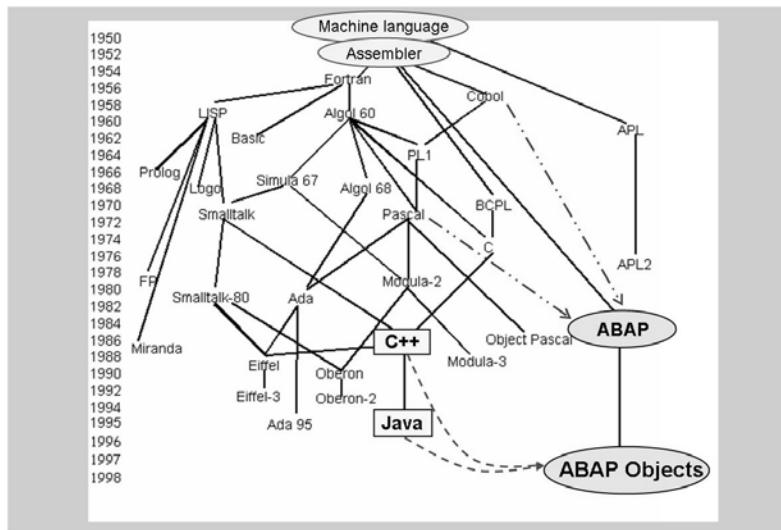
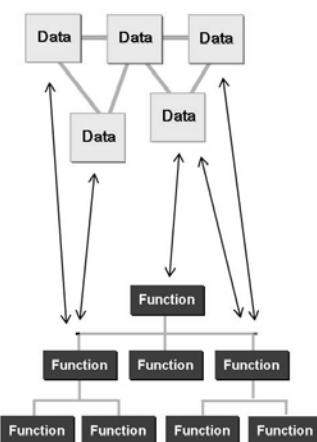


Figure 1: History of Selected Programming Languages

ABAP was created with the intention of improving reporting. It was developed relatively independently as an in-house programming language, although it was influenced by other programming languages like COBOL and PASCAL. ABAP/4 was then extended to form ABAP Objects. Therefore, ABAP Objects unites object-oriented and procedural elements in one programming language.

For the object-oriented part, only those object-oriented concepts that had proved their worth for enterprise application development in other languages, such as Java, C++, and Smalltalk, were adopted. ABAP Objects, like ABAP/4 before it, also contains some unique, very advantageous concepts.



- Separation of data and functions
- Usually non-encapsulated access to data
- Possibility of encapsulating functions using modularization

Figure 2: Characteristics of the Procedural Programming Model

Data and functions are usually kept separate in the procedural programming model. Global variables for a program usually contain the data, while subroutines contain the functions.

Essentially, every subprogram can access every variable. This means that the programming model itself does **not** support consistent access to some related parts of the data.



```
REPORT ....  
*-----  
TYPES: ...  
  
DATA: ...  
  
...  
  
PERFORM form1 ...  
  
CALL FUNCTION 'FB1'  
...  
CALL FUNCTION 'FB2'  
...  
  
*-----  
FORM f1 ...  
...  
ENDFORM.
```

- **Type definitions**
- **Data declarations**
- **Main program**
 - **Calling subroutines**
 - **Calling function modules**
- **Definition of subroutines**

Figure 3: Typical Procedural ABAP Program

A typical procedural ABAP program consists of type definitions and data declarations, which describe the structure of the data the program uses when it is executed. Modularization units (for example, subroutines or function modules) can be encapsulated. However, on the main program level, there is no special protection for the global data objects. Any variables can be accessed by any means.

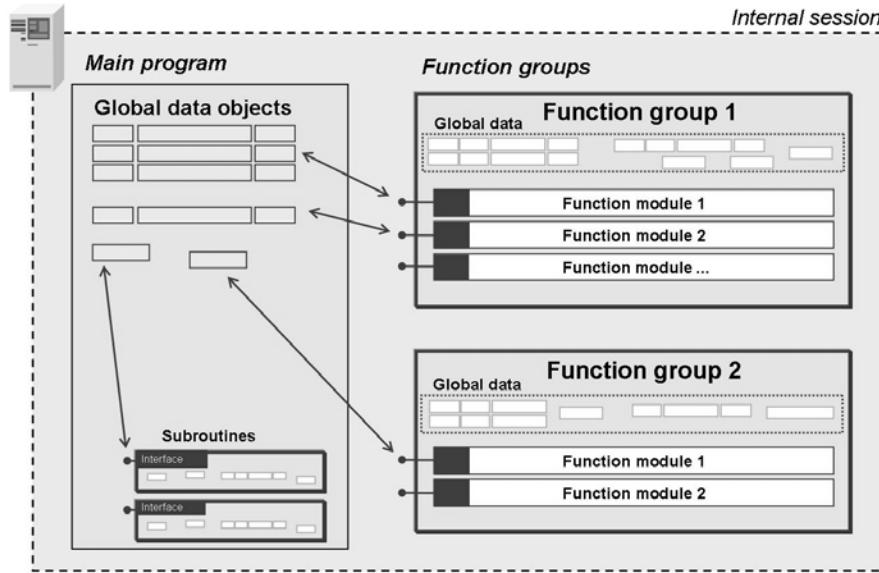


Figure 4: Encapsulating Data Using Function Groups

Every time a function module is called in a main program, its function group is loaded into the internal session. The function group remains active until the main program is finished. The main program and the function groups that were called in it are all stored in separate memory areas. Even if their data objects have the same names, they are **not** shared.

You can only call the function modules of the function groups from the main program. In turn, the function modules can access the other components – particularly the global data – of the function groups. In other words, it is not possible to access the function group's global data directly from the main program.

Encapsulation also incorporates the idea that the implementation of a service can be hidden from the system's other components, so that these cannot and do not have to make assumptions about the internal status of the modularization unit. This way, the design of these other components is not dependent on a specific implementation of the other modularization units.

Therefore, a function group is a unit of data and functions that manages this data. Encapsulated access to data and services – one of the many concepts of the object-oriented programming model – can therefore be supported in the procedural part of ABAP Objects. This meant that **BAPIs** could be implemented as function modules and **Business Objects** could be implemented as function groups.



```
FUNCTION-POOL s_vehicle.

* speed is a global variable
* used in the function-pool
DATA: speed TYPE i.

...
FUNCTION inc_speed.
...
    ADD imp_speed TO speed.
ENDFUNCTION.

FUNCTION dec_speed.
...
    SUBTRACT imp_speed FROM speed.
ENDFUNCTION.

FUNCTION get_speed.
    exp_speed = speed.
ENDFUNCTION.

...
```

Function group with functions for controlling the speed of a car

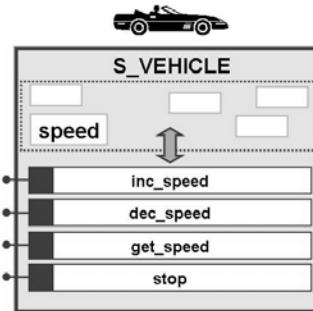


Figure 5: Example of a Function Group

The fictional function group S_VEHICLE provides a user or client with the services INC_SPEED, DEC_SPEED, and GET_SPEED. These services are the function group's interface and access the global data object SPEED, which belongs to the whole function group.



```
REPORT .....

TYPES: .....

DATA: wa_car TYPE .....

* no direct access to speed
* use functions of pool

CALL FUNCTION 'INC_SPEED'
...
CALL FUNCTION 'GET_SPEED'
...
CALL FUNCTION 'STOP'
...
```

Main program that uses function modules of this function group

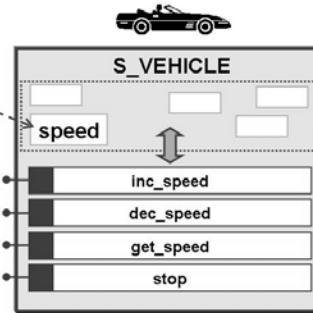


Figure 6: Example of Using the Function Group

The main program **cannot** access the function group's data object SPEED directly.

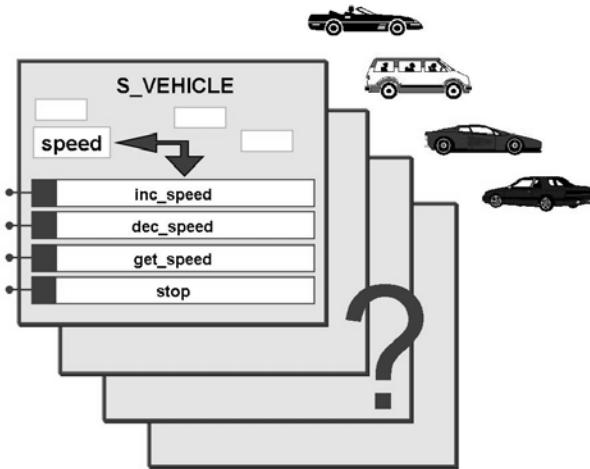


Figure 7: Several Instances of One Function Group?

If the main program is to work with several vehicles, this is not possible without extra programming and administrative effort. Most importantly, one specific vehicle could no longer be represented by a whole function group.

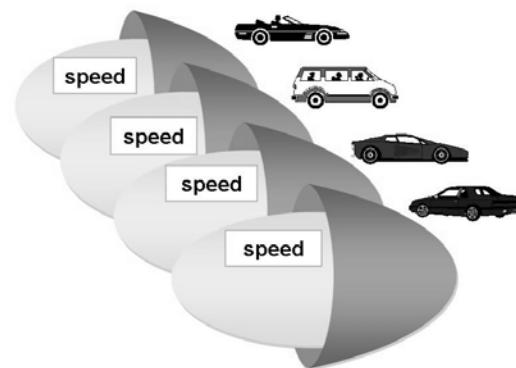


Figure 8: Multiple Instantiation in Object-Oriented Programming

The possibility of creating several runtime instances in a capsule for each program context is one of the key characteristics of object-oriented programming.

In this example, four vehicles were created, all of which have different characteristic instances. However, they all share the same data structure, the same range of functions, and the ability to protect their data against access from outside.

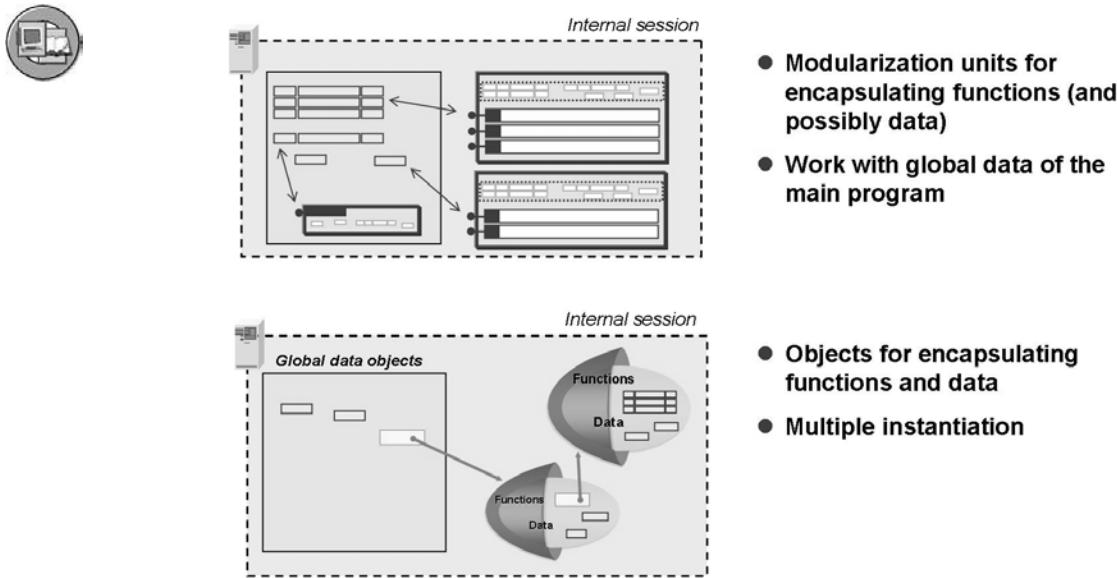


Figure 9: ABAP Main Memory and Encapsulation

Like the function groups, the objects are also stored in the same internal session as the program that is being used. Also, all data areas are separated from each other and are therefore protected.

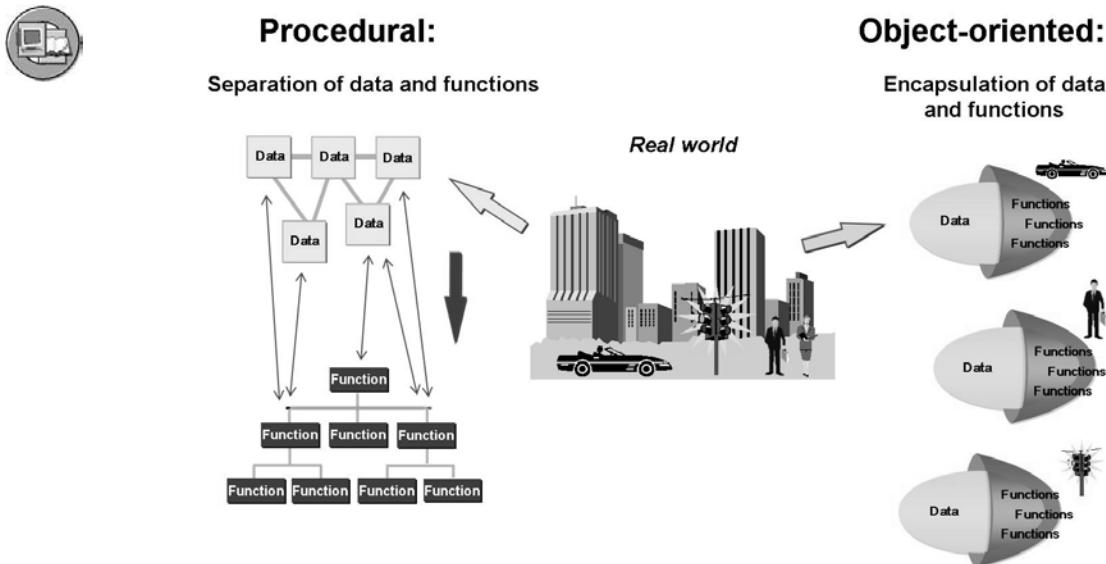


Figure 10: Data Management in Procedural and Object-Oriented Models – Summary

Unlike in procedural programming, the use of multiple instantiation in object-oriented programming allows you to create a direct abstraction of a real object. The established concept of encapsulation was systematically extended in the process.

The Object-Oriented Programming Model of ABAP Objects

The object-oriented concepts of ABAP Objects are essentially the same as those of other modern object-oriented languages like C++ or Java. A small number of concepts that did not prove to be successful in these other languages were not included in ABAP Objects. On the other hand, ABAP Objects also has helpful language elements that C++ and Java do not offer.

Some specific features of ABAP Objects only exist because of the guaranteed upward compatibility of older ABAP language elements.

Major differences in comparison to other object-oriented languages are in the development environment. You can use the entire range of functions of the *ABAP Workbench* with ABAP Objects.



```
REPORT .....

DATA: counter TYPE i,
      wa TYPE knal.
      ...

CLASS lcl_car DEFINITION.
  ...
ENDCLASS.

----- main program -----
counter = counter + 1.

CREATE OBJECT ...

MOVE wa TO ...
```

- **ABAP Objects statements can be used in procedural ABAP programs**
- **Objects (classes) contain procedural ABAP statements**

In the object-oriented context:

- **Only object-oriented concepts that have proved useful**
- **Increased use of type checks**
- **Obsolete statements are prohibited**

Figure 11: ABAP Objects as a Compatible Extension of ABAP

ABAP Objects is not a new language, but has been designed as a systematic extension of ABAP. All of the extensions, including the old procedural parts, are upwardly compatible.

Type checks in the object-oriented contexts of ABAP Objects are stricter than those in the procedural contexts.

In developing ABAP Objects, the ABAP language was cleaned up, in particular in the object-oriented contexts. This means that obsolete statements lead to syntax errors.

However, it is also advisable to avoid obsolete statements in the purely procedural environment, as this creates source texts that are safer and more flexible. Nevertheless, as the language is upwardly compatible, it is not possible to prevent the use of such statements entirely.

For a list of obsolete language elements, refer to the *ABAP keyword documentation*. Each of the obsolete statements is also specifically noted as forbidden in the object-oriented context.



- Objects behave towards each other like client/server systems.
- Every object can essentially fulfill either role.
- Distribution of responsibilities to avoid redundancy through delegation

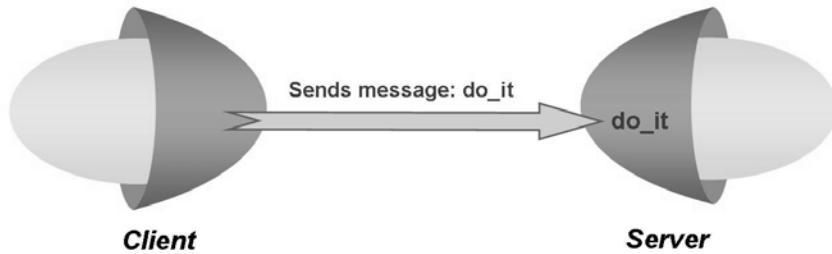


Figure 12: Client/Server Relationships Between Objects

Objects behave like client/server systems: When one object sends a message to another object, telling it to behave in a certain way, the first object can be seen as a client and the other as a server.

To be able to separate requests and deliveries of services, the following must be true:

- The client object must adhere to the server object's protocol.
- The protocol must be clearly described so that a potential client can follow it without any problems.

In principle, objects can perform both roles simultaneously: They can offer services to other objects while requesting services at the same time.

In object-oriented programming, the services are distributed amongst the objects in such a way as to avoid redundancies and so that each object offers exactly those services that are within its area of responsibility. If an object needs any other services, it requests these from other objects. This is known as the **principle of delegation** of tasks.

Example:

The common task “data retrieval and output” should be distributed over at least two objects. One is responsible for data retrieval and one for output. As long as the data retrieval object does not change its protocol, its internal implementation can be altered, without any changes to the output object becoming necessary. Alternatively, the data retrieval object could even be replaced by a different object, providing that the new object uses the same protocol. These exchanges can also take place at runtime.



- **Inheritance**
- **Polymorphism for support of generic programming**
- **Event controlling**

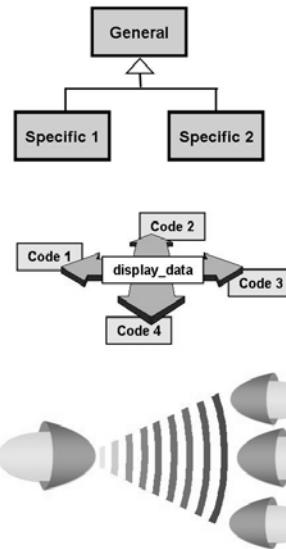


Figure 13: Additional Concepts of the Object-Oriented Programming Model

inheritance

Inheritance defines the implementation relationships between classes, so that one class (the subclass) adopts the structure and behavior of another class (superclass), possibly also adapting or extending it.

polymorphism

In object-orientation, polymorphism is when instances of different classes respond differently to the same messages.

event control

Instead of sending messages directly to specific objects, objects can also trigger events. Events can be triggered if it is not yet known at the time of development if objects will react, and if so, how they will react.

To summarize, the object-oriented programming model of ABAP Objects has the following key characteristics:

Characteristics of the Object-Oriented Programming Model



- Objects are a direct abstraction of the real world
- Objects are units made up of data and the functions belonging to that data
- Processes can be implemented realistically

The model has the following advantages:

Advantages of the Object-Oriented Programming Model over the Procedural Programming Model



- Improved software structure and consistency in the development process
- Reduced maintenance effort and less susceptibility to errors
- Better integration of the customer/user into the analysis, design, and maintenance process
- The options for extending the software are simpler and more secure

A standardized language is used in the various phases of software development (analysis, specification, design, and implementation). Communication is much easier when changing between phases.

In object-oriented programming, analysis and design decisions have an even greater effect on the implementation than they do in procedural programming.

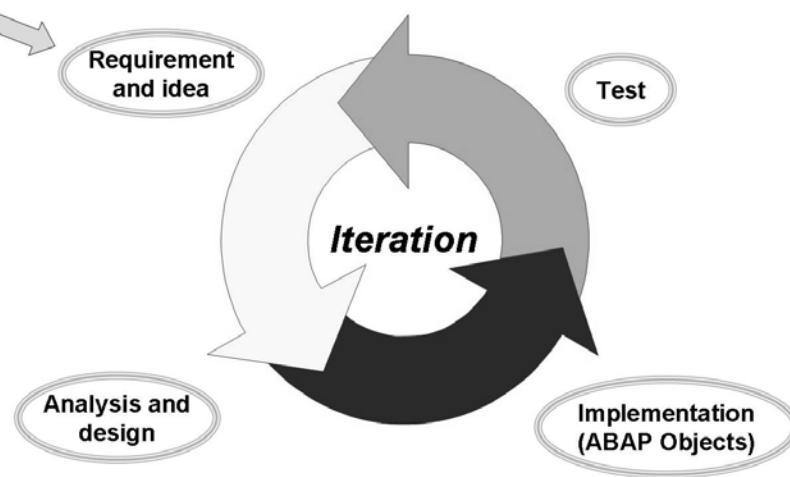


Figure 14: The Software Development Process

Therefore, you should already structure and formally standardize the analysis and design phase. You can use modeling languages to do this.



Lesson Summary

You should now be able to:

- Explain the differences between procedural and object-oriented programming models
- List the advantages of the object-oriented programming model

Lesson: Analysis and Design with UML

Lesson Overview

This lesson will help you work out how to develop an object-oriented solution to a business application problem, from classifying your objects through to defining the relationships between them.

We will be using parts of the UML modeling standard as a visual aid.



Lesson Objectives

After completing this lesson, you will be able to:

- Name the most important diagram types in UML
- Create simple class diagrams
- Create simple object diagrams
- Describe sequence diagrams

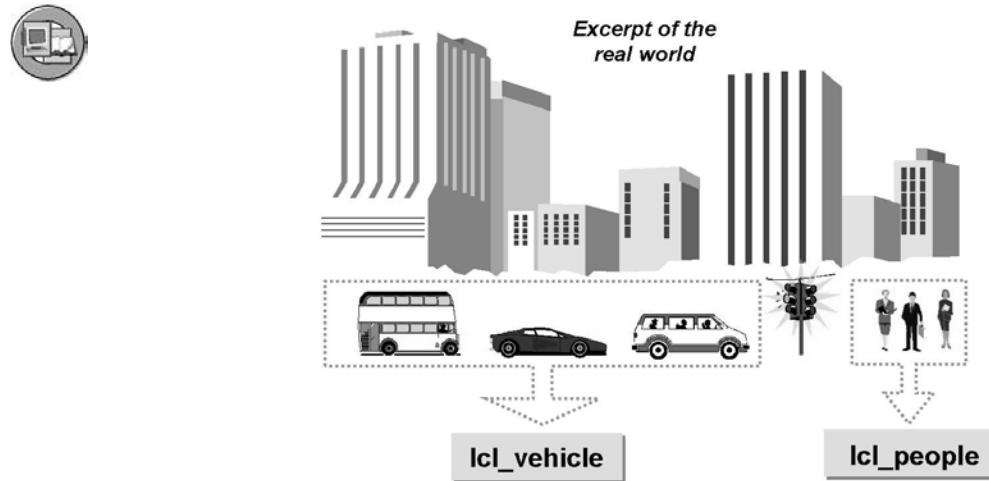
Business Example

A business application requirement is to be modeled before it is implemented.

Classification

With object-oriented programming, the real world is viewed as a collection of objects – for example, different airplanes, cars, and people. Some of these objects are very similar. In other words, they can be described using the same characteristics and they exhibit the same behavior.

All characteristics and behavior of these similar objects are now grouped into one central class. This class is used to describe every object that derives from it. A class is therefore a description of a quantity of objects that are typified by the same characteristics and the same behavior.



Class creation / modeling

Figure 15: Classification of Objects

For example, the vehicle “make x, ... series n”, is an object of class “car”. This object is therefore a concrete instance of its class.

Consequently, an object has an identity, a status (number of characteristic instances), and a behavior. Do not confuse the concepts of identity and status. Identity is an attribute that distinguishes each object from all other objects of its class. Two different objects can have identical attribute values and still not be identical.

Example: Two coffee cups have the same height and diameter, have the same handle, and are both white. Although their statuses are therefore completely identical, they are clearly two different coffee cups.

Literature on the subject of object orientation often speaks of **instances**. This simply means an object.

→ **Note:** In the literal sense of the word instance, the meaning is slightly more specific. It means a concrete – that is, uniquely identifiable – instance of a class.

In the following pages, we will make a distinction between instance and object.

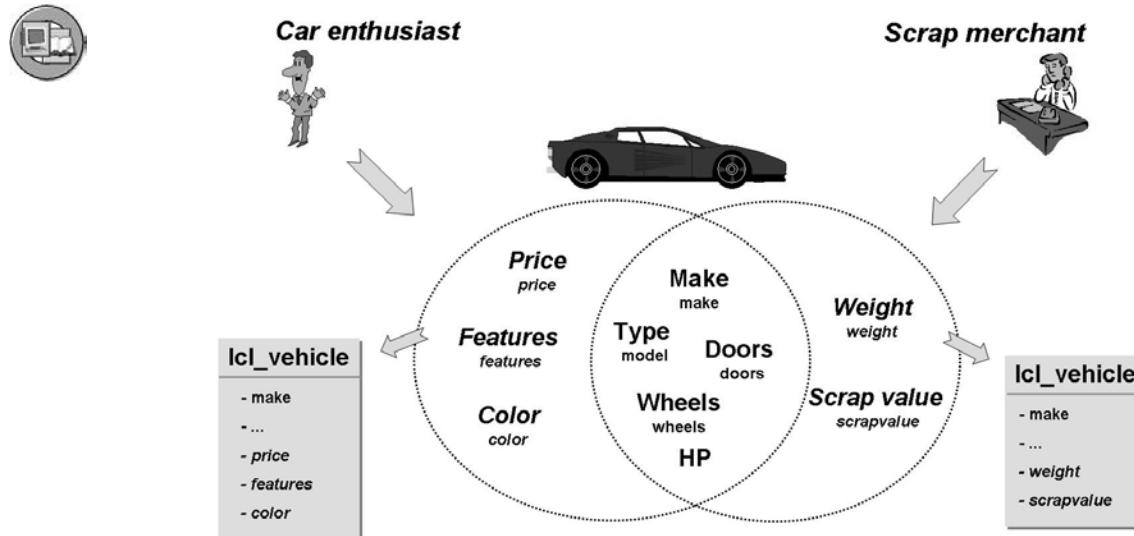


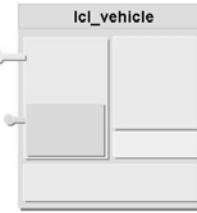
Figure 16: Classes as Abstraction Forms

In a software context, abstractions are simplified representations of complex relationships in the real world. A real, existing object is abstracted to the dimensions that are relevant for simulating the required section of the real world.

This example concerns vehicles. Software for a vehicle enthusiast and software for a scrap merchant contain different abstractions (classes) for these objects. So, depending on the type of abstraction, a class can contain very different aspects of the objects.

**Class**

- General description of objects
- Specifies status data (attributes) and behavior (methods)

Represented by*or***Object**

- Representation of part of the real world
- Concrete form/specimen-instance of a class

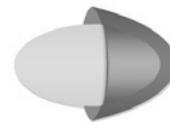
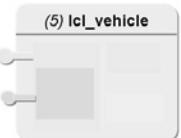
*or*

Figure 17: Comparison of Classes and Objects

A sure understanding of the relationship between classes and objects as summarized here again is an absolute prerequisite for proceeding with the following lessons successfully.

Modeling in UML

Unified Modeling Language (UML) is a globally standardized modeling language. It is used for the specification, construction, visualization, and documentation of models for software systems and enables uniform communication between users.

UML is an industry standard and has been developed by the **Object Management Group** (OMG) since September 1997. SAP uses UML as the company-wide standard for object-oriented modeling.

You can find the UML specifications on the OMG homepage at: <http://www.omg.org>

UML describes a number of different diagram types in order to represent different views of a system. The following three diagram types are of particular significance in this context:



Class diagrams

Show the classes and the relationships between them, that is, a **static** view of a model.

Behavior diagrams

Pay particular attention to the **sequence** in which the **objects** relate to each other.

Component diagrams

Show the organization and dependencies of components.

Later, we will need to find concrete methods for realizing the first two aspects in the list in the programming language.

The third aspect is realized in the Repository object package. In other words:
Packages can be used to realize the structure of software components within SAP Web AS according to the structure's specification in a component diagram.

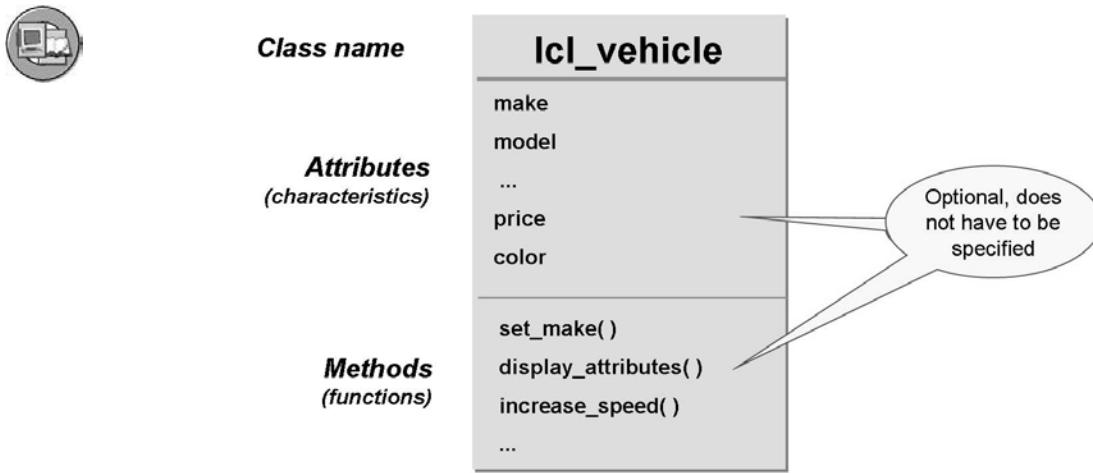


Figure 18: Representation of a Class

A class is represented by a rectangle in UML notation. First, the class's name is given, then its attributes, and finally its methods. However, you also have the option of omitting the attribute part and/or the method part.

Attributes describe the data that can be stored in the objects of a class. They also determine the status of an object.

Methods describe the functions that an object can perform. They therefore determine the object's behavior.

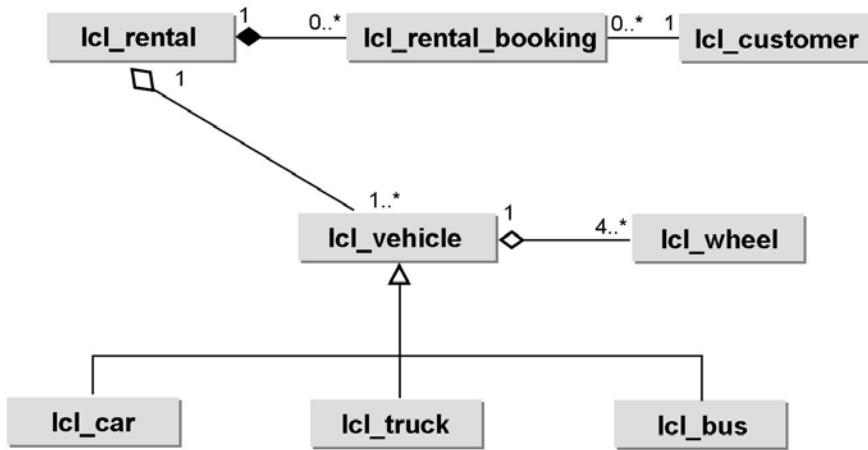


Figure 19: Example of a Class Diagram

A class diagram describes all **static** relationships between the classes. There are two basic forms of static relationships:

Association

In this example: A customer books a car at a rental car company.

Generalization/Specialization

In this example: a car, a bus, and a truck are all vehicles.

- **Note:** As mentioned previously, classes can also be shown in class diagrams with their attributes and methods. Here, these have been left out to improve clarity.

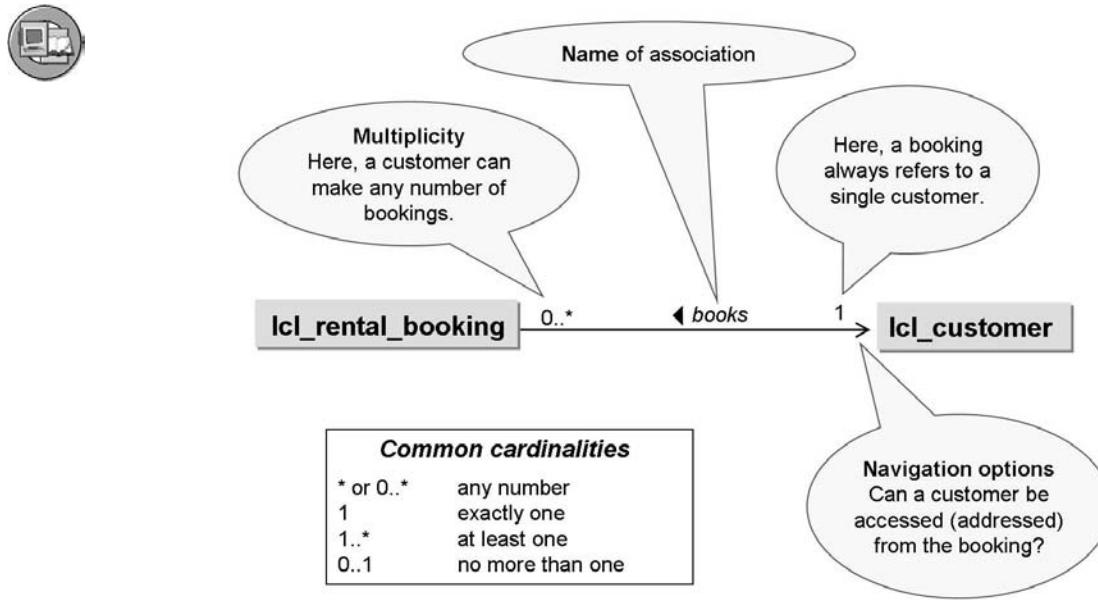


Figure 20: association

An association describes a semantic relationship between classes. The specific relationship between objects in these classes is known as an object link. Object links are therefore the instances of associations.

An association is usually a relationship between different classes (binary association). However, an association can also be recursive; in this case, the class would have a relationship with itself. In most cases, recursive associations are used to link two different objects in one class. The points below assume that the associations are binary.

Each association has two roles, one for each direction of the association. Each role can be described with an association name. Each role has a cardinality that shows how many instances can participate in this relationship. The multiplicity is the number of participating objects in one class that have a relationship to an object in the other class. Like all other elements of the model, cardinalities are dependent on the concrete situation that is being modeled.

In this example, you could also require a cardinality of “at least one”, to indicate that only a person who actually makes a booking becomes a customer of the rental car company. On the other hand, the cardinality “any number” would allow for a more general definition of a customer.

- An association is represented by a line between the class symbols.
- The cardinality (also referred to as multiplicity) of the relationship can be specified at each end of the line.
- Arrows can also be used to indicate the navigation options, that is, the accessibility of the association partner.
- This association name is written in italics above the line and may contain an arrow to show the read direction.
- If roles are defined for both partners, role names can be entered at the end of the lines.

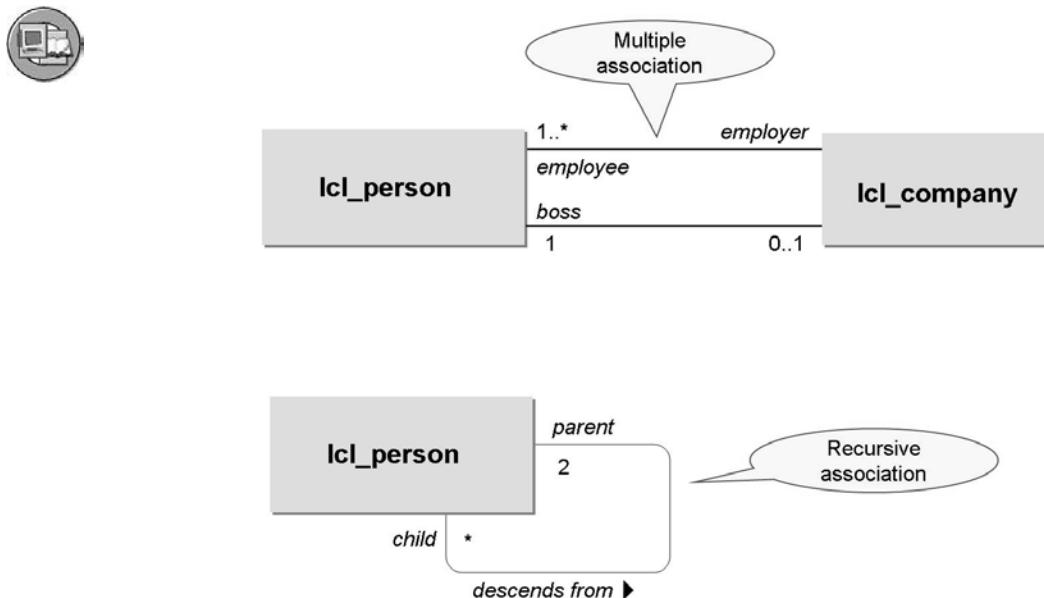
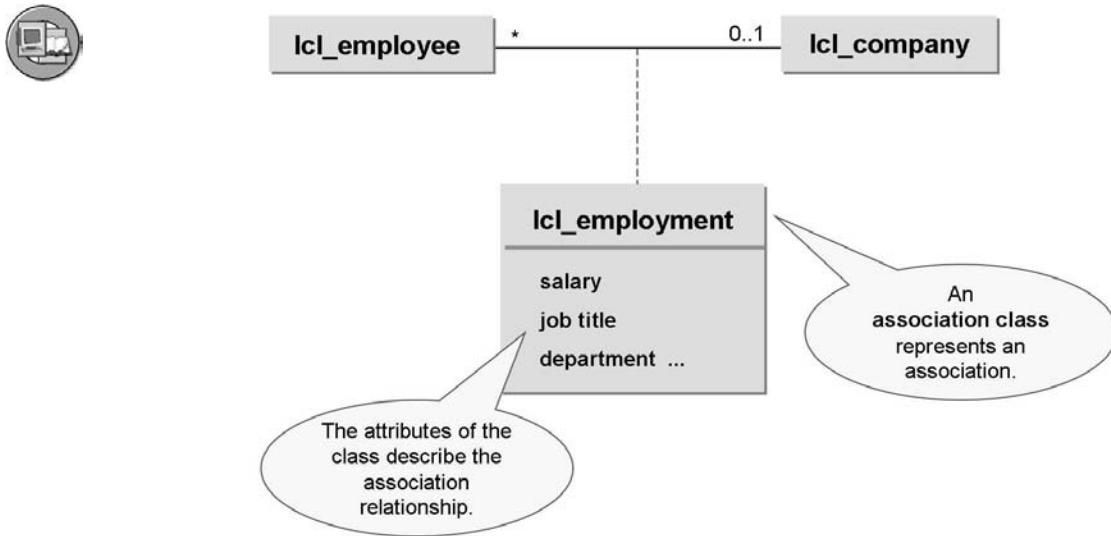


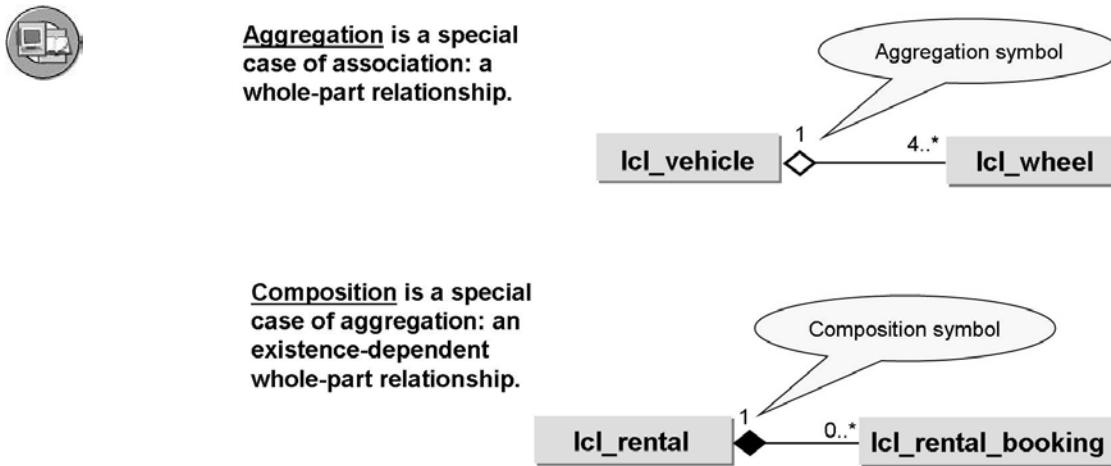
Figure 21: Association with Roles - Examples

In the example of **multiple association** shown here, role names are used at the end of the association lines to describe in greater detail the relationships between the classes involved. A person could appear in the role of employee or of company director here.

In the **recursive association** shown here, the two roles of “child” and “parent” are similarly defined using role names. Two instances of the LCL_PERSON class thus have a relationship with each other and represent two roles.

**Figure 22: Association Classes**

If association is used to link two classes, this relationship can be better represented by a special class. The various characteristics of the relationship are described using the attributes of the association class. A dotted line connects this additional class to the association line.

**Figure 23: Aggregation and Composition**

Aggregation and composition are specializations of association. They show that an object consists of other objects or contains other objects (composition part). The relationship can be described with the words “consists of” or “is a part of”. For example, a car consists of wheels, among other things.

Aggregation and composition are displayed as a line between two classes that is labeled with a small rhombus. The rhombus indicates the **aggregate**, that is, the composition. Otherwise the notation conventions are the same as for associations.

Composition is a **specialization of aggregation**. Composition means that the contained object **cannot exist without the aggregate** (for example, a car reservation cannot exist without the car rental). Therefore, the cardinality of the aggregate can only be exactly one. The lifetime of the individual parts is linked to the lifetime of the aggregate: Parts are created either with or after the aggregate, and they are destroyed either with or before the aggregate.

In UML notation, composition is denoted by a filled-in rhombus.

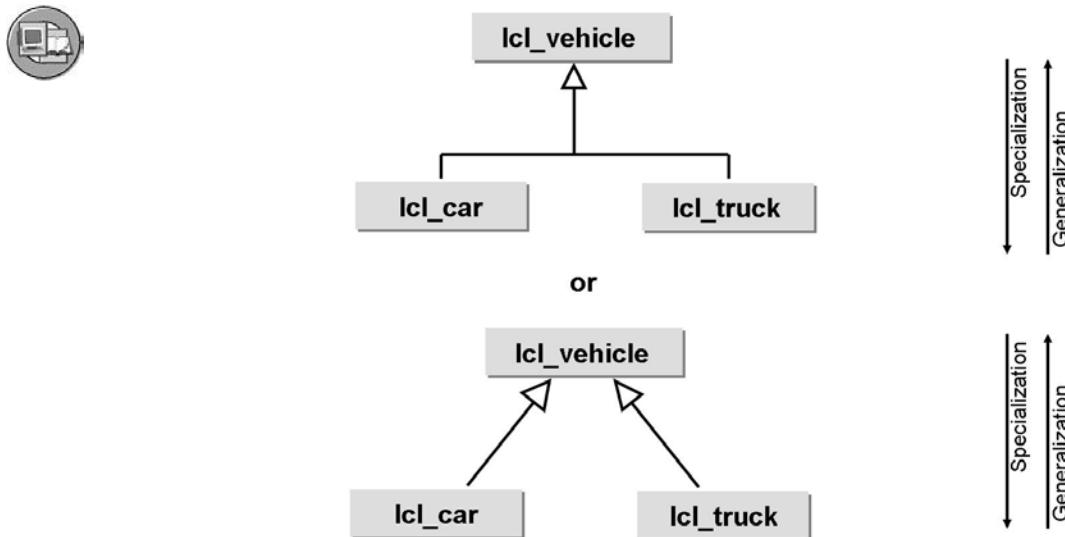
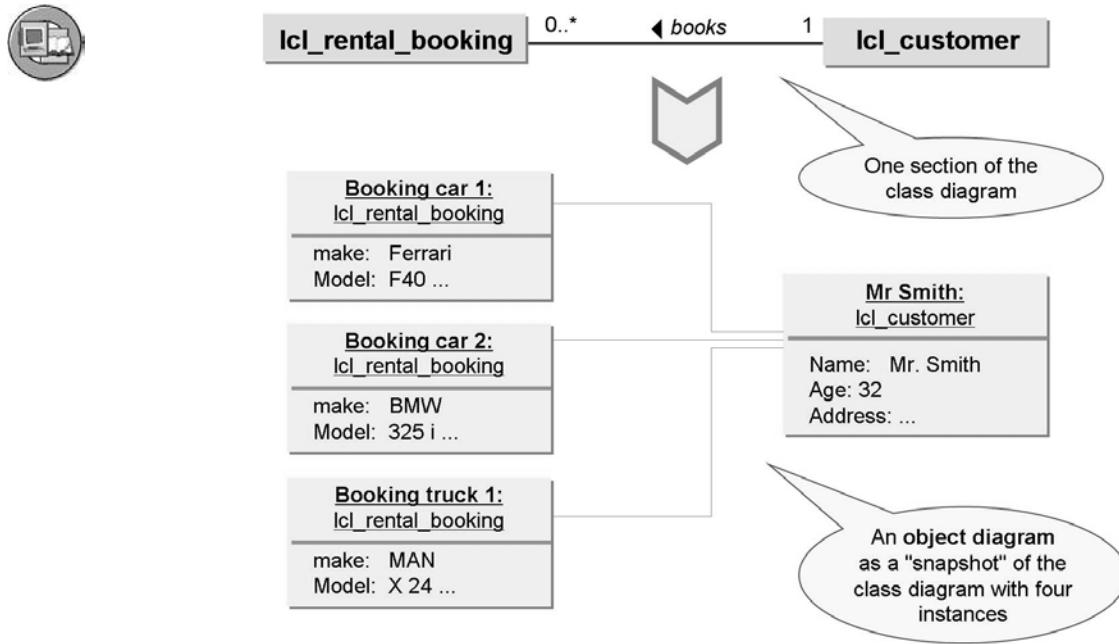


Figure 24: Generalization and Specialization

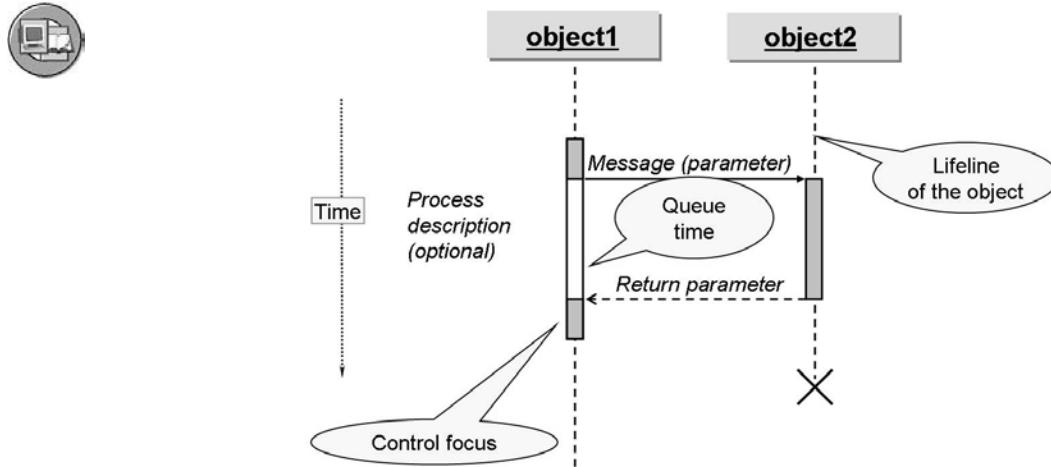
Generalization and specialization relationships are always bidirectional.
Generalization can be described with the words “is a special.”

Generalization/specialization relationships are indicated by a triangular arrow. This arrow always points to the more general class. The level of generalization increases in the direction of the arrow. Trees can be built up using several of these relationships.

**Figure 25: object diagram**

An **object diagram** is a “snapshot” taken during program execution, which describes the instances of the classes and the relationships between them.

It is not a new type of diagram. Rather, it is a variant of the class diagram and is only useful for representing a complex class diagram.

**Figure 26: sequence diagram**

Sequence diagrams are used to display certain **processes** or **situations**. Sequence diagrams focus on the time sequence of the behavior:

- Creating and deleting objects
- Exchanging messages between objects

In UML notation, the object lifeline is represented by dotted vertical lines with a box containing the object name at the top. An 'X' marks the end of the lifeline.

The control focus is shown as a vertical rectangle on the object lifeline. The control focus shows the object's "active" period:

- An object is **active** when actions are executed
- An object is **indirectly active** if it is waiting for a subordinate procedure to end

Messages are shown as horizontal arrows between the object lines. The message is written above the arrow in the form *nachricht (parameter)*. There are various ways of representing the reply; in this example, it is shown as a dotted returning arrow.

You can also include a description of the process and add comments to the object lifeline as required.

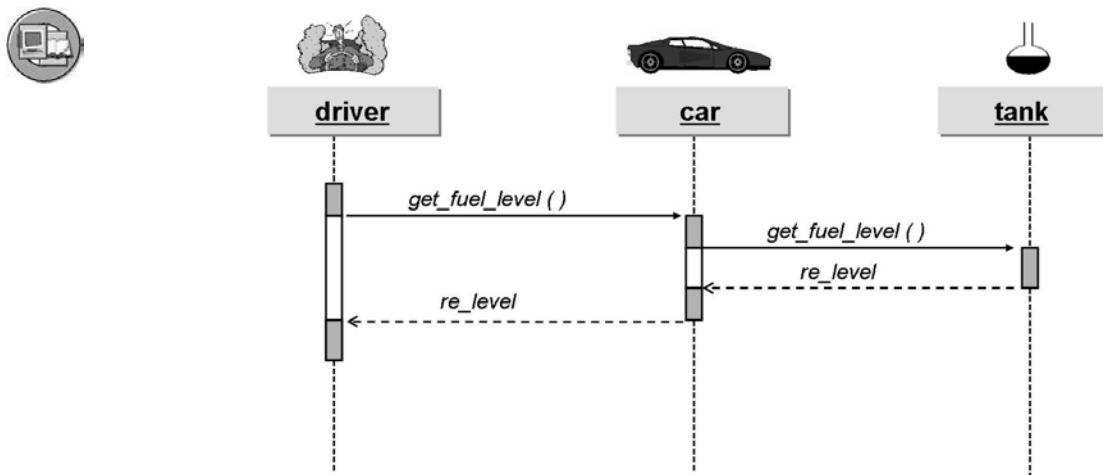


Figure 27: Delegation Principle in a Sequence Diagram

In delegation, two objects are involved in handling a request: The recipient of the request delegates the execution of the request to a delegate.

In this example, the driver (object DRIVER) sends the message GET_FUEL_LEVEL to the vehicle (object CAR). The receipt of this message causes the car to send a message to the tank (object TANK) to find out what the tank contains. In other words, the car **delegates** this task to the tank. If necessary, the car formats the information containing the current value of the tank contents before it passes it back to the driver.

Exercise 1: UML Class Diagrams

Exercise Objectives

After completing this exercise, you will be able to:

- Design simple UML class diagrams
- Model basic object classifications

Business Example

Modeling simple airplane management.

Task 1: Model a UML Class Diagram

You want to model some key classes for simple airplane management.

1. Your UML class diagram should contain the following classes:

LCL_CARRIER	for the airline companies
LCL_AIRPLANE	for airplanes (general)
LCL_PASSENGER_PLANE	for passenger planes
LCL_CARGO_PLANE	for cargo planes

2. Include some appropriate attributes and methods for each class.
3. Define relationships between your classes.
Choose suitable association types.
4. Choose suitable cardinalities.

Task 2: Optional: Object Diagrams

Decide whether the following object diagrams are correct.

1. A class diagram is shown (see the following figures).
Eight object diagrams are drawn for this class diagram.
Decide whether each object diagram is correct and check the provided box if it is correct.

Continued on next page

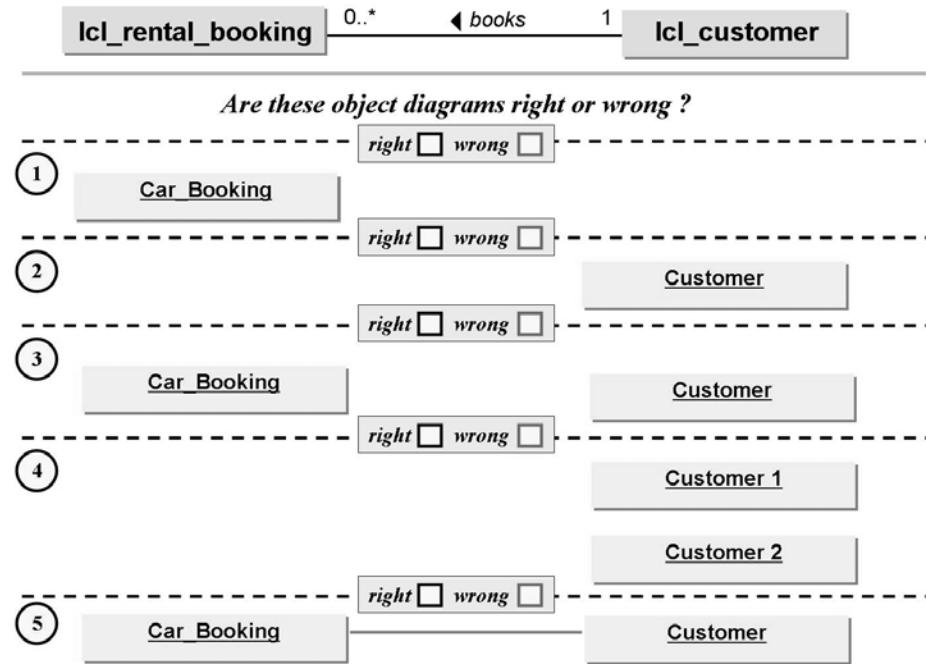


Figure 28: Possible Object Diagrams? (1)

Continued on next page

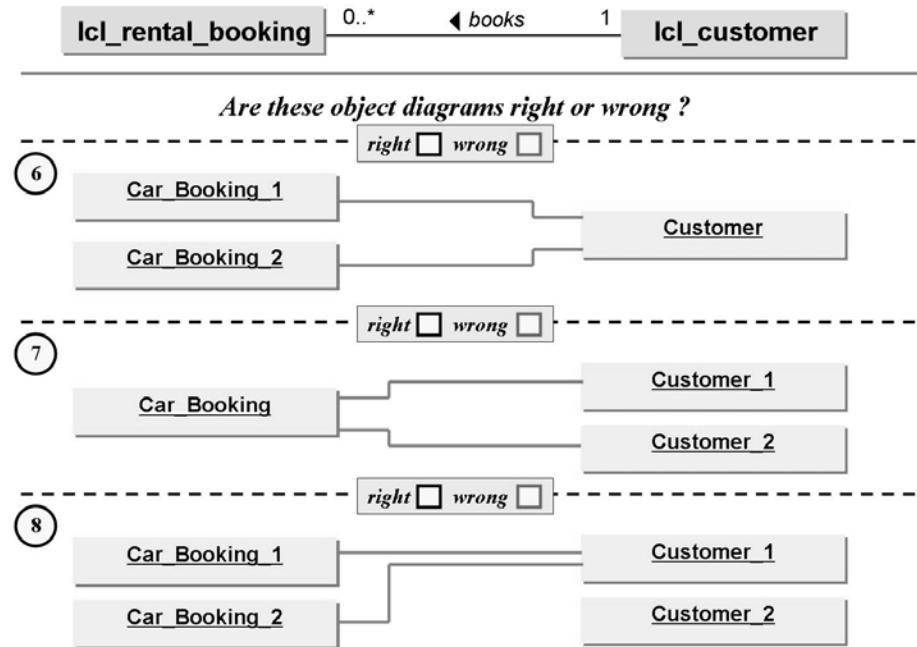


Figure 29: Possible Object Diagrams? (2)

Solution 1: UML Class Diagrams

Task 1: Model a UML Class Diagram

You want to model some key classes for simple airplane management.

1. Your UML class diagram should contain the following classes:

LCL_CARRIER	for the airline companies
LCL_AIRPLANE	for airplanes (general)
LCL_PASSENGER_PLANE	for passenger planes
LCL_CARGO_PLANE	for cargo planes

- a) Use the model solution as a guide.
2. Include some appropriate attributes and methods for each class.
 - a) The general attributes and methods for the airplanes should be contained in LCL_AIRPLANE. Continue using model solution as a guide.

3. Define relationships between your classes.

Choose suitable association types.

- a) A generalization/specialization relationship between LCL_AIRPLANE and LCL_PASSENGER_PLANE or LCL_CARGO_PLANE seems to be appropriate. An aggregation should exist between LCL_AIRPLANE and LCL_CARRIER.

Use the model solution as a guide.

Continued on next page

4. Choose suitable cardinalities.
- a) Various cardinalities can be used in this case. Use the relevant sections of the lesson and the model solution as a guide.

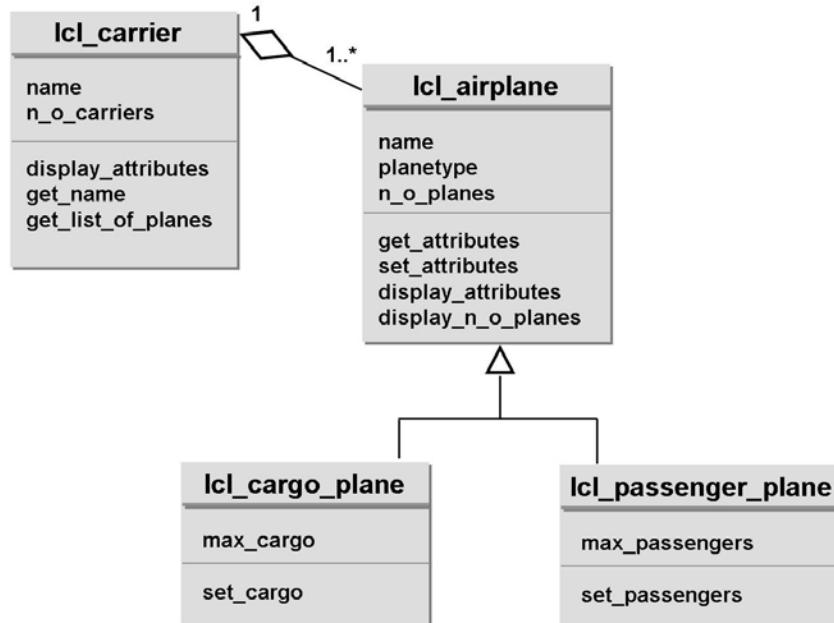


Figure 30: Class Diagram for Exercise: CARRIER/AIRPLANE

Task 2: Optional: Object Diagrams

Decide whether the following object diagrams are correct.

1. A class diagram is shown (see the following figures).

Eight object diagrams are drawn for this class diagram.

Decide whether each object diagram is correct and check the provided box if it is correct.

Continued on next page

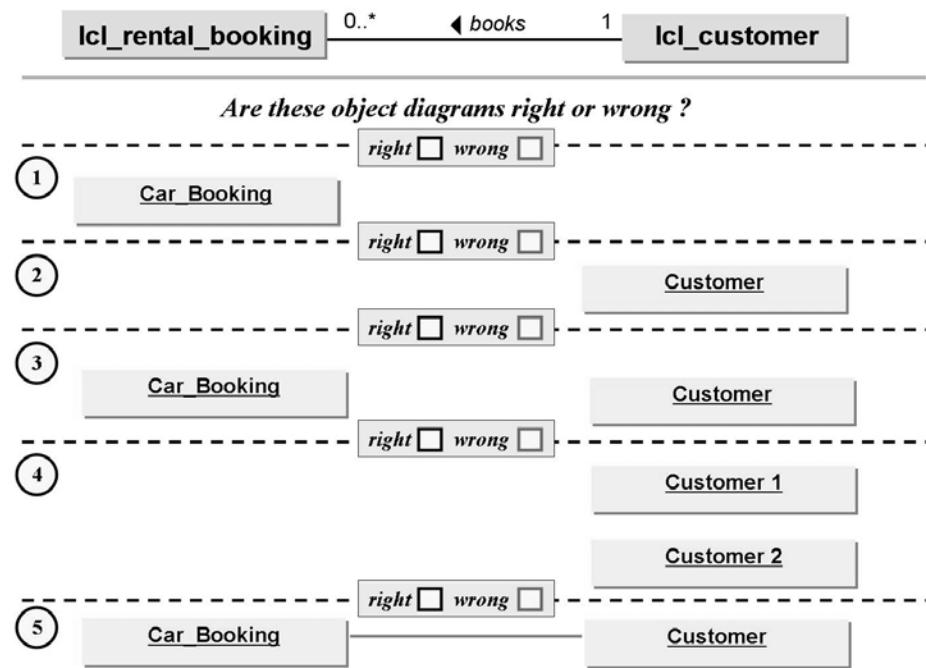
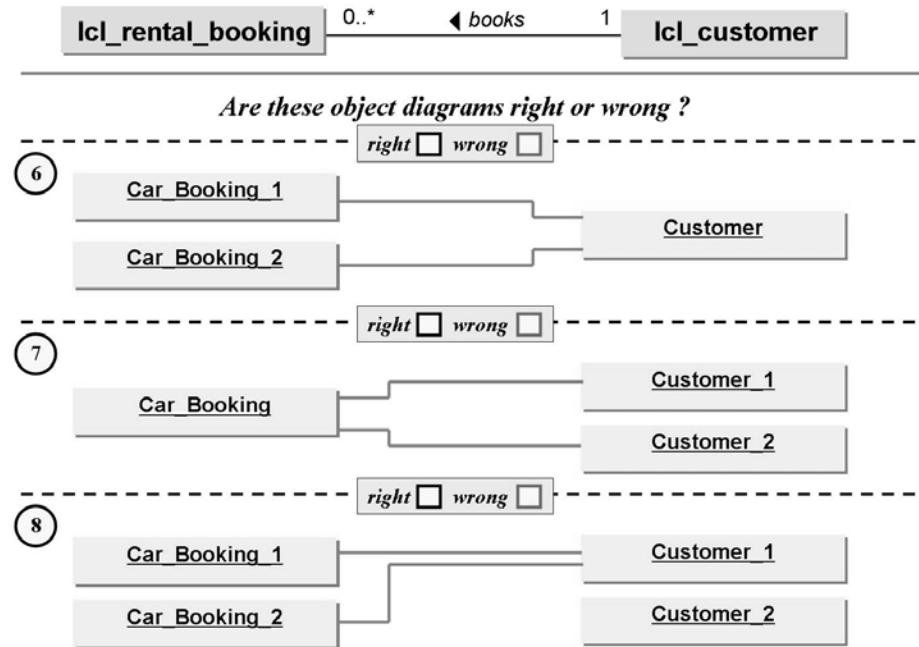


Figure 31: Possible Object Diagrams? (1)

Continued on next page



- a) The object diagrams numbered 2, 4, 5, 6 and 8 are correct.



Lesson Summary

You should now be able to:

- Name the most important diagram types in UML
- Create simple class diagrams
- Create simple object diagrams
- Describe sequence diagrams

Lesson: Fundamental Object-Oriented Syntax Elements

Lesson Overview

In the course of this lesson, you will put your models into practice. First, you need to learn the fundamental object-oriented syntax elements.

You will receive a step-by-step introduction to the definition of local classes and their basic uses, accompanied by several exercises.



Lesson Objectives

After completing this lesson, you will be able to:

- Define classes
- Generate and delete objects
- Access attributes
- Call methods

Business Example

You want to implement classes, objects, and associations of your model in ABAP Objects.

Classes, Attributes, and Methods

The concept of classes is the foundation for all object-oriented thinking. This section will explain and define the main components of a class.

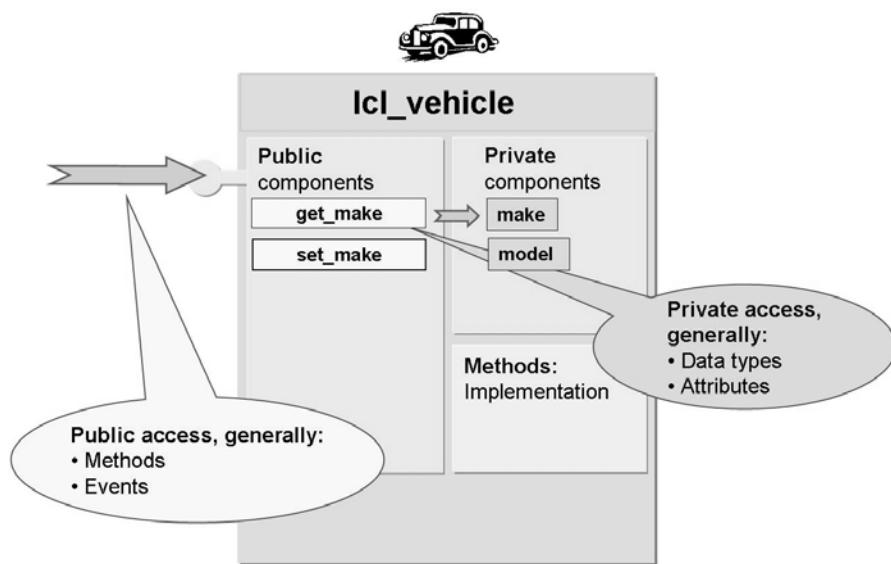


Figure 33: Example of a Class

This picture shows a vehicle as an example of a class. Using this example, we will examine the individual concepts. The node on the left shows that the public components of the class can be accessed “from outside”. On the other hand, private attributes of the class should not be accessible “from outside”.

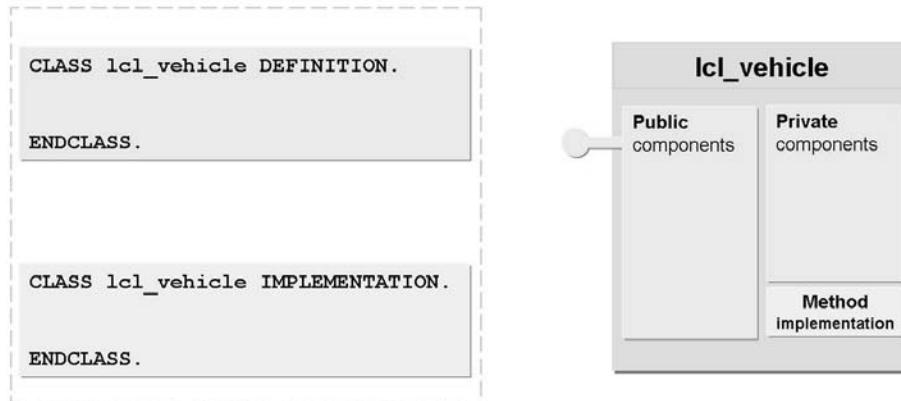


Figure 34: Defining Classes

A class is a set of objects that have the same structure and the same behavior. A class is therefore like a blueprint based on which all objects in that class are created.

All components of the class are defined in the definition part. The components are attributes, methods, events, constants, types, and implemented interfaces. Only methods are implemented in the implementation part.

The CLASS statement **cannot** be nested, that is, you cannot define a class within a class.

→ **Note:** However, you can define local auxiliary classes for global classes.

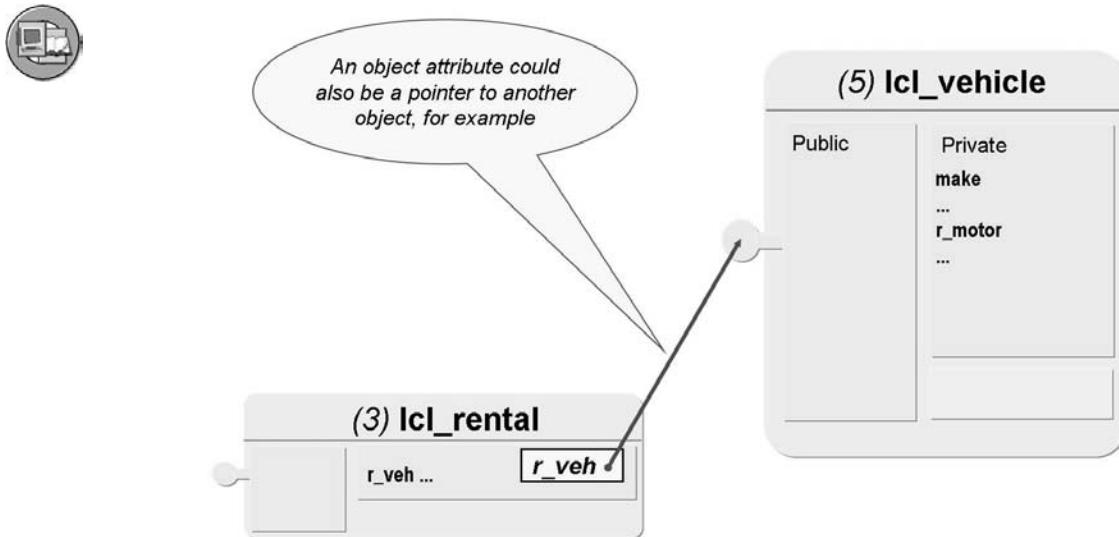


Figure 35: Example of Attributes

Attributes contain the data that can be stored in the objects of a class. Class attributes can be one of three types: elementary, structured, or table-type. They can consist of local or global data types or reference types.

Examples of attributes for the class LCL_VEHICLE are:

MAKE	Vehicle make
MODEL	Type or model
SER_NO	Serial number
COLOR	Color
MAX_SEATS	Number of seats
R_MOTOR	Reference to class LCL_MOTOR



```
CLASS class_name DEFINITION.  
  ...  
  
  TYPES: ... .  
  CONSTANTS: ... .  
  
  DATA: variable1 TYPE local_type,  
        variable2 TYPE global_type,  
        variable3 LIKE variable1,  
  
        variable4 TYPE built_in_type VALUE val,  
        variable5 TYPE ... READ-ONLY,  
  
        variable6 TYPE REF TO class_name,  
        variable7 TYPE REF TO interface_name,  
        variable8 TYPE REF TO type_name.  
  
  ENDCCLASS.
```

Reference variables
reference classes,
interfaces or types

Figure 36: Definition of Attributes, Types, and Constants

In classes, you can only use the **TYPE addition** to refer to data types. The **LIKE** reference is allowed for local data objects or SY fields (for example SY-DATE, SY-UNAME, and so on).

The **READ-ONLY** addition means that a public attribute that was declared with **DATA** can be read from outside, but can only be changed by methods in the same class. You can currently only use the **READ-ONLY** addition in the public visibility section (**PUBLIC SECTION**) of a class declaration or in an interface definition.

With **TYPE REF TO**, an attribute can be typed as any reference. This will be discussed in more detail later.

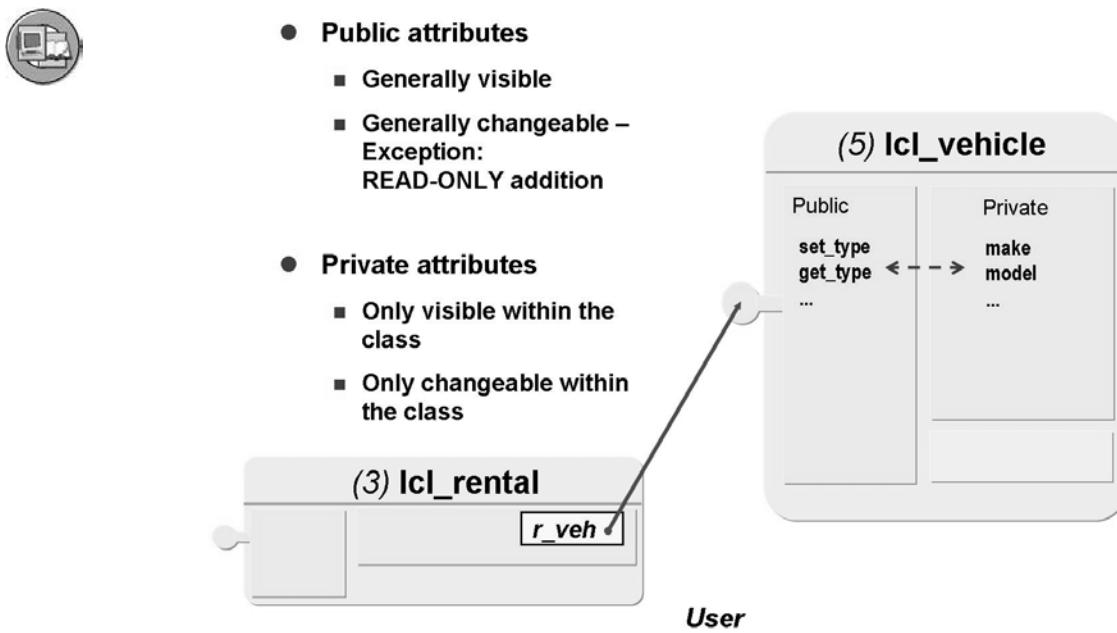


Figure 37: Visibility Sections of Attributes

You can protect attributes against access from outside by characterizing them as **private** attributes. The private components of the class cannot be addressed directly from outside. They are not visible to the outside user.

→ **Note:** The **friendship** concept is an exception to this rule.

Attributes that an outside user can access directly are **public** attributes. The public components of a class are sometimes collectively known as the **class's interface**.

Using the private visibility section is also known as **information hiding** or encapsulation. In part, this is to protect the **user** of a class: Assume that the private components of a class are changed at some point, but its interface remains the same. All external users can only access their components through the interface of the class, and so can continue to work with the class as usual after the change is made. The user does not notice the change. Only the internal implementation was changed.

Conversely, if the public components of a class were incompatibly changed, every outside user would have to take these changes into account. You should therefore use public attributes very sparingly, or avoid making subsequent incompatible changes to the public components of your classes altogether.



```
CLASS lcl_vehicle DEFINITION.  
  PUBLIC SECTION.  
    DATA: make TYPE string,  
          model TYPE string.   
  PRIVATE SECTION.  
  ...  
ENDCLASS.
```

```
CLASS lcl_vehicle DEFINITION.  
  PUBLIC SECTION.  
  ...  
  PRIVATE SECTION.  
    DATA: make TYPE string,  
          model TYPE string.   
  ENDCLASS.
```

Figure 38: Accessing Private Attributes

Define private attributes in the PRIVATE SECTION of a class. Define public attributes in the PUBLIC SECTION.

It is syntactically impossible to access private methods directly from outside. However, this is possible using public methods that output or change the attributes. The marginally higher runtime requirement (method calls in comparison with direct value assignment) is taken into account to satisfy the encapsulation concept:

The signature of the public method clearly establishes **which values** must or can be transferred, and **what types are to be assigned** to them. This relieves the outside user of all responsibility. The method itself is only responsible for ensuring that all private attributes are dealt with consistently.

For this example, imagine that the MAKE and MODEL attributes are public. The risk would be too large, since a user might forget to supply one of the two attributes or specify two inconsistent attributes. Instead, you could use a public method SET_TYPE to ensure that values are specified for both attributes. A strict syntax check governs method calls to check that all obligatory parameters are transferred. It would even be possible for the method itself to perform a consistency check (to see if a certain vehicle make produces the chosen model), and to raise an exception if an error occurs.

To save runtime, individual attributes are sometimes defined in the public visibility section, but they must then be given the addition READ-ONLY.



- **Instance attributes**
 - Exist per instance
 - Defined with DATA

```
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION.
...
PRIVATE SECTION.
DATA: make TYPE string,
...

CLASS-DATA n_o_vehicles TYPE i.
ENDCLASS.
```

- **Static attributes**
 - Exist per class
 - Definition with CLASS-DATA

Figure 39: Comparison of Instance Attributes with Static Attributes

There are two kinds of attributes:

Instance attributes

Instance attributes are attributes that exist once per object, that is, once **per runtime instance** of the class. They are defined with the syntax element DATA.

Static attributes

Static attributes exist once **for each class** and are visible for all runtime instances in that class. They are defined with the syntax element CLASS-DATA.

Static attributes usually contain information that applies to **all instances**, such as:

- Types and constants
- Central application data buffers
- Administrative information, such as the instance counter

Technical literature often refers to static attributes as **class attributes** (compare to the CLASS-DATA syntax element). In ABAP Objects, as in C++ and Java, the official term is **static attribute**.

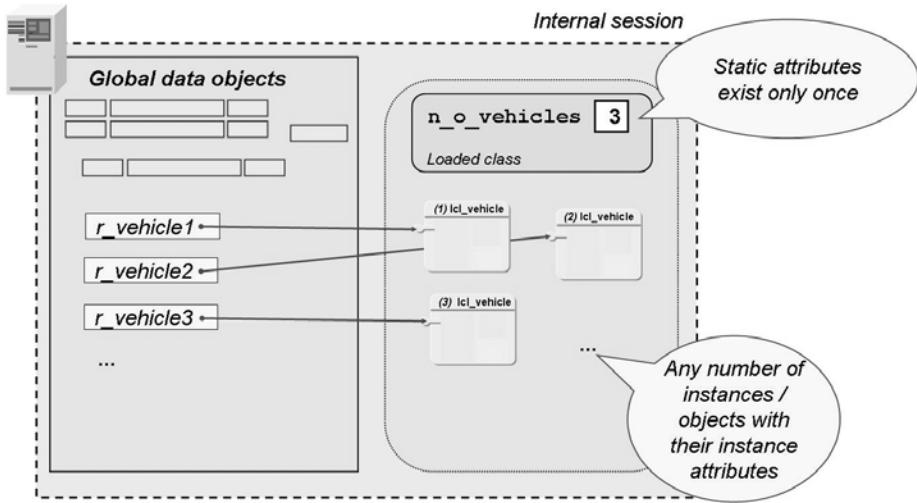


Figure 40: Instance Attributes and Static Attributes in the Program Context

The graphic shows an example of how the static attribute N_O_VEHICLES is related to the other program elements in the memory: It exists only once (in the loaded class) regardless of the number of instances of LCL_VEHICLE. Therefore, you can say that instances share their common attributes.



Caution: Here, an integer data object is defined in order to count the instances. It is **not possible** to find out the number of created instances from the system.

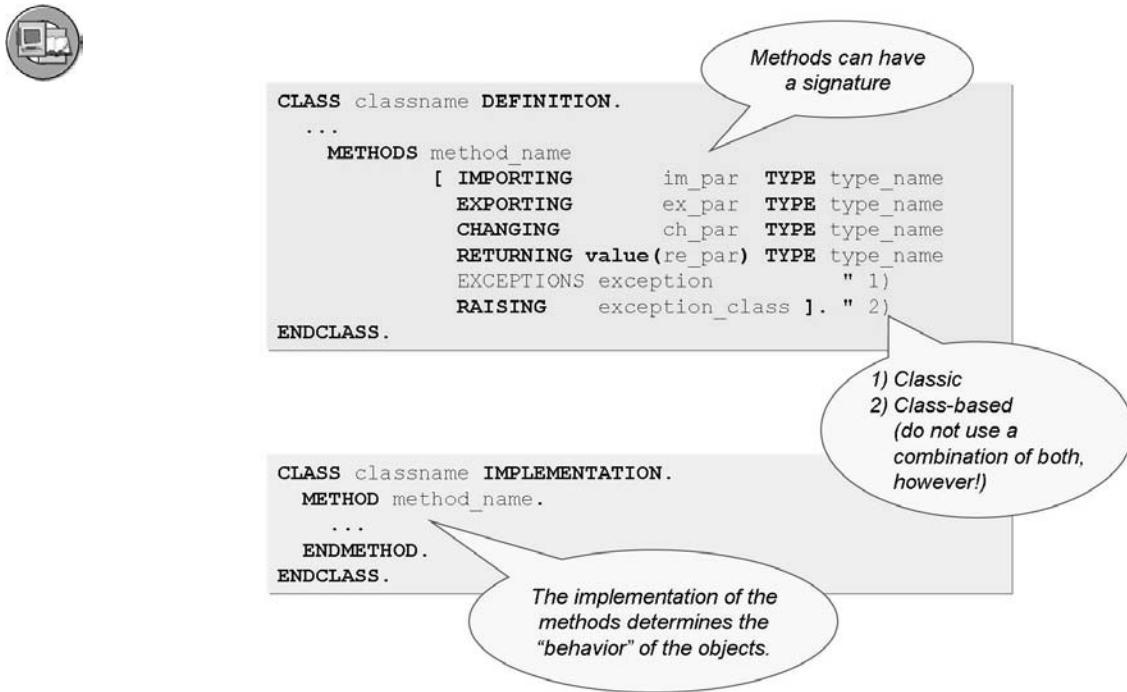


Figure 41: Syntax for Methods

Methods are internal procedures in classes that determine the behavior of the objects. They can access all attributes in their class and can therefore change the state of other elements.

Methods have a **signature** (interface parameters and exceptions) that enables them to receive values when they are called and pass values back to the calling program. Methods can have any number of IMPORTING, EXPORTING, and CHANGING parameters. All parameters can be passed by value or reference.

One method return value can be defined using the RETURN parameter. It must always be transferred as a value. In this case, you cannot then define the EXPORTING and CHANGING parameters. You can also use the RETURNING parameter to define **functional methods**. This will be discussed in more detail later.

All input parameters (IMPORTING and CHANGING parameters) can be defined as optional parameters in the declaration using the OPTIONAL or DEFAULT additions. These parameters then do not necessarily have to be transferred when the object is called. If you use the OPTIONAL addition, the parameter remains initialized according to type, whereas the DEFAULT addition allows you to enter a start value.

Like function modules, methods also support the return value SY-SUBRC, but only if the signature exceptions were defined using EXCEPTIONS (see point 1 on the graphic). As of SAP Web AS 6.10, the **RAISING addition** can be used in its place to propagate class-based exceptions (see point 2 on the graphic). The caller then handles these class-based exceptions without evaluating the SY-SUBRC return value.

Do not use both concepts together in one program!



- **Public methods**

Can be called from anywhere

- **Private methods**

Can only be called within the class

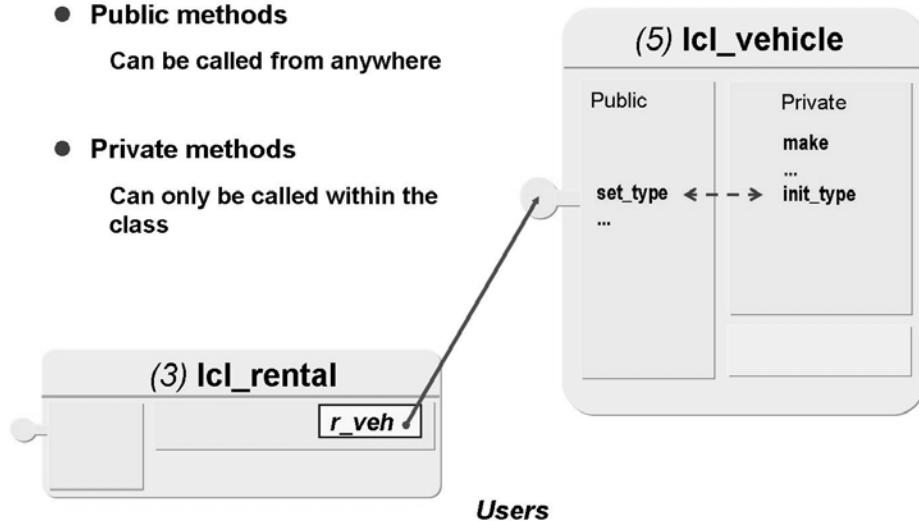


Figure 42: Visibility Sections of Methods

Methods also have to be assigned to a visibility section. This determines whether the methods are called from outside the class or only from within the class. Thus, private methods only serve the purpose of internal modularization.



```
CLASS lcl_vehicle DEFINITION.  
  PUBLIC SECTION.  
    METHODS set_type IMPORTING im_make TYPE string  
          im_model TYPE string.  
  PRIVATE SECTION.  
    METHODS init_type.  
    DATA: make TYPE string, model TYPE string.  
  ENDCLASS.
```

```
CLASS lcl_vehicle IMPLEMENTATION.  
  Method init_type.  
    CLEAR: make, model.  
  ENDMETHOD.  
  Method set_type.  
    IF im_make IS INITIAL.  
    *      calling Method init_type ...  
    ELSE.  
      make = im_make.  
      model = im_model.  
    ENDIF.  
  ENDMETHOD.  
ENDCLASS.
```

Figure 43: Accessing Private Methods

You define private methods in the PRIVATE SECTION of a class. You define public attributes in the PUBLIC SECTION.

It is not possible to directly access private methods from outside. However, a private method can be called by a public method.

In this example, INIT_TYPE is a private method that is called by the public method SET_TYPE. The instruction to initialize the attributes could exist in other contexts as well, so the definition of this private “auxiliary method” is useful.

→ **Note:** This is an introductory example. Later, you will learn more programming techniques for evaluating formal parameters.

Namespace: Within a class, attribute names, method names, event names, constant names, type names, and alias names all **share the same namespace**. As with subroutines and function modules, there is an additional local namespace within methods. This means that local declarations override those for the whole class.



- Instance methods
 - Both static and instance components can be accessed
 - Definition using METHODS

- Static methods
 - Only static components can be accessed
 - Defined with CLASS-METHODS

```
CLASS lcl_vehicle DEFINITION.
PUBLIC SECTION.
METHODS set_type ... .
CLASS-METHODS get_n_o_vehicles
EXPORTING ex_count TYPE i.

PRIVATE SECTION.
DATA: make ..., model ...
CLASS-DATA n_o_vehicles TYPE i.
ENDCLASS.
```

```
CLASS lcl_vehicle IMPLEMENTATION.
...
METHOD get_n_o_vehicles .
ex_count = n_o_vehicles.
ENDMETHOD.
...
ENDCLASS.
```

Figure 44: Comparison of Instance Methods and Static Methods

Instance methods are defined using the syntax keyword METHODS.

Static methods are defined at class level. The restriction that only static components can be accessed applies in the implementation part. This means that static methods do not need instances, that is, they can be accessed directly through the class. This will be discussed in more detail later. The methods are defined using the syntax keyword CLASS-METHODS.

In this example, only the static attribute N_O_VEHICLES can be accessed within the static method GET_N_O_VEHICLES. All other attributes of the class are instance attributes and can only appear within instance methods.

Technical literature often refers to static methods as “**class methods**” (compare with the CLASS-METHODS syntax element). In ABAP Objects, as in C++ and Java, the official term is **static method**.

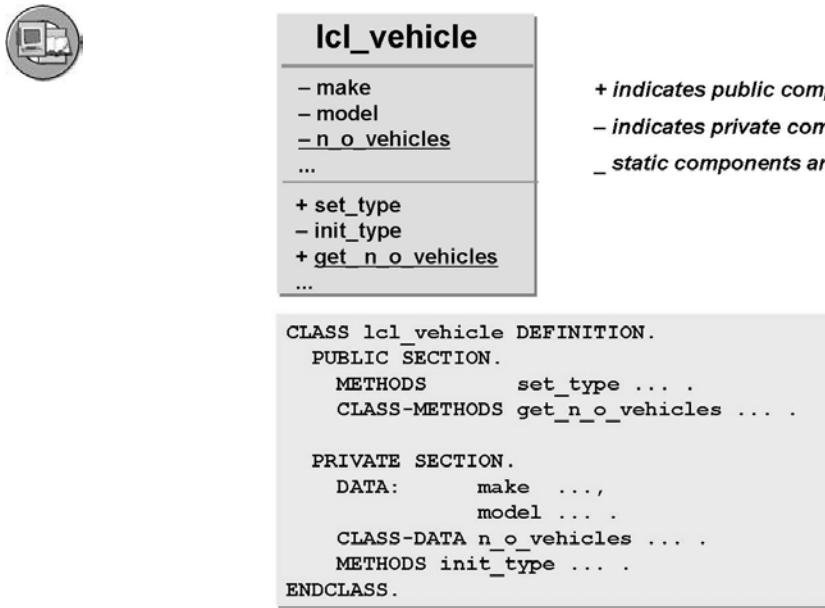


Figure 45: Visibility Sections and UML Notation

A UML class diagram lists the class name first and class attributes and methods below it. The visibility of components in a class is shown in UML using the characters + and -: Alternatively, public and private can prefix the method.

UML also allows manufacturers of modeling tools to create their own symbols for visibility. Representation of visibility characteristics is optional and is normally only used for models that are close to implementation.

Static components are marked with an underscore.

The method signature is represented as follows (optional):

- The input and output parameters and the parameters to be changed are shown in parentheses after the method name. The types are always specified after a colon.
- For a function method, the method name and the parentheses are followed by the event parameter, separated by a colon.

Objects: Instances of Classes

You could use classes to write complete applications by only using static components. However, the reasoning behind object-oriented programming is to create and work with runtime instances of classes.



- Objects are created using the statement
CREATE OBJECT ref_name
- They can only be created and addressed using reference variables

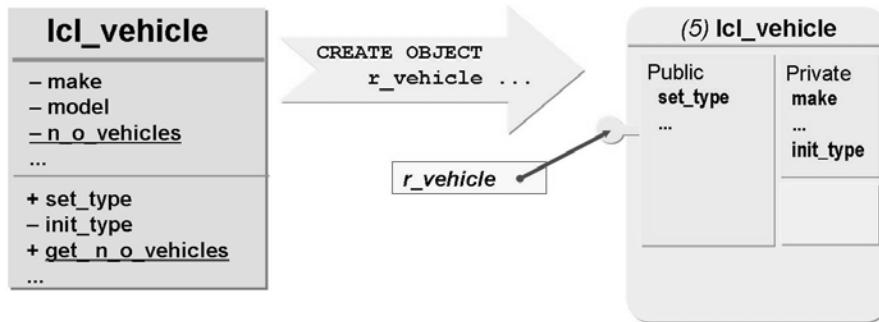


Figure 46: Overview of Instances of Classes

A class contains the generic description of an object and describes all characteristics that all objects of the class have in common. During the program runtime, the class is used to create discrete objects (instances) in the memory. This process is called **instantiation**. If this is the first time the class is accessed, the class is also loaded into the memory.

Which values may be written into which attributes is irrelevant. Technically, the object has an ID (in this case, 5) which is **not** accessible, however.

Example: Instantiation of class LCL_VEHICLE creates a vehicle object. The private attributes still contain the technical initial values.



```

...
CLASS lcl_vehicle DEFINITION.
  PUBLIC SECTION.
  ...
  PRIVATE SECTION.
  ...
ENDCLASS.

CLASS lcl_vehicle IMPLEMENTATION.
  ...
ENDCLASS.

DATA: r_vehicle1 TYPE REF TO lcl_vehicle,
      r_vehicle2 LIKE r_vehicle1.

START-OF-SELECTION.
...

```

`r_vehicle1` →

`r_vehicle2` ←

Figure 47: Definition of Reference Variables

`DATA r_vehicle1 TYPE REF TO lcl_vehicle` is used to define a reference variable, which is thereby typed as a pointer to objects of type `lcl_vehicle`. The null reference is the technical initial value of a reference variable. (The pointer is pointing to nothing.)



```

DATA: r_vehicle1 TYPE REF TO lcl_vehicle,
      r_vehicle2 LIKE r_vehicle1.

CREATE OBJECT r_vehicle1.
CREATE OBJECT r_vehicle2.

```

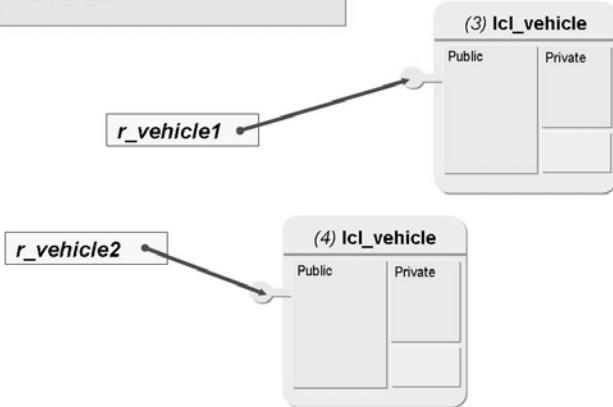


Figure 48: Creating Objects

The statement CREATE OBJECT creates an object in the memory. Its attribute values are then either initial or assigned according to the VALUE specification.

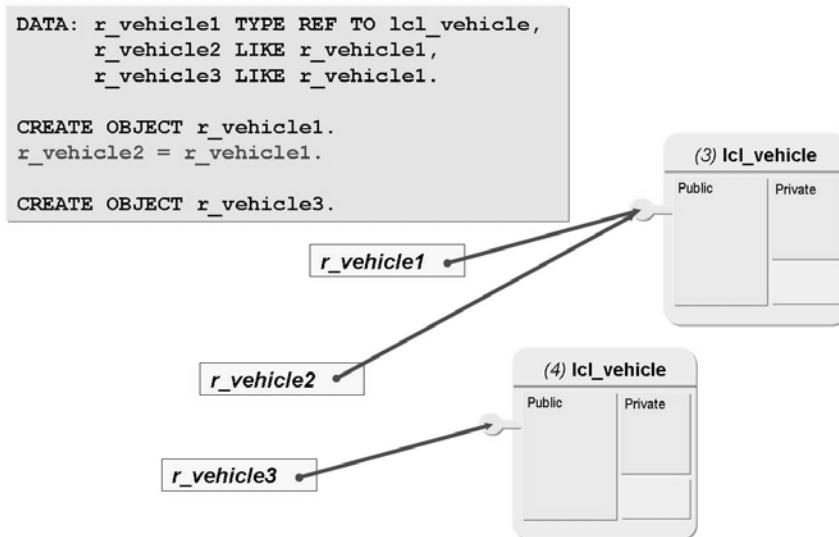


Figure 49: Reference Semantics of Object References

Reference variables can also be assigned to each other.

For the above example, this would mean that after the MOVE statement, `R_VEHICLE1` and `R_VEHICLE2` point to the same object.



- If no independent reference points to an object, the object can no longer be accessed syntactically.
- All objects that can no longer be accessed syntactically are deleted by the Garbage Collector.

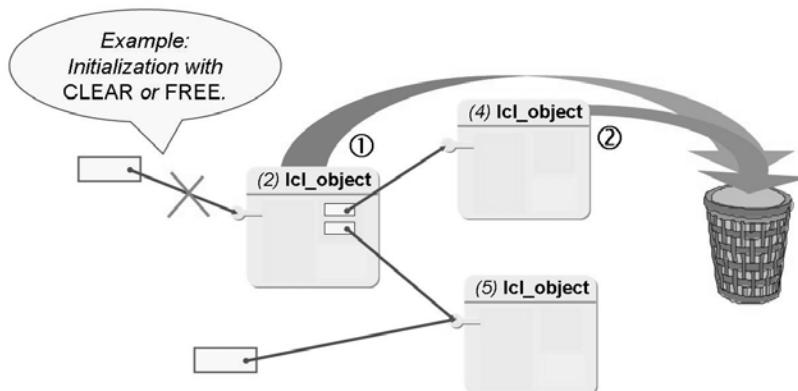


Figure 50: Garbage Collector

Independent references are references that have not been defined within a class. The **Garbage Collector** is a system routine that is automatically started whenever the runtime system does not have more important tasks to carry out.

In this example, the reference to object (2)LCL_OBJECT is initialized. Afterwards, no references point to this object. Therefore, the Garbage Collector deletes it. Consequently, no references point to object (4)LCL_OBJECT anymore, so it is deleted as well.

You can use the logical query IF r_obj IS INITIAL to determine whether r_obj contains the null reference, in other words whether it does not point to any object.

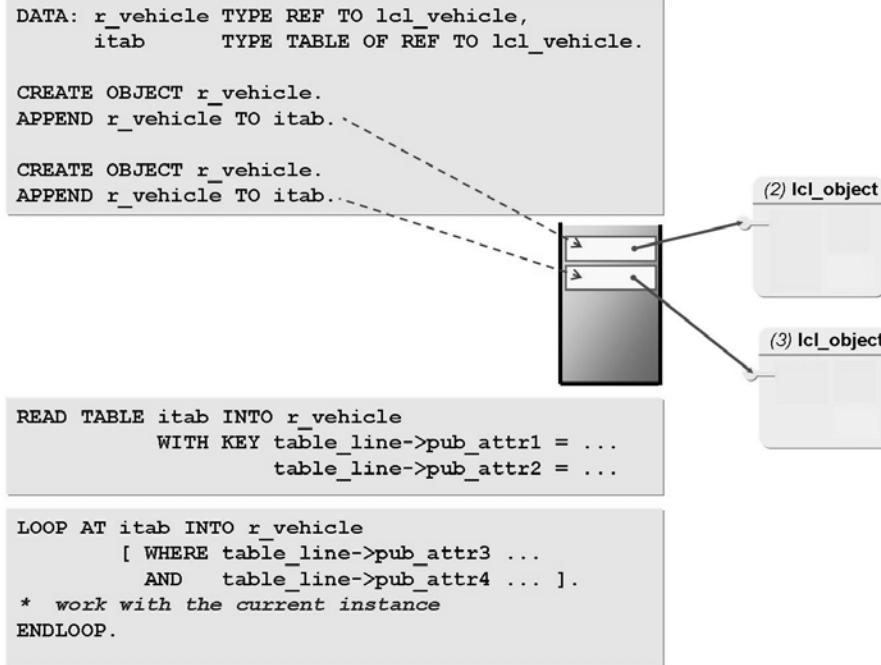


Figure 51: Reference Administration with Multiple Instantiation

If you want to keep several objects from the same class in your program, you can define an internal table that contains one column with the object references for this class. These objects can be administered in the internal table with the usual statements for internal tables such as APPEND, READ or LOOP.

To use logical conditions, the pseudo-component TABLE_LINE must be used when you use single-column internal tables. To do this, however, the relevant attributes need to be public.



```
DATA: r_vehicle TYPE REF TO lcl_vehicle,
      r_wheel    TYPE REF TO lcl_wheel.
```

```
CREATE OBJECT r_vehicle.
DO ... TIMES.
  CREATE OBJECT r_wheel.
  * add wheel to vehicle
  ...
ENDDO.
```

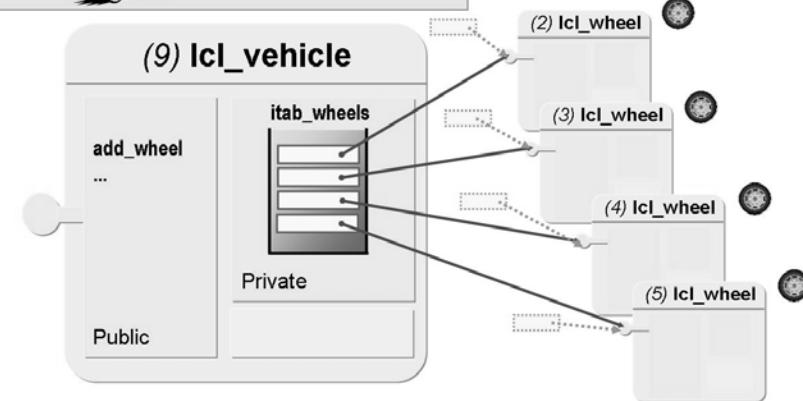


Figure 52: Example of Aggregation

The objects in the class LCL_WHEEL have their own identity. They can be created in this example, regardless of the existence of an object in the class LCL_VEHICLE.

References are transferred to objects in class LCL_VEHICLE to create the desired association.

Accessing Attributes and Methods

In this section you learn how you can use classes and their instances. That is, you will learn about the entire process, starting with the static connections of various instances, through to their practical effects.

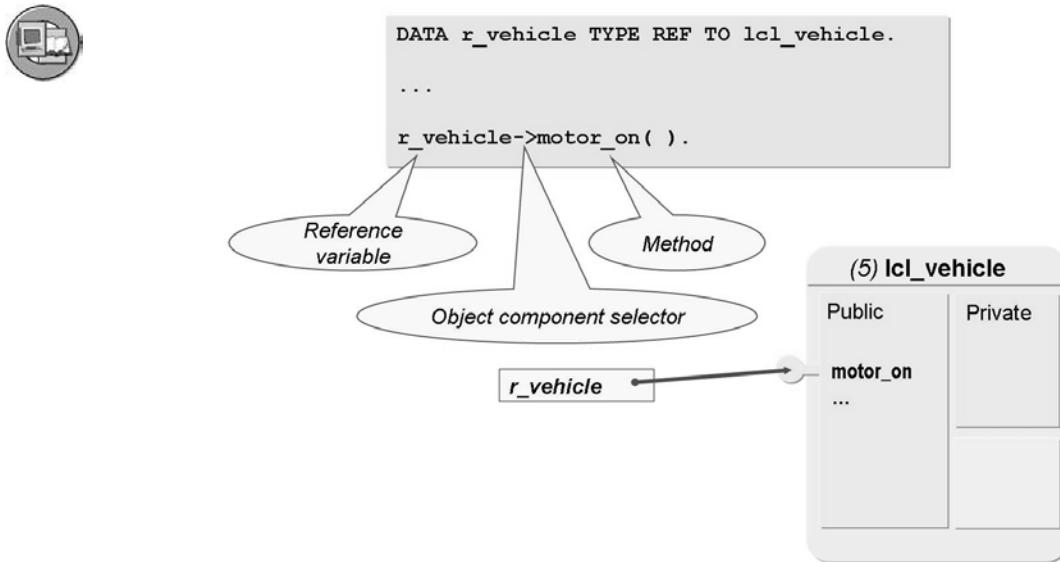


Figure 53: Calling Methods

An object that requires the services of another object sends a message to the object providing the services. This message names the operation to be executed. The implementation of this operation is known as a method. For the sake of simplicity, method will henceforth be used as a synonym for operation and message. Therefore, an object's behavior is determined by its method. A method's signature can also be used to exchange values.

The example shows the **shorter** syntax for method calls (supported as of SAP Web AS 6.10) in which the CALL-METHOD prefix is omitted.



```
CALL Method ref->method_name
EXPORTING im_par      = val_ex ...
IMPORTING ex_par      = val_im ...
CHANGING ch_par      = val_chg ...
RECEIVING re_par      = val_res
EXCEPTIONS exception = val_rc ... .
```

*See
functional
methods*

Additional shorter syntax available as of SAP Web AS 6.10:

```
ref->method_name(
    EXPORTING im_par      = val_ex ...
    IMPORTING ex_par      = val_im ...
    CHANGING ch_par      = val_chg ...
    RECEIVING re_par      = val_res
    EXCEPTIONS exception = val_rc ... ).
```

*See
functional
methods*

Example:

```
DATA: r_vehicle TYPE REF TO lcl_vehicle,
      make TYPE string, model TYPE string.
...
r_vehicle->get_type( IMPORTING ex_make = make
                      ex_model = model ).
```

Figure 54: Calling Instance Methods – Syntax

Instance methods are called with **CALL METHOD ref->method_name**. When calling an instance method from within another instance method, you can omit the instance name **ref**. The method is automatically executed for the current object.

A shorter syntax is also supported as of SAP Web AS 6.10. In this case, **CALL METHOD** is omitted and the parameters are listed in parentheses. There must be **no space** before the parentheses, but there **must be at least one** after the parentheses. When you call a method that has only one import parameter, you can specify the actual parameter in the parentheses without any other additions. When you call a method that only has import parameters, you can omit the **EXPORTING** addition.

The **RECEIVING**, **IMPORTING**, and **CHANGING** parameters are mutually exclusive. Please see the section about functional methods for more details. Otherwise, the same rules apply here as they do for calling a function module.



```
CALL METHOD class_name=>method_name
  EXPORTING im_par      = val_ex ...
  IMPORTING ex_par      = val_im ...
  CHANGING ch_par      = val_chg ...
  RECEIVING re_par      = val_res ...
  EXCEPTIONS exception = val_rc ... .
```

Refer to
functional
methods!

Shorter syntax also available as of SAP Web AS 6.10 :

```
class_name=>method_name(
  EXPORTING im_par      = val_ex ...
  IMPORTING ex_par      = val_im ...
  CHANGING ch_par      = val_chg ...
  RECEIVING re_par      = val_res ...
  EXCEPTIONS exception = val_rc ... ).
```

Refer to
functional
methods!

Example:

```
DATA number TYPE i.
...
lcl_vehicle=>get_n_o_vehicles( IMPORTING ex_count = number ).
```

Figure 55: Calling Static Methods - Syntax

Static methods (also referred to as class methods) are called using **CALL METHOD class_name=>method_name . . .**

Like static attributes, static methods are addressed with their class name, since they do not need instances.

As with instance methods, when you are calling a static method from within the class, you can omit the `classname`. Otherwise, the same rules apply here as for calling an instance method.



- **Definition:**

- Precisely one RETURNING parameter
- Otherwise, only IMPORTING parameters and exceptions are possible

- **Call:**

Explicit assignment of RECEIVING parameter or
implicit calls possible in various expressions:

- ◆ MOVE, CASE, LOOP statement
- ◆ Logical expressions (IF, WHILE, CHECK, WAIT UNTIL)
- ◆ Arithmetic and bit expressions (COMPUTE)

Possible as of SAP R/3 4.6A:

```
result = ref->func_method_name( im_par_1 = val_ex_1
                                ...
                                im_par_n = val_ex_n ).
```



```
result = class_name=>func_method_name( im_par_1 = val_ex_1
                                ...
                                im_par_n = val_ex_n ).
```

Figure 56: Functional Methods

Methods that have a RETURNING parameter are described as functional methods. This means that they can have neither an EXPORTING nor a CHANGING parameter. The RETURNING parameter must always be passed using the VALUE addition, that is, passed by value.

Functional methods can be called directly **within** various expressions:

- Logical expressions: IF, ELSEIF, WHILE, CHECK, WAIT
- Case conditions: CASE, WHEN
- Arithmetic expressions and bit expressions: COMPUTE
- Sources of values as a local copy: MOVE
- Search clauses for internal tables, assuming that the operand is not a component of the table row: LOOP AT ... WHERE



```

CLASS lcl_vehicle DEFINITION.
PUBLIC SECTION.
METHODS get_average_fuel
    IMPORTING im_distance      TYPE s_distance
          im_fuel             TYPE s_capacity
    RETURNING value(re_fuel)   TYPE s_consum.
ENDCLASS.

DATA: r_veh1 TYPE REF TO lcl_vehicle,
      r_veh2 TYPE REF TO lcl_vehicle,
      avg_fuel TYPE s_consum.
...
avg_fuel =
    r_veh1->get_average_fuel( im_distance = 500 im_fuel = '50.0' )
    + r_veh2->get_average_fuel( im_distance = 600 im_fuel = '60.0' ).

DATA number TYPE i.
...
number = lcl_vehicle=>get_n_o_vehicles( ).
```

Figure 57: Functional Methods – Examples

In the first of these examples, two calls of functional instance methods represent the two addends of an addition.

The second example shows the call of a functional static method in the short form: The NUMBER data object is the actual parameter for the method's RETURNING parameter. The detailed syntax is as follows:

```

DATA number TYPE i.
...
CALL METHOD lcl_vehicle=>get_n_o_vehicles
RECEIVING re_count = number.
```

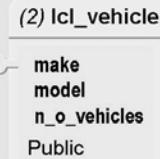


```

CLASS lcl_vehicle DEFINITION.
PUBLIC SECTION.
  DATA: make TYPE string READ-ONLY,
        model TYPE string READ-ONLY.
  CLASS-DATA: n_o_vehicles TYPE i READ-ONLY.
  ...
ENDCLASS.

...
DATA r_vehicle TYPE REF TO lcl_vehicle.
DATA: gd_make TYPE string,
      gd_count TYPE i.
...
CREATE OBJECT r_vehicle.
...
gd_make = r_vehicle->make.
gd_count = lcl_vehicle=>n_o_vehicles.

```



r_vehicle

*Only possible if
attributes are
public*

Figure 58: Accessing Public Attributes

You access public attributes from outside a class the same way as method calls: Static attributes are accessed using `classname=>static_attribute`. Instance attributes are accessed with `ref->instance_attribute`.

Constructors

There are two types of methods in ABAP Objects. They are generally not called explicitly with CALL METHOD (or the relevant short form), rather they are called implicitly. Constructors are the first type of method.

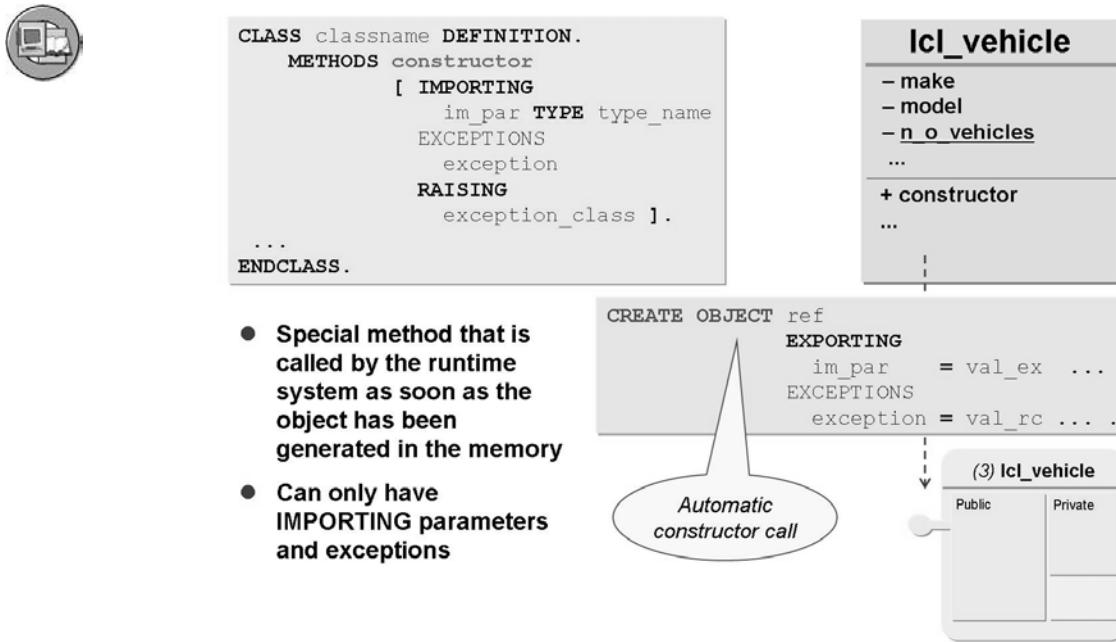


Figure 59: (Instance) Constructor

The **constructor** is a special **instance** method in a class and is always named CONSTRUCTOR. This abbreviated term actually means the instance constructor.

The constructor is automatically called at runtime with the CREATE OBJECT statement. Always consider the following points when you define constructors:

- Each class can have no more than one (instance) constructor.
 - The constructor must be defined in the public area.
 - The constructor's signature can only have importing parameters and exceptions.
 - When exceptions are raised in the constructor, instances are not created, so no main memory space is occupied.
 - Except for one exceptional case, you cannot normally call the constructor explicitly.
- **Note:** There is **no** destructor in ABAP Objects. That is, there is no instance method that is automatically called from the memory immediately before the object is deleted.

(The corresponding comments in the *SAP Library* and the menu paths outside of the *ABAP Workbench* are only contained in internal system calls.)



```
CLASS lcl_vehicle DEFINITION.  
  PUBLIC SECTION.  
    METHODS constructor IMPORTING imc_make TYPE string  
          imc_model TYPE string.  
  PRIVATE SECTION.  
    DATA: make TYPE string, model TYPE string.  
    CLASS-DATA n_o_vehicles TYPE i.  
  ENDCLASS.  
  
CLASS lcl_vehicle IMPLEMENTATION.  
  METHOD constructor.  
    CALL METHOD set_type EXPORTING im_make = imc_make  
          im_model = imc_model.  
    ADD 1 TO n_o_vehicles.  
  ENDMETHOD.  
ENDCLASS.  
  
DATA r_vehicle TYPE REF TO lcl_vehicles.  
...  
CREATE OBJECT r_vehicle  
  EXPORTING imc_make = 'Ferrari'  
        imc_model = 'F40'.
```

Figure 60: Constructor – Example

For example, a constructor is necessary if, after the instantiation of a class:

- You need to allocate resources
- You need to initialize attributes that **cannot** be covered by the VALUE addition to the DATA statement
- You need to modify static attributes
- You need to send messages containing the information that a new object was created



```
CLASS lcl_vehicle DEFINITION.
PUBLIC SECTION.
  CLASS-METHODS class_constructor.
  METHODS constructor IMPORTING ... .
PRIVATE SECTION.
  CLASS-DATA n_o_vehicles TYPE i.
ENDCLASS.
```

```
CLASS lcl_vehicle IMPLEMENTATION.
  Method class_constructor.
  ...
  ENDMETHOD.
  Method constructor.
  ...
  ENDMETHOD.
ENDCLASS.
```

```
gd_count = lcl_vehicle=>get_n_o_vehicles( ).
```

- Special method that the runtime system calls when the class is accessed for the first time during that program run
- Cannot have parameters or exceptions

If this is the first access to this class, the static constructor is executed first

Figure 61: Example of a Static Constructor

The static constructor is a special static method in a class and is always named CLASS_CONSTRUCTOR. It is executed no more than once per program (and class). The static constructor is called automatically before the class is first accessed, but **before** any of the following actions are executed **for the first time**:

- Creating an instance of this class (CREATE OBJECT)
- Accessing a static attribute of this class
- Calling a static method of this class
- Registering an event handler method for an event in this class

Always consider the following points when you define static constructors:

- Each class has no more than one static constructor
- The static constructor must be defined in the public area
- The constructor's signature **cannot** have importing parameters or exceptions
- The static constructor cannot be called explicitly

Self-Reference

In some cases, you need to have a self-reference available. In ABAP Objects, self-references are always predefined, but they are only useful in certain contexts and only there are they syntactically available.



```

CLASS lcl_vehicle DEFINITION.
  PUBLIC SECTION.
    ...
    METHODS set_type
      IMPORTING im_make TYPE string
      ...
    PRIVATE SECTION.
      DATA: make TYPE string,
      ...
  ENDCCLASS.

CLASS lcl_vehicle IMPLEMENTATION.
  Method set_type.
    DATA make TYPE string.

    me->make = im_make.
    TRANSLATE me->make TO UPPER CASE.

    make = me->make.
    CONCATENATE '_ ' make INTO make.
    ...
ENDMETHOD.
ENDCLASS.

```

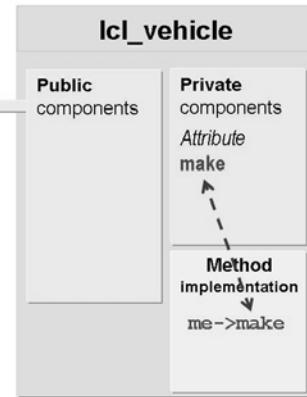


Figure 62: Self-Reference

You can address an object itself by using the predefined reference variable ME within its instance methods . Generally, you do not need to use the prefix `me->` in such cases, but you may use it to improve readability.

However, it is required when you want to show a distinction between local data objects and instance attributes with the same name.

The following case shows another important use: When a foreign method is called, a client object is to export a reference to itself. ME can then be used as an actual parameter with EXPORTING or CHANGING.

Exercise 2: Local Classes

Exercise Objectives

After completing this exercise, you will be able to:

- Declare local classes
- Define attributes
- Define and implement methods

Business Example

You are a developer for an airline corporation that owns several airline carriers. Start to develop an object-oriented program that can manage the airline carriers and their airlines.

Task 1:

Create a new program.

1. Create an executable program without a TOP include.

Program name: **ZBC401##_MAIN**

(where ## is your two-digit group number).)

2. Create an include program and include it in your ZBC401##_MAIN main program.

Program name: **ZBC401##_AIRPLANE**

(where ## is your two-digit group number).)

Task 2:

Declare a class for airplanes.

1. **Within your include program**, declare the local class LCL_AIRPLANE.
2. Define the two private instance attributes

NAME (name of airplane), data type STRING

PLANETYPE (type of airplane), data type SAPLANE-PLANETYPE

and the private static attribute

N_O_AIRPLANES (instance counter), data type I.

Continued on next page

3. Define the public instance method SET_ATTRIBUTES for setting the private instance attributes.

Your signature should consist of two suitable import parameters that are defined as compatible with the two attributes.

Implement the method in such a way that the two instance attributes are set.

4. Define the public instance method DISPLAY_ATTRIBUTES for displaying the private instance attributes.

Implement the method in such a way that the values of the two instance attributes are output as an ABAP list. You can also output icons if the ICON type group is loaded.



Hint: To do this, use the statement **TYPE-POOLS icon**.



Note: Strictly speaking, to adhere to the delegation principle, the reading of the attribute values and their output should **not** be implemented in the same method. However, do it here anyway because of time constraints.

5. Define the public static method DISPLAY_N_O_AIRPLANES to display the private static attribute.

Implement the method in such a way that the value of the static attributes is output in the ABAP list.



Note: So far, your class does not have a mechanism that ensures that the instance counter is increased each time an object is created. It is up to you to decide if you want to leave this out for now, or if you want to **temporarily** control the incrementation using the SET_ATTRIBUTES method.

Solution 2: Local Classes

Task 1:

Create a new program.

1. Create an executable program without a TOP include.

Program name: **ZBC401##_MAIN**

(where ## is your two-digit group number).)

- a) Carry out this step in the usual manner. Additional information is available in the SAP Library.

Model solution: SAPBC401_AIRS_MAIN_A

2. Create an include program and include it in your ZBC401##_MAIN main program.

Program name: **ZBC401##_AIRPLANE**

(where ## is your two-digit group number).)

- a) Carry out this step in the usual manner. Additional information is available in the SAP Library.

Model solution: SAPBC401_AIRS_A

- b) See the source code extract from the model solution.

Task 2:

Declare a class for airplanes.

1. **Within your include program**, declare the local class LCL_AIRPLANE.

- a) See the source code extract from the model solution.

2. Define the two private instance attributes

NAME (name of airplane), data type STRING

PLANETYPE (type of airplane), data type SAPLANE-PLANETYPE

and the private static attribute

N_O_AIRPLANES (instance counter), data type I.

- a) See the source code extract from the model solution.

Continued on next page

3. Define the public instance method SET_ATTRIBUTES for setting the private instance attributes.

Your signature should consist of two suitable import parameters that are defined as compatible with the two attributes.

Implement the method in such a way that the two instance attributes are set.

- a) See the source code extract from the model solution.

4. Define the public instance method DISPLAY_ATTRIBUTES for displaying the private instance attributes.

Implement the method in such a way that the values of the two instance attributes are output as an ABAP list. You can also output icons if the ICON type group is loaded.



Hint: To do this, use the statement **TYPE-POOLS icon**.



Note: Strictly speaking, to adhere to the delegation principle, the reading of the attribute values and their output should **not** be implemented in the same method. However, do it here anyway because of time constraints.

- a) See the source code extract from the model solution.

5. Define the public static method DISPLAY_N_O_AIRPLANES to display the private static attribute.

Implement the method in such a way that the value of the static attributes is output in the ABAP list.



Note: So far, your class does not have a mechanism that ensures that the instance counter is increased each time an object is created. It is up to you to decide if you want to leave this out for now, or if you want to **temporarily** control the incrementation using the SET_ATTRIBUTES method.

- a) See the source code extract from the model solution.

Result

Source code extract:

Continued on next page

SAPBC401_AIRS_MAIN_A

```
REPORT  sapbc401_airs_main_a.

TYPE-POOLS icon.

INCLUDE sapbc401_airs_a.
```

SAPBC401_AIRS_A

```
*&-----*
*&   Include          SAPBC401_AIRS_A           *
*&-----*
*-----*
*      CLASS lcl_airplane DEFINITION           *
*-----*
CLASS lcl_airplane DEFINITION.
```

PUBLIC SECTION.

```
"-----"
CONSTANTS: pos_1 TYPE i VALUE 30.
```

```
METHODS: set_attributes IMPORTING
         im_name      TYPE string
         im_planetype TYPE saplane-planetype,
         display_attributes.
```

CLASS-METHODS: display_n_o_airplanes.

PRIVATE SECTION.

```
"-----"
DATA: name      TYPE string,
      planetype TYPE saplane-planetype.
```

CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS. "lcl_airplane DEFINITION

```
*-----*
*      CLASS lcl_airplane IMPLEMENTATION       *
```

Continued on next page

```
*-----*
CLASS lcl_airplane IMPLEMENTATION.

METHOD set_attributes.
    name          = im_name.
    planetype     = im_planetype.
*   doesn't make sense so much -
*   only in order to get an effect
*   after calling display_n_o_airplanes:
    n_o_airplanes = n_o_airplanes + 1.
ENDMETHOD.                           "set_attributes

METHOD display_attributes.
    WRITE: / icon_ws_plane AS ICON,
        / 'Name of airplane:'(001), AT pos_1 name,
        / 'Airplane type'(002), AT pos_1 planetype.
ENDMETHOD.                           "display_attributes

METHOD display_n_o_airplanes.
    WRITE: /, / 'Total number of planes'(cal),
        AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
ENDMETHOD.                           "display_n_o_airplanes

ENDCLASS.                            "lcl_airplane IMPLEMENTATION
```

Exercise 3: Objects

Exercise Objectives

After completing this exercise, you will be able to:

- Define reference variables
- Instantiate objects
- Process object references using an internal table

Business Example

Create instances of your airplane class and ensure that their addresses are not lost.

Task 1:

Define reference variables.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.
Define a reference variable for the instances of your class LCL_AIRPLANE.
2. Define an internal table for buffering references to instances of the class LCL_AIRPLANE.

Task 2:

Create airplane objects.

1. Create several instances of the local class LCL_AIRPLANE and buffer their references into the internal table.
2. Observe the execution of the program in the *ABAP Debugger*.

Solution 3: Objects

Task 1:

Define reference variables.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.

Define a reference variable for the instances of your class LCL_AIRPLANE.

- a) Model solution: SAPBC401_AIRS_MAIN_B
- b) See the source code extract from the model solution.

2. Define an internal table for buffering references to instances of the class LCL_AIRPLANE.

- a) See the source code extract from the model solution.

Task 2:

Create airplane objects.

1. Create several instances of the local class LCL_AIRPLANE and buffer their references into the internal table.
 - a) See the source code extract from the model solution.

Continued on next page

2. Observe the execution of the program in the *ABAP Debugger*.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.

Result

Source code extract:

```
SAPBC401_AIRS_MAIN_B

REPORT  sapbc401_airs_main_b.

TYPE-POOLS icon.

INCLUDE sapbc401_airs_a.

DATA: r_plane TYPE REF TO lcl_airplane,
      plane_list TYPE TABLE OF REF TO lcl_airplane.

START-OF-SELECTION.
*#####
CREATE OBJECT r_plane.
APPEND r_plane TO plane_list.

CREATE OBJECT r_plane.
APPEND r_plane TO plane_list.

CREATE OBJECT r_plane.
APPEND r_plane TO plane_list.
```


Exercise 4: Method Calls

Exercise Objectives

After completing this exercise, you will be able to:

- Call non-functional methods
- Define functional methods
- Call functional methods

Business Example

You need to fill the attributes of the “empty” airplane objects with suitable values.

Task 1:

Call the methods of your class.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.

Call the static method DISPLAY_N_O_AIRPLANES twice: Once before and once after the instantiations.



Caution: For reasons explained in the first exercise, you may not notice any effects at the moment.

2. Use the SET_ATTRIBUTES method to set the attributes for all objects already created. Choose a unique name for the airplanes. When you are assigning airplane types, use the information in the SAPLANE table as a guide (for example, '747-400'). Also use at least one **invalid** airplane type.
3. Display the attribute values for all the airplanes in a loop in the ABAP list using the DISPLAY_ATTRIBUTES method.

Task 2:

Add a functional method to your class.

1. In your class, define the public static functional method GET_N_O_AIRPLANES. The signature must only consist of the result parameter RE_COUNT, which must be a whole number.
2. Call this method in the main program and output the value in the ABAP list.

Solution 4: Method Calls

Task 1:

Call the methods of your class.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.

Call the static method DISPLAY_N_O_AIRPLANES twice: Once before and once after the instantiations.



Caution: For reasons explained in the first exercise, you may not notice any effects at the moment.

- a) Model solution: SAPBC401_AIRS_MAIN_C
- b) See the source code extract from the model solution.
2. Use the SET_ATTRIBUTES method to set the attributes for all objects already created. Choose a unique name for the airplanes. When you are assigning airplane types, use the information in the SAPLANE table as a guide (for example, '747-400'). Also use at least one **invalid** airplane type.
 - a) See the source code extract from the model solution.
3. Display the attribute values for all the airplanes in a loop in the ABAP list using the DISPLAY_ATTRIBUTES method.
 - a) See the source code extract from the model solution.

Task 2:

Add a functional method to your class.

1. In your class, define the public static functional method GET_N_O_AIRPLANES. The signature must only consist of the result parameter RE_COUNT, which must be a whole number.
 - a) See the source code extract from the model solution.
2. Call this method in the main program and output the value in the ABAP list.
 - a) See the source code extract from the model solution.

Result

Source code:

Continued on next page

SAPBC401_AIRS_MAIN_C

```

REPORT  sapbc401_airs_main_c.

TYPE-POOLS icon.

INCLUDE sapbc401_airs_c.

DATA: r_plane TYPE REF TO lcl_airplane,
      plane_list TYPE TABLE OF REF TO lcl_airplane,
      count TYPE i.

START-OF-SELECTION.
*#####
lcl_airplane=>display_n_o_airplanes( ).

CREATE OBJECT r_plane.
APPEND r_plane TO plane_list.
r_plane->set_attributes( im_name = 'LH Berlin'
                           im_planetype = 'A321' ).

CREATE OBJECT r_plane.
APPEND r_plane TO plane_list.
r_plane->set_attributes( im_name = 'AA New York'
                           im_planetype = '747-400' ).

CREATE OBJECT r_plane.
APPEND r_plane TO plane_list.
r_plane->set_attributes( im_name = 'US Hercules'
                           im_planetype = '747-500' ).

LOOP AT plane_list INTO r_plane.
  r_plane->display_attributes( ).
ENDLOOP.

* long syntax for functional call:
* CALL METHOD lcl_airplane=>get_n_o_airplanes
*   RECEIVING

```

Continued on next page

```

*      re_count = count.

* a little bit shorter:
* lcl_airplane=>get_n_o_airplanes( RECEIVING re_count = count ).

* the shortest syntax for functional call:
count = lcl_airplane=>get_n_o_airplanes( ).  

SKIP 2.  

WRITE: / 'Gesamtzahl der Flugzeuge'(cal), count.

```

SAPBC401_AIRS_C

```

*&-----*
*& Include          SAPBC401_AIRS_C
*& Show functional static method get_n_o_airplanes
*&-----*
*-----*
*      CLASS lcl_airplane DEFINITION
*-----*
CLASS lcl_airplane DEFINITION.

PUBLIC SECTION.
-----  

CONSTANTS: pos_1 TYPE i VALUE 30.

METHODS: set_attributes IMPORTING
         im_name      TYPE string
         im_planetype TYPE saplane-planetype,
         display_attributes.

CLASS-METHODS: display_n_o_airplanes,
               get_n_o_airplanes RETURNING value(re_count) TYPE i.

PRIVATE SECTION.
-----  

DATA: name      TYPE string,
      planetype TYPE saplane-planetype.

CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.           "lcl_airplane DEFINITION

```

Continued on next page

```
*-----*
*      CLASS lcl_airplane IMPLEMENTATION
*-----*
CLASS lcl_airplane IMPLEMENTATION.

METHOD set_attributes.
    name      = im_name.
    planetype = im_planetyp.
    n_o_airplanes = n_o_airplanes + 1.
ENDMETHOD.                      "set_attributes

METHOD display_attributes.
    WRITE: / icon_ws_plane AS ICON,
        / 'Name des Flugzeugs:'(001), AT pos_1 name,
        / 'Flugzeugtyp'(002), AT pos_1 planetyp.
ENDMETHOD.                      "display_attributes

METHOD display_n_o_airplanes.
    WRITE: /, / 'Gesamtzahl der Flugzeuge'(cal),
        AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
ENDMETHOD.                      "display_n_o_airplanes

METHOD get_n_o_airplanes.
    re_count = n_o_airplanes.
ENDMETHOD.                      "get_n_o_airplanes

ENDCLASS.                      "lcl_airplane IMPLEMENTATION
```


Exercise 5: Constructors

Exercise Objectives

After completing this exercise, you will be able to:

- Define and implement instance constructors
- Create instances of classes that contain an instance constructor
- Define and implement static constructors

Business Example

Make your program more realistic: The airplane objects receive their attributes as soon as they are created.

Task 1:

Define an instance constructor.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise. (where ## is your two-digit group number).)

Define a reference variable with a suitable signature for the instances of your class LCL_AIRPLANE.

Implement it so that the two instance attributes are set and the instance counter N_O_AIRPLANES is increased by one.

2. If you previously used the SET_ATTRIBUTES method to increase the instance counter, remove the relevant statement from the method now.

Task 2:

Create airplane objects.

1. Your CREATE OBJECT statements from the previous exercise should now be syntactically incorrect. Adapt and correct them.
2. If necessary, remove the calls of the SET_ATTRIBUTES method.
3. Follow the execution of the program in the *ABAP Debugger*.

Task 3:

Define and implement the **static constructor**

1. Define the LIST_OF_PLANETYPES internal table as a private class attribute.

Continued on next page

To specify the type for this internal table, define the table types Z_##_PLANETYPE in the ABAP Dictionary. Alternatively, you can also use the pre-defined types TY_PLANETYPES.

Use the PLANETYPE field as a key for the internal table.

2. Define the static constructor in the LCL_AIRPLANE class.

Implement the constructor in such a way that the LIST_OF_PLANETYPES internal table is filled with all rows from the SAPLANE database table. You can use the ARRAY FETCH technique for this purpose.

3. Check that the static constructor is called correctly and that the internal table is filled in the main program. Where is the static constructor called in the main program?

Solution 5: Constructors

Task 1:

Define an instance constructor.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise. (where ## is your two-digit group number).)

Define a reference variable with a suitable signature for the instances of your class LCL_AIRPLANE.

Implement it so that the two instance attributes are set and the instance counter N_O_AIRPLANES is increased by one.

- a) Model solution: SAPBC401_AIRS_MAIN_D
- b) See the source code extract from the model solution.
2. If you previously used the SET_ATTRIBUTES method to increase the instance counter, remove the relevant statement from the method now.
 - a) See the source code extract from the model solution.

Task 2:

Create airplane objects.

1. Your CREATE OBJECT statements from the previous exercise should now be syntactically incorrect. Adapt and correct them.
 - a) See the source code extract from the model solution.
2. If necessary, remove the calls of the SET_ATTRIBUTES method.
 - a) See the source code extract from the model solution.
3. Follow the execution of the program in the *ABAP Debugger*.
 - a) Carry out this step in the usual manner. Additional information on the ABAP Debugger is available in the SAP Library.

Task 3:

Define and implement the **static constructor**

1. Define the LIST_OF_PLANETYPES internal table as a private class attribute.

Continued on next page

To specify the type for this internal table, define the table types Z_##_PLANETYPE in the ABAP Dictionary. Alternatively, you can also use the pre-defined types TY_PLANETYPES.

Use the PLANETYPE field as a key for the internal table.

- a) See the source code extract from the model solution.
2. Define the static constructor in the LCL_AIRPLANE class.
Implement the constructor in such a way that the LIST_OF_PLANETYPES internal table is filled with all rows from the SAPLANE database table. You can use the ARRAY FETCH technique for this purpose.
a) See the source code extract from the model solution.
 3. Check that the static constructor is called correctly and that the internal table is filled in the main program. Where is the static constructor called in the main program?
a) Before the class is first accessed (before DISPLAY_N_O_AIRPLANES in this case).

Result

Source code extract:

SAPBC401_AIRS_MAIN_D

```
REPORT  sapbc401_airs_main_d.

TYPE-POOLS icon.

INCLUDE sapbc401_airs_d.

DATA: r_plane TYPE REF TO lcl_airplane,
      plane_list TYPE TABLE OF REF TO lcl_airplane.

START-OF-SELECTION.
*#####
lcl_airplane=>display_n_o_airplanes( ).

CREATE OBJECT r_plane EXPORTING im_name = 'LH Berlin'
                                         im_planetype = 'A321'.
```

Continued on next page

```

APPEND r_plane TO plane_list.

r_plane->display_attributes( ).

CREATE OBJECT r_plane EXPORTING im_name = 'AA New York'
          im_planetype = '747-400'.
APPEND r_plane TO plane_list.

r_plane->display_attributes( ).

CREATE OBJECT r_plane EXPORTING im_name = 'US Hercules'
          im_planetype = '747-500'.
APPEND r_plane TO plane_list.

r_plane->display_attributes( ).

lcl_airplane=>display_n_o_airplanes( ).

```

SAPBC401_AIRS_D

```

*&-----*
*&   Include      SAPBC401_AIRS_D
*&-----*
*-----*
*       CLASS lcl_airplane DEFINITION
*-----*
CLASS lcl_airplane DEFINITION.
PUBLIC SECTION.
"
CONSTANTS: pos_1 TYPE i VALUE 30.

METHODS: constructor IMPORTING
          im_name      TYPE string
          im_planetype TYPE saplane-planetype,
display_attributes.

CLASS-METHODS: class_constructor.

CLASS-METHODS: display_n_o_airplanes.

```

Continued on next page

```

PRIVATE SECTION.

-----  

DATA: name      TYPE string,  

      planetype TYPE saplane-planetype.  

CLASS-DATA: n_o_airplanes TYPE i.  

CLASS-DATA: list_of_planetypes TYPE ty_planetypes. "itab type in Dic.  

ENDCLASS.  

*-----*-----*-----*-----*  

*      CLASS lcl_airplane IMPLEMENTATION  

*-----*-----*-----*-----*  

CLASS lcl_airplane IMPLEMENTATION.  

METHOD constructor.  

  name      = im_name.  

  planetype = im_planetype.  

  n_o_airplanes = n_o_airplanes + 1.  

ENDMETHOD.           "constructor  

METHOD class_constructor.  

  SELECT * FROM saplane INTO TABLE list_of_planetypes.  

ENDMETHOD.  

METHOD display_attributes.  

  WRITE: / icon_ws_plane AS ICON,  

         / 'Name des Flugzeugs'(001), AT pos_1 name,  

         / 'Type of airplane: '(002), AT pos_1 planetype.  

ENDMETHOD.           "display_attributes  

METHOD display_n_o_airplanes.  

  WRITE: /, / 'Number of airplanes: '(cal),  

         AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.  

ENDMETHOD.           "display_n_o_airplanes  

ENDCLASS.           "lcl_airplane IMPLEMENTATION

```

Exercise 6: Private Methods

Exercise Objectives

After completing this exercise, you will be able to:

- Use the delegation principle
- Structure and call private methods
- Use the keyword ME

Business Example

You want to display the technical status data for an airplane.

Task 1:

Declare another method.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.

Within your LCL_AIRPLANE class, define the **private** static method GET_TECHNICAL_ATTRIBUTES.

The signature must consist of the import parameter for the airplane type and the two export parameters for weight and tank capacity. Use the transparent table SAPLANE as a guide for specifying the types of these formal parameters.

2. Implement the method in such a way that the values for the export parameters can be determined by single-record access to the internal table LIST_OF_PLANETYPES.

 **Note:** If the table does not contain any values for the airplane type, it should be ignored for the time being.

Strictly speaking, the correct unit of measure should also be selected and exported. However, time constraints mean that you do not need to do it in this exercise.

Task 2:

Call the method.

1. If possible, call the method GET_TECHNICAL_ATTRIBUTES from the main program.

Continued on next page

2. Call the method GET_TECHNICAL_ATTRIBUTES from the method DISPLAY_ATTRIBUTES to obtain additional technical data. Enter this additional data in the calling method.
3. Observe the execution of the program in the *ABAP Debugger*.
4. Which alternative solutions could be used to solve the tasks?

Task 3: (optional)

Create and handle an exception.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.

If reading the airplane type in the method GET_TECHNICAL_ATTRIBUTES is not successful (SY-SUBRC \neq 0), the exception WRONG_PLANETYPE should be created.

To do this, use the EXCEPTION addition in the method interface.

The exception should be created or triggered using the RAISE command.

→ **Note:** You probably still know this classic exception handling technique from using function modules.

At a later stage in this course, we will replace this classic, older technique with a modern technique of exception handling using exception classes.

2. When you call the private method GET_TECHNICAL_ATTRIBUTES in the calling method DISPLAY_ATTRIBUTES, this exception must now be declared using the EXCEPTIONS addition.

If an exception occurs in GET_TECHNICAL_ATTRIBUTES, you can check this using the SY-SUBRC in the calling method (see known concept with function modules).

In this exercise, the use of an incorrect airplane type when SY-SUBRC \neq 0 should simply be documented with a WRITE statement.

Solution 6: Private Methods

Task 1:

Declare another method.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.

Within your LCL_AIRPLANE class, define the **private** static method GET_TECHNICAL_ATTRIBUTES.

The signature must consist of the import parameter for the airplane type and the two export parameters for weight and tank capacity. Use the transparent table SAPLANE as a guide for specifying the types of these formal parameters.

- a) Model solution: SAPBC401_AIRS_MAIN_E
- b) See the source code extract from the model solution.
2. Implement the method in such a way that the values for the export parameters can be determined by single-record access to the internal table LIST_OF_PLANETYPES.



Note: If the table does not contain any values for the airplane type, it should be ignored for the time being.

Strictly speaking, the correct unit of measure should also be selected and exported. However, time constraints mean that you do not need to do it in this exercise.

- a) See the source code extract from the model solution.

Task 2:

Call the method.

1. If possible, call the method GET_TECHNICAL_ATTRIBUTES from the main program.
 - a) If it is a private method, you cannot call it from the main program. This is only possible for public methods.
2. Call the method GET_TECHNICAL_ATTRIBUTES from the method DISPLAY_ATTRIBUTES to obtain additional technical data. Enter this additional data in the calling method.
 - a) See the source code extract from the model solution.

Continued on next page

3. Observe the execution of the program in the *ABAP Debugger*.
 - a) Carry out this step in the usual manner. Additional information on the ABAP Debugger is available in the SAP Library.
4. Which alternative solutions could be used to solve the tasks?

Answer:

- **Do not use the PLANETYPE import parameter.**

The method can be defined as an instance method without an import parameter. This method then accesses the class attribute LIST_OF_PLANETYPES with ME->PLANETYPE.

- **Do not use the method at all**

In the method display_attributes, you can access the class attribute LIST_OF_PLANETYPES directly with ME->PLANETYPE.

Task 3: (optional)

Create and handle an exception.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise.

If reading the airplane type in the method GET_TECHNICAL_ATTRIBUTES is not successful (SY-SUBRC <> 0), the exception WRONG_PLANETYPE should be created.

To do this, use the EXCEPTION addition in the method interface.

The exception should be created or triggered using the RAISE command.

→ **Note:** You probably still know this classic exception handling technique from using function modules.

At a later stage in this course, we will replace this classic, older technique with a modern technique of exception handling using exception classes.

- a) Model solution: SAPBC401_AIRS_MAIN_F
2. When you call the private method GET_TECHNICAL_ATTRIBUTES in the calling method DISPLAY_ATTRIBUTES, this exception must now be declared using the EXCEPTIONS addition.

Continued on next page

If an exception occurs in GET_TECHNICAL_ATTRIBUTES, you can check this using the SY-SUBRC in the calling method (see known concept with function modules).

In this exercise, the use of an incorrect airplane type when SY-SUBRC \neq 0 should simply be documented with a WRITE statement.

- a) Refer to the model solution.

Result

Source code:

SAPBC401_AIRS_F

```
*&-----*
*&   Include          SAPBC401_AIRS_F
*&-----*
*-----*
*      CLASS lcl_airplane DEFINITION
*-----*
CLASS lcl_airplane DEFINITION.

PUBLIC SECTION.

CONSTANTS: pos_1 TYPE i VALUE 30.

METHODS: constructor IMPORTING
          im_name      TYPE string
          im_planetype TYPE saplane-planetype,
          display_attributes.

CLASS-METHODS: display_n_o_airplanes.
CLASS-METHODS: class_constructor.

PRIVATE SECTION.

CLASS-METHODS: get_technical_attributes
               IMPORTING im_type      TYPE saplane-planetype
               EXPORTING ex_weight    TYPE s_plan_wei
                           ex_tankcap   TYPE s_capacity
               EXCEPTIONS wrong_planetype.
```

Continued on next page

```

DATA: name      TYPE string,
      planetype TYPE saplane-planetype.

CLASS-DATA: list_of_planetypes TYPE ty_planetypes.
CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.           "lcl_airplane DEFINITION

*-----*
*      CLASS lcl_airplane IMPLEMENTATION          *
*-----*
CLASS lcl_airplane IMPLEMENTATION.

METHOD class_constructor.
  SELECT * FROM saplane INTO TABLE list_of_planetypes.
ENDMETHOD.          "class_constructor

METHOD constructor.
  name      = im_name.
  planetype = im_planetype.
  n_o_airplanes = n_o_airplanes + 1.
ENDMETHOD.          "constructor

METHOD display_attributes.

DATA: weight TYPE saplane-weight,
      cap TYPE saplane-tankcap.

get_technical_attributes(
  EXPORTING im_type = planetype
  IMPORTING ex_weight = weight
            ex_tankcap = cap
  EXCEPTIONS wrong_planetype = 4 ).

WRITE: / icon_ws_plane AS ICON,
       / 'Name of airplane'(001), AT pos_1 name,
       / 'Type of airplane:'(002), AT pos_1 planetype.
IF sy-subrc <> 0.

WRITE: / icon_failure AS ICON, 'WRONG_PLANETYPE'.

```

Continued on next page

```
ELSE .
  WRITE: / 'weight of airplane'(003),
          AT pos_1 weight LEFT-JUSTIFIED,
  / 'tank capacity of airplane '(004),
          AT pos_1 cap LEFT-JUSTIFIED.

ENDIF.
ENDMETHOD.           "display_attributes

METHOD display_n_o_airplanes.
  WRITE: /, / 'Number of airplanes: '(ca1),
          AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
ENDMETHOD.           "display_n_o_airplanes

METHOD get_technical_attributes.
  DATA: wa TYPE saplane.

  READ TABLE list_of_planetypes INTO wa
    WITH TABLE KEY planetype = im_type
      TRANSPORTING weight tankcap.

  IF sy-subrc = 0.
    ex_weight = wa-weight.
    ex_tankcap = wa-tankcap.
  ELSE.
    RAISE wrong_planetype.
  ENDIF.

ENDMETHOD.           "get_technical_attributes

ENDCLASS.           "lcl_airplane IMPLEMENTATION
```



Lesson Summary

You should now be able to:

- Define classes
- Generate and delete objects
- Access attributes
- Call methods

Related Information

Further information about this subject is available in the SAP Library and the ABAP keyword documentation for the individual statements.



Unit Summary

You should now be able to:

- Explain the differences between procedural and object-oriented programming models
- List the advantages of the object-oriented programming model
- Name the most important diagram types in UML
- Create simple class diagrams
- Create simple object diagrams
- Describe sequence diagrams
- Define classes
- Generate and delete objects
- Access attributes
- Call methods



Test Your Knowledge

1. The object-oriented programming model was developed considerably later than the procedural one. It offers more options for solving problems that previously could not be solved with purely procedural programming languages.

Determine whether this statement is true or false.

- True
- False

2. What does multiple instantiation mean?

3. What does encapsulation mean?

4. In ABAP Objects, what is meant by the term class?

5. What is the difference between a class's static components and its instance components?

6. In ABAP Objects, what is meant by the term constructor?

7. You are defining a class. Must you always also define a constructor?



Answers

1. The object-oriented programming model was developed considerably later than the procedural one. It offers more options for solving problems that previously could not be solved with purely procedural programming languages.

Answer: False

Refer to the relevant section of the lesson.

2. What does multiple instantiation mean?

Answer: The ability to create and manage any number of runtime instances for each program context

Refer to the relevant section of the lesson.

3. What does encapsulation mean?

Answer: Gathering data and functions into reusable units from which users can only call certain functions and cannot access the data **directly**.

Refer to the relevant section of the lesson.

4. In ABAP Objects, what is meant by the term class?

Answer: A class is the technical description of identical objects. It can contain attribute and method definitions and can generally also contain the implementations of methods.

Refer to the relevant section of the lesson

5. What is the difference between a class's static components and its instance components?

Answer: You must access static components through the class. There are static components for each program and class no more than once in the memory. An object must be instantiated to access them.

You must access instance components via the objects of this class. Any number of instance components per program and class may be contained in the memory.

Refer to the relevant section of the lesson.

6. In ABAP Objects, what is meant by the term constructor?

Answer: A class's constructor is a special method and is always named CONSTRUCTOR. It is normally only called by the runtime system, whenever an object of this class is created using CREATE OBJECT. Therefore, it is referred to more specifically as the instance constructor.

Refer to the relevant section of the lesson.

7. You are defining a class. Must you always also define a constructor?

Answer: No.

Refer to the relevant section of the lesson.

Unit 2

Object-Oriented Concepts and Programming Techniques

Unit Overview

This unit deals with the basic programming techniques common to all object-oriented languages. As far as these concepts are concerned, the only difference between ABAP Objects and other languages like Java or C++ is the syntax.

Every concept exists for a specific reason. It is important that you understand them **all** without exception so that you can use each of them effectively later on. You can only capitalize on the strengths of object-oriented programming if you use all of the concepts in the intended manner.



Unit Objectives

After completing this unit, you will be able to:

- Define inheritance relationships between classes
- Redefine methods
- Create up-cast assignments (Widening Cast)
- Create down-cast assignments (Narrowing Cast)
- Explain the concept of polymorphism with reference to inheritance
- Use cast assignments with inheritance to make generic calls
- Define and implement interfaces
- Implement interface methods
- Use interface references to make up-cast assignments
- Use interface references to make down-cast assignments
- Explain the term polymorphism with reference to interfaces
- Use cast assignments with interfaces to make generic calls
- Define and trigger events
- Handle events
- Register and deregister event handling

- Explain the key differences between explicit method calls and event-controlled method calls

Unit Contents

Lesson: Inheritance and Casting	103
Exercise 7: Class Hierarchies	123
Exercise 8: Polymorphism	133
Exercise 9: Aggregation and Generic Calls.....	137
Lesson: Interfaces and Casting.....	145
Exercise 10: Interface Definition und Implementation	161
Exercise 11: Use of Interfaces	177
Lesson: Events.....	184
Exercise 12: Events in Superclasses	193
Exercise 13: Events in Interfaces (Optional).....	207

Lesson: Inheritance and Casting

Lesson Overview

In this lesson, you will learn how to create class hierarchies using ABAP Objects. The first step will be to program the relevant relationship types that were devised in the modeling process. Then you will learn to identify a number of interesting programming possibilities that inheritance provides.



Lesson Objectives

After completing this lesson, you will be able to:

- Define inheritance relationships between classes
- Redefine methods
- Create up-cast assignments (Widening Cast)
- Create down-cast assignments (Narrowing Cast)
- Explain the concept of polymorphism with reference to inheritance
- Use cast assignments with inheritance to make generic calls

Business Example

You want to implement generalization/specialization relationships from your model in ABAP Objects.

Creating Generalization/Specialization Relationships Using Inheritance

Specialization (UML) is a relationship in which one class (the subclass) inherits all the main characteristics of another class (the superclass). The subclass can also add new components (attributes, methods, and so on) and replace the implementations with inherited methods. In the latter case, the method name in the UML diagram is renamed within the subclass.

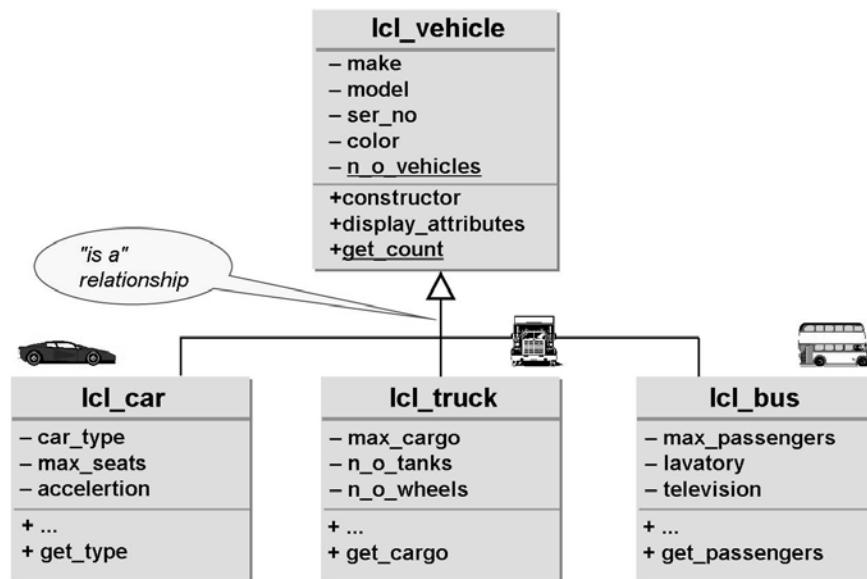


Figure 63: Example of Generalization/Specialization

Specialization is an implementation relationship that emphasizes similarities of the classes. In the example above, the similarities of classes LCL_CAR, LCL_TRUCK, and LCL_BUS are extracted to a superclass, LCL_VEHICLE. Therefore, the components they have in common are only defined and implemented in the superclass. They also exist in all subclasses.

Specialization is often described as an “is a” relationship. In this example, you would say: “A truck is a (specific) vehicle.”

Reversing the point of view is referred to as **generalization**.



- Common components only exist once in the superclass, so they can be maintained centrally
- Components in the superclasses are available in all subclasses: avoids redundant implementations
- Subclasses contain extensions/changes ("programming by difference")
- Subclasses are dependent on superclasses ("white box reuse")

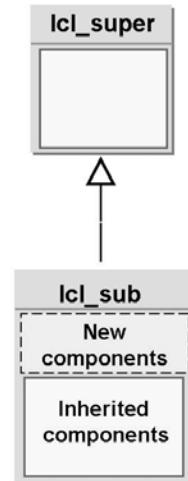


Figure 64: Characteristics of Generalization/Specialization

Generalization/specialization, if used properly, provides a significantly better structure for your software because commonly used elements only need to be stored once in a central location (in the superclass) and are then automatically available to all subclasses. Changes made at a later stage have an immediate effect on the subclasses. Therefore, you must not alter the semantics when you change a superclass.

You need exact knowledge of the implementation of the superclass to decide whether the inherited components from the superclass are sufficient for the subclass or if they need to be extended. Generalization/specialization therefore provides very strong links between the superclass and the subclass.

When you develop additional subclasses, you often have to adapt the superclass, too. Therefore, the creation of a subclass sometimes leads to additional requirements for the superclass, for example, when a subclass requires certain protected components or when the details of a superclass's implementation are required to change the method implementations in subclasses. The developer of a (super)class cannot normally predict everything that the subclasses will later require from the superclass.



```
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION.
METHODS estimate_fuel
IMPORTING im_distance      TYPE s_distance
RETURNING value(re_fuel)    TYPE s_capacity.

PRIVATE SECTION.
DATA: make      TYPE string,
...
ENDCLASS.
```



```
CLASS lcl_truck DEFINITION INHERITING FROM lcl_vehicle.

PUBLIC SECTION.
METHODS get_cargo RETURNING value(re_cargo) TYPE s_plan_car.

PRIVATE SECTION.
DATA max_cargo TYPE s_plan_car.

ENDCLASS.
```

Figure 65: Inheritance – Syntax

In ABAP Objects, an inheritance relationship is defined for a subclass using the **INHERITING FROM** addition, followed by the superclass that is directly above the subclass. Inheritance hierarchies of varying complexity arise when this superclass inherits from another superclass above it. In contrast, there is no multiple inheritance in ABAP Objects. That is, only one superclass can be specified directly above a class. However, you can use interfaces in ABAP Objects to simulate multiple inheritance.

Inheritance should be used to implement generalization and specialization relationships. A superclass is a generalization of its subclasses. The subclasses are in turn different specializations of their superclass. Thus, only additions or changes are permitted in ABAP Objects, which means that you can never remove anything from a superclass in a subclass.

Inheritance is a one-sided relationship. In other words, subclasses recognize their direct superclasses but (super)classes do not recognize their subclasses. In this example, the subclass also contains the ESTIMATE_FUEL method. The subclass also defines the GET_CARGO method.

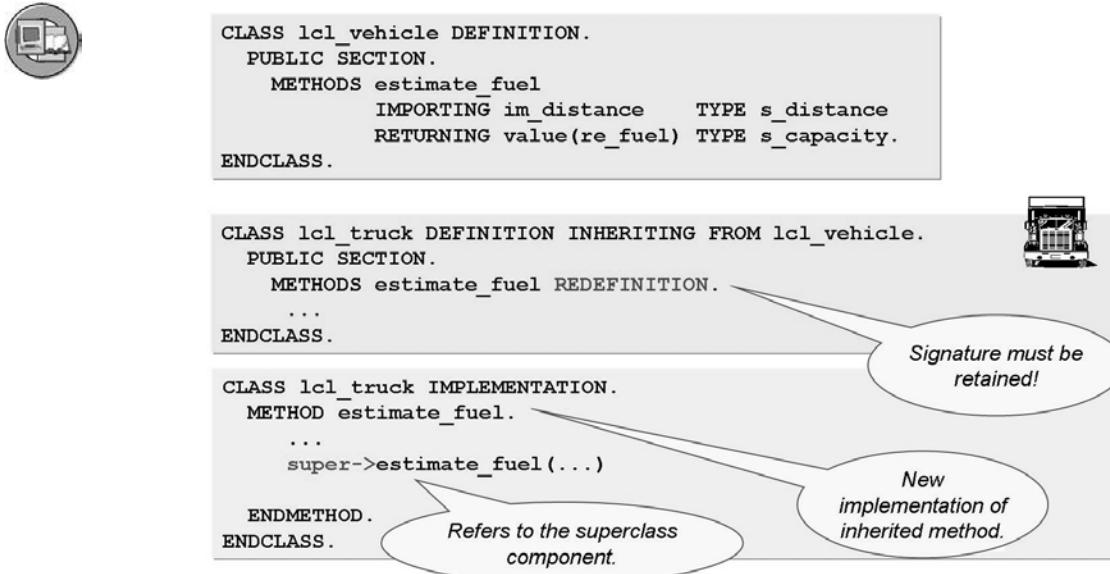


Figure 66: Redefining Methods

Redefinition is when the implementation of an inherited instance method is changed for the subclass, without changing the signature. At the same time, the visibility section for the superclass must remain the same. (Therefore, redefinition is not possible within the PRIVATE SECTION.)

When you use the REDEFINITION addition, you must specify a (new) implementation part for the inherited method. As the signature may not be changed, you do not need to define the method parameters and exceptions again.

Within the redefined method's implementation part, you can use the predefined prefix **super->...** to access components in the superclass directly above where you are working. You often need to do this when redefining a method to call the original method of the superclass.

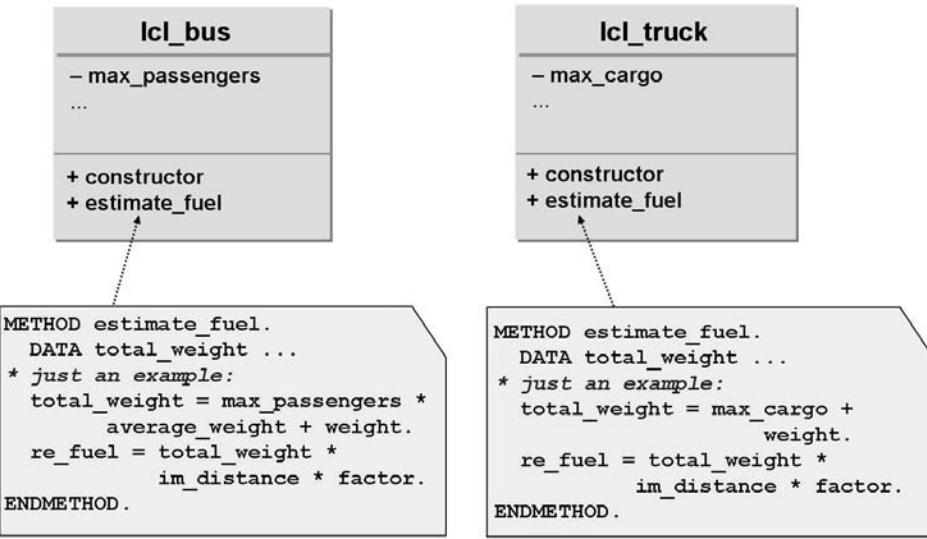


Figure 67: Preserving Semantics During Redefinition

In this example, both redefined methods calculate the return code in different ways. The important point is that the method's semantics stay the same.

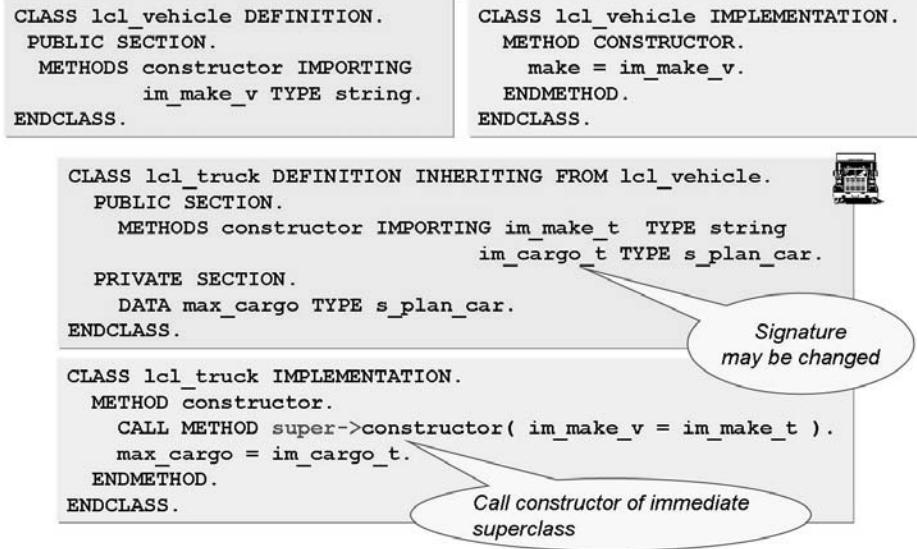


Figure 68: Definition of the constructor in subclasses

In most cases, a redefinition as described for the methods above would not be useful in the case of the constructor. Either the superclass's constructor can be used without any need to change it or the subclass has been expanded and other parameters are now

required in the constructor's signature. In ABAP Objects, the constructor can only be “overwritten” as part of inheritance. It can be overwritten in the sense that both the signature and the implementation part can be **adjusted** in the subclass.



Hint: In this context, the concept of **overloading** is of relevance. With overloading, a method has several definitions with different signatures and thus also different implementations. This is not supported in ABAP Objects.

The constructor of the direct superclass **must** be called within the constructor of the subclass. This is because of the specialization relationship: If a constructor was defined in the superclass, it contains implementations that will always be executed when an object is created in this superclass or its subclasses. However, this can only be automatically ensured by the runtime system if the subclass's constructor was not changed. In most cases, supplying consistent private attributes from the superclass is another prerequisite for calling the superclass constructor.

In contrast to instance constructors, the static constructor in the superclass is called automatically. This means that the runtime system automatically ensures that the static constructors of all its superclasses have already been executed before the static constructor in a particular class is executed.



- **Case 1:**
Class of instance to be created **has** constructor
⇒ supply its parameters

- **Case 2:**
Class of instance to be created
has inherited constructor
⇒ search in the inheritance tree for
the next highest superclass with
a constructor
⇒ supply its parameters

for example:

```
DATA: r_2 TYPE REF TO lcl_2,
      r_3 TYPE REF TO lcl_3.

* case 2:
CREATE OBJECT r_2 EXPORTING im_a1 = 100.

* case 1:
CREATE OBJECT r_3 EXPORTING im_a1 = 100
                  im_a2 = 1000.
```

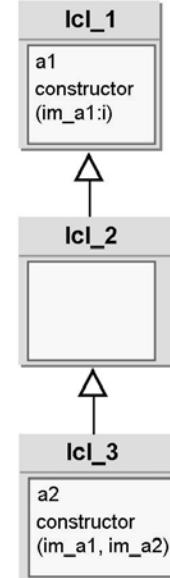


Figure 69: Rules for Calling the Constructor

If a subclass has not changed its instance constructor, the constructor is left unchanged and is adopted from the superclass. Therefore, the implementation is executed from the superclass.



- **Public components**
 - Visible to all
- **Protected components**
 - Visible within the class
 - Visible within all subclasses
- **Private components**
 - Only visible within the class

```
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION. ...

PROTECTED SECTION.
  DATA r_tank TYPE REF TO lcl_tank.

PRIVATE SECTION.
  DATA make TYPE string.

ENDCLASS.
```

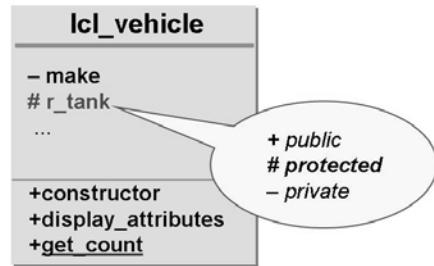


Figure 70: Inheritance and Visibility

Inheritance provides an extension of the visibility concept: There are protected components (PROTECTED SECTION). The visibility of these components lies between public and private. Protected components are visible to all subclasses and the class itself.

When defining local classes in ABAP Objects, you must follow the syntactical sequence of PUBLIC SECTION, PROTECTED SECTION, PRIVATE SECTION.

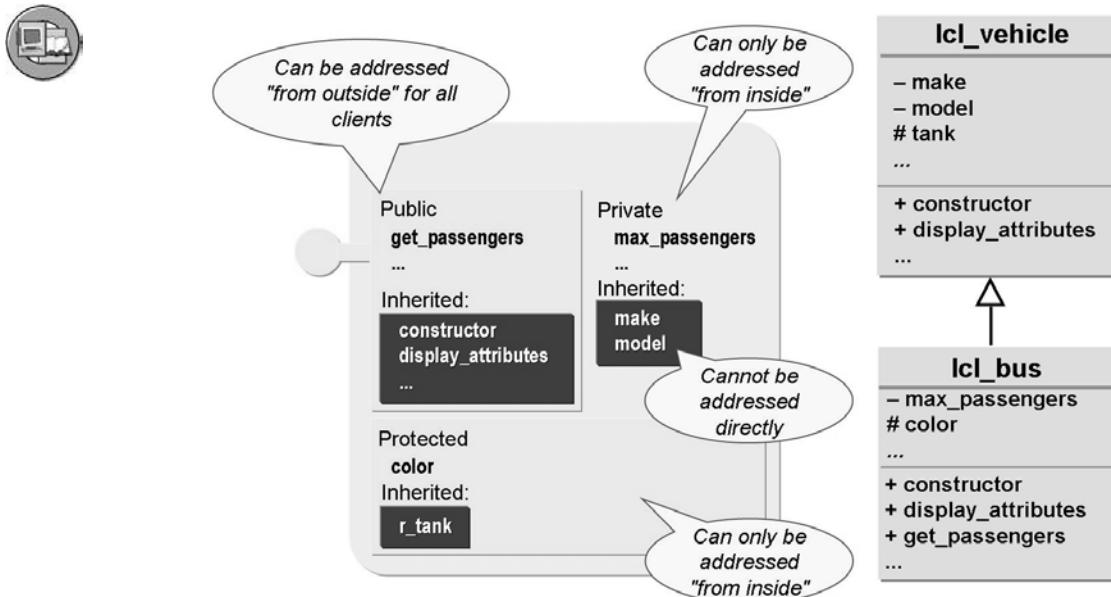


Figure 71: Protected Versus Private Section

The fact that all components of a superclass are available to the subclass is not related to the component's visibility. A subclass also receives the private components of its superclass. However, these cannot be addressed in the syntax of the subclass. Private components of superclasses can only be addressed indirectly using public or protected methods from the superclass, which, in turn, can access the private attributes. These restrictions are necessary to ensure that centralized maintenance is possible.

In this example, it is possible to access the protected attribute TANK in superclass LCL_VEHICLE directly from its subclass LCL_BUS. On the other hand, the only way to access the MAKE and MODEL attributes from the subclasses of LCL_VEHICLE is to use public methods.

Using the private visibility section, you can change superclasses without the need to know the subclasses. As long as the changes you make do not affect the semantics, you do not need to adapt the subclasses. This is because they only indirectly access the private components from the superclass.

There is only one static component per program context. To summarize:

Static Components and Inheritance

- A class that defines a public or protected static attribute shares this attribute with all its subclasses
- Static methods cannot be redefined



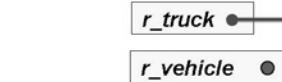
Up-Cast (Widening Cast)

Variables of the type “reference to superclass” can also refer to subclass instances at runtime.



```
DATA: r_vehicle TYPE REF TO lcl_vehicle,
      r_truck   TYPE REF TO lcl_truck.
```

```
CREATE OBJECT r_truck.
```



```
* Up-Cast (Widening-Cast):
r_vehicle = r_truck.
```



(5) lcl_truck

Public:

get_cargo

...

Inherited:

get_make

get_count

Redefined:

display_attributes

set_attributes

estimate_fuel

Figure 72: Up-Cast (Widening Cast) with object references

If you assign a subclass reference to a superclass reference, this ensures that all components that can be accessed syntactically after the cast assignment are actually available in the instance. The subclass always contains at least the same components as the superclass. After all, the name and the signature of redefined methods are identical.

User can therefore address only those methods and attributes from the subclass instance that they could from the superclass instance.



Hint: With redefined methods you should note that the implementation of the subclass is executed using the superclass static type of reference.

In this example, after the assignment, the methods GET_MAKE, GET_COUNT, DISPLAY_ATTRIBUTES, SET_ATTRIBUTES and ESTIMATE_FUEL of the instance LCL_TRUCK can only be accessed using the reference R_VEHICLE.

If there are any restrictions regarding visibility, they are left unchanged. It is not possible to access the specific components from the class LCL_TRUCK of the instance (GET_CARGO in the above example) using the reference R_VEHICLE. The view or possible access to methods is therefore usually narrowed (or at best left

unchanged). There is a switch from a view of several components to a view of a few components. As the target variable can accept more dynamic types in comparison to the source variable, this assignment is also called **widening cast**



- **The static type of a reference variable**

- Is defined using `TYPE REF TO`
- Remains the same throughout the program flow
- Defines which attributes and methods can be addressed

```
DATA r_vehicle
      TYPE REF TO lcl_vehicle.
```

- **The dynamic type of a reference variable**

- Is determined by the assignment
- Can change during the program run
- Defines which implementations are carried out for inherited methods

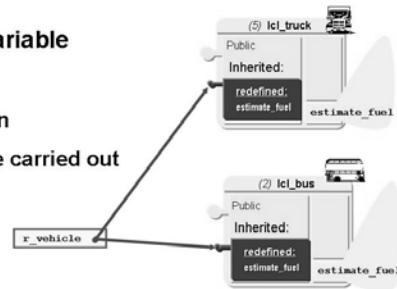


Figure 73: Static and Dynamic Types of References

A reference variable always has two types at runtime: static and dynamic.

In the example, LCL_VEHICLE is the static type of the variable R_VEHICLE. Depending on the cast assignment, the dynamic type is either LCL_BUS or LCL_TRUCK. In the *ABAP Debugger*, the dynamic type is specified in the form of the following object display.

```
object_id<\PROGRAM=program_name\CLASS=dynamic_type>
```

→ **Note:** Assignments between reference variables are possible whenever the static type of the target variables is more general or equal to the dynamic type of the source variables.

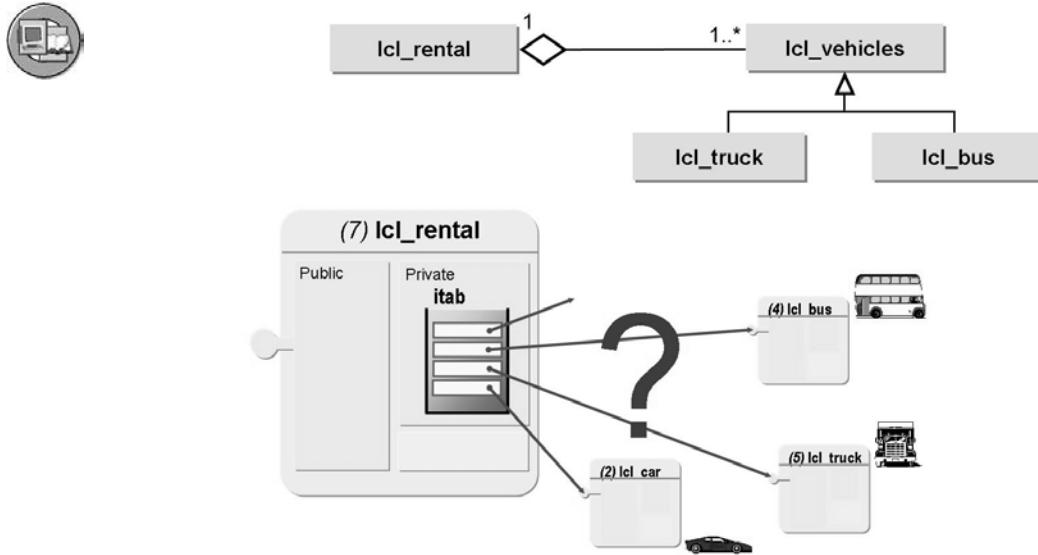


Figure 74: Generic Access After Up-Cast Assignments

A typical use for up-cast assignments is to prepare for generic access: A user who is not at all interested in the finer points of the instances of the subclasses but who simply wants to address the shared components, could use a superclass reference for this access.

In the example shown here, a travel agency (LCL_RENTAL) needs to manage all imaginable kinds of vehicles in one list. This leads to the question of what type should be assigned to the internal table for the references to airplane instances. You should also assume that the car rental company needs to be able to calculate only the required amount of fuel for all its vehicles. Correspondingly, the ESTIMATE_FUEL method is defined in the superclass LCL_VEHICLE and is redefined in all subclasses.

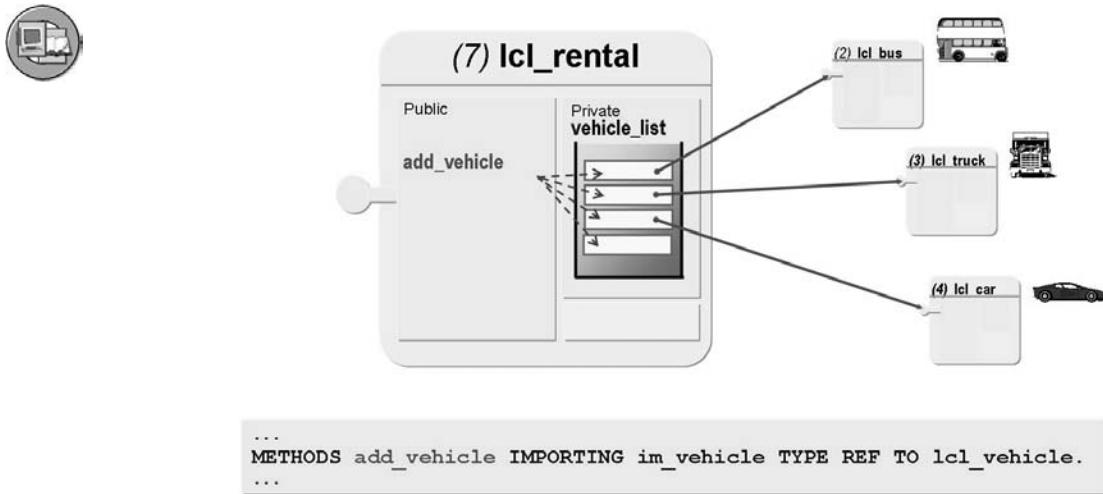


Figure 75: Row Type of the Internal Table in the Application Example

When objects of different classes (LCL_BUS, LCL_TRUCK, and LCL_CAR in the above example) are specified as type superclass references (LCL_VEHICLE in the above example), they can be stored in an internal table. The shared components of the subclass objects can then be accessed uniformly. For this example, the method ADD_VEHICLE is therefore needed. This copies the references to the vehicle types in this internal table. Its import parameter is already typed as the reference to the superclass.

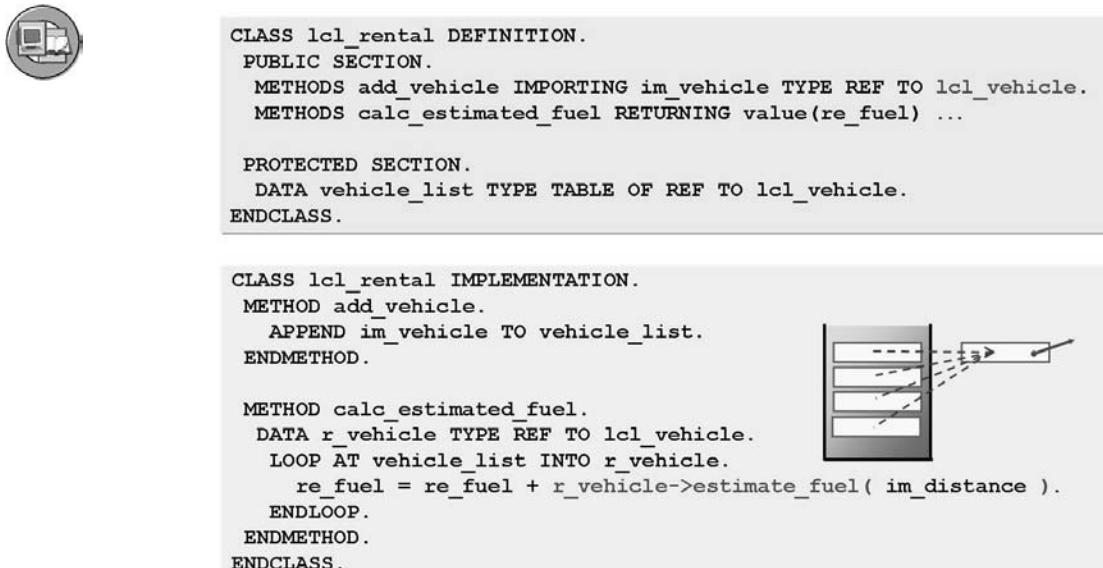


Figure 76: Up-Cast and Generic Access in the Application Example

In this example, the up-cast assignment occurs when the vehicle reference is transferred to the formal parameter of the ADD_VEHICLE method. The shared component is generically accessed within the loop around the internal table containing all of the vehicle references: The ESTIMATE_FUEL method was inherited from the LCL_VEHICLE superclass and may have been redefined.



```
METHOD calc_estimated_fuel.
DATA r_vehicle TYPE REF TO lcl_vehicle.
LOOP AT vehicle_list INTO r_vehicle.
  re_fuel = re_fuel + r_vehicle->estimate_fuel( im_distance ).
ENDLOOP.
ENDMETHOD.
```

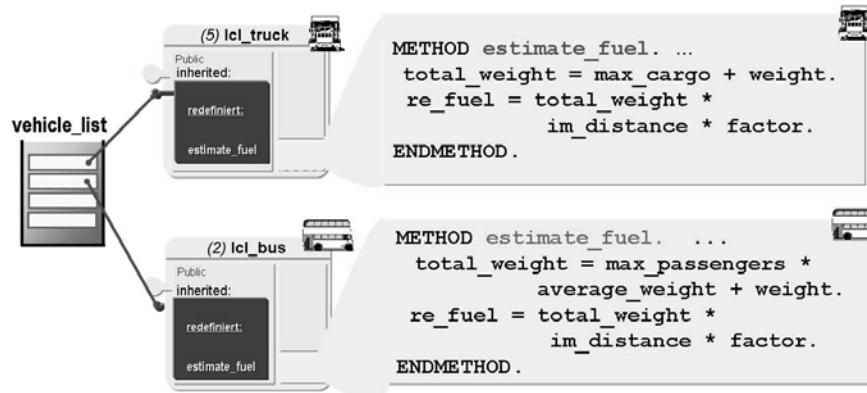


Figure 77: Polymorphism: Generic Access Using the Superclass Reference

Which implementation is executed when ESTIMATE_FUEL is called now depends on which object the superclass reference R_VEHICLE currently refers to. The dynamic type and not the static type of the reference variable is used to search for the implementation of a method. Therefore, when `r_vehicle->estimate_fuel` is called, the implementation is not executed from LCL_VEHICLE (static type of R_VEHICLE) because the method was redefined in all vehicle classes.

When an instance receives a message to execute a particular method, the method that implemented the class of this instance is executed. If the class has not been redefined in the method, the implementation from the superclass is executed.

When objects from different classes react differently to the same method calls, this is known as **polymorphism**. The possibility of polymorphism is one of the main strengths of inheritance: A client can handle different classes uniformly, irrespective of their implementation. The runtime system searches for the right implementation of a method on behalf of the client.

Polymorphism can be used to write programs that are highly generic, that is, they do not need to be changed significantly if use cases are added. For instance, this would make it very easy to add motorbikes to this example: You would simply have to define a new subclass of LCL_VEHICLE, which you could call LCL_MOTORBIKE. You would also have to redefine the inherited method ESTIMATE_FUEL. Without needing any more changes, your vehicle management system could then work with motorbikes as well and calculate the required fuel levels for them.



```

FUNCTION-POOL s_vehicle.
...
FUNCTION estimate_fuel_truck.
...
ENDFUNCTION.

FUNCTION estimate_fuel_bus.
...
ENDFUNCTION.
...

DATA: vehicle_list TYPE TABLE OF vehicle_type,
      vehicle      LIKE LINE OF vehicle_list,
      gd_fuel      TYPE ....
      .
      .
      .
LOOP AT vehicle_list INTO vehicle.
  CALL FUNCTION vehicle-func_name
    EXPORTING im_dist = distance
    IMPORTING ex_fuel = fuel.
  gd_fuel = gd_fuel + fuel.
ENDLOOP.
...

```

vehicle_list	
name	func_name
truck1	ESTIMATE_FUEL_TRUCK
bus1	ESTIMATE_FUEL_BUS
bus2	ESTIMATE_FUEL_BUS
truck2	ESTIMATE_FUEL_TRUCK
truck3	ESTIMATE_FUEL_TRUCK

Figure 78: Generic Calls in the Procedural Programming Model

Using dynamic function modules, you can program generically in ABAP Objects, even without an object-oriented programming model. Compared with polymorphism through inheritance, this means that the source code is less self-explanatory and is more susceptible to errors. For example, the syntax check can only check that the function module is called correctly but not whether the internal table contains a valid function module name for **each vehicle**.

Down-cast (Narrowing Cast)

Variables of the type “reference to superclass” can also refer to subclass instances at runtime. You may now want to copy such a reference (back) to a suitable variable of the type “reference to subclass”.

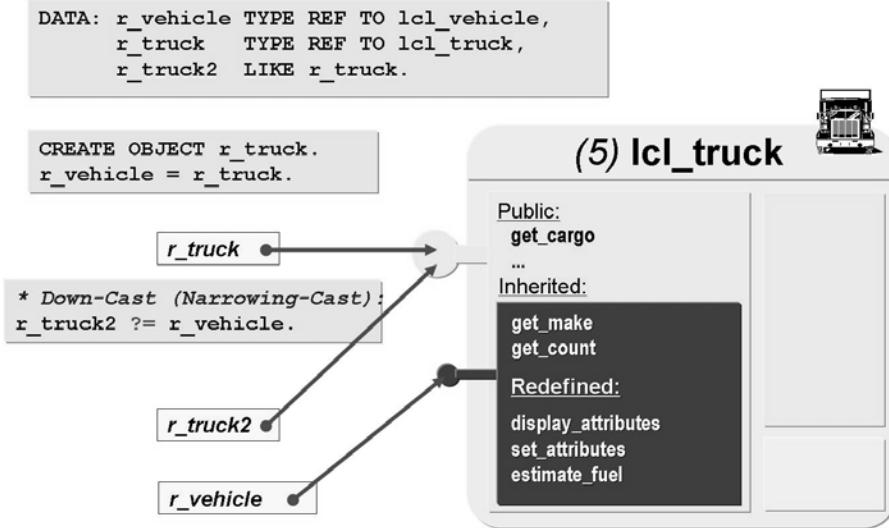


Figure 79: Down Cast (Narrowing Cast) with Object References

If you want to assign a superclass reference to a subclass reference, you must use the down cast assignment operator **MOVE ... ?TO ...** or its short form **?=**. Otherwise, you would get a message stating that it is not certain that all components that can be accessed syntactically after the cast assignment are actually available in the instance. As a rule, the subclass class contains more components than the superclass.

After assigning this type of reference (back) to a subclass reference to the implementing class, clients are no longer limited to inherited components: In the example given here, all components of the LCL_TRUCK instance can be accessed (again) after the assignment using the reference R_TRUCK2.

The view is thus usually widened (or at least unchanged). This type of assignment of reference variables is known as **down cast**. There is a switch from a view of a few components to a view of more components. As the target variable can accept less dynamic types after the assignment, this assignment is also called **narrowing cast**.

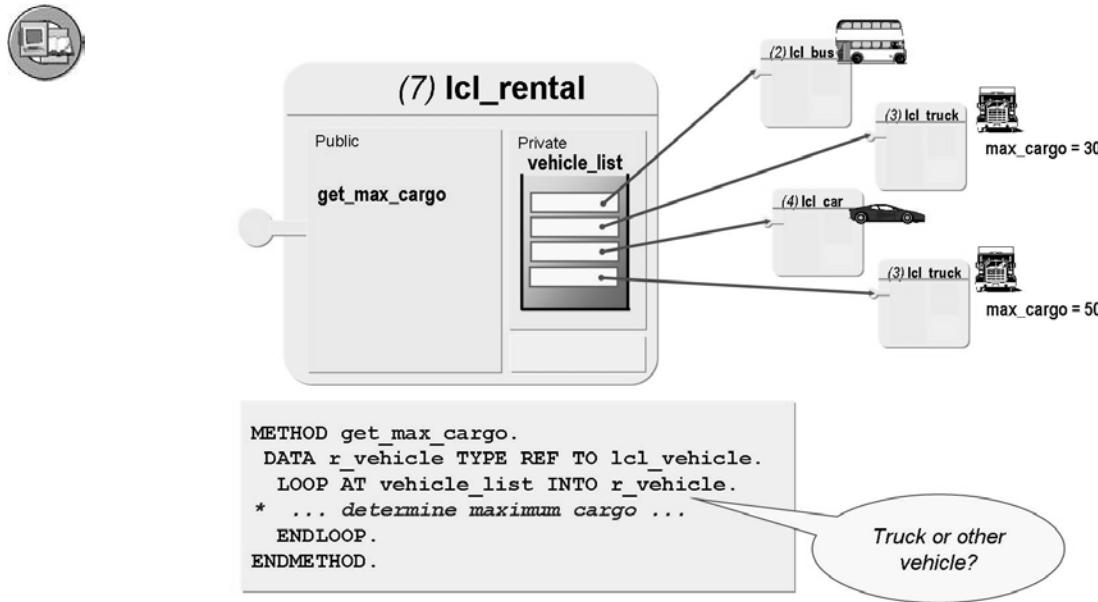


Figure 80: Specific Access After Down Cast Assignments

A typical use for down cast assignments is when specific components of instances need to be addressed and their references are kept in variables that are typed on the superclass. A user who is interested in the finer points of the instances of a subclass cannot use the superclass reference for this access because it only allows access to the shared components.

In this example, a car rental company (LCL_RENTAL) wants to determine the maximum capacity of its trucks, but it stores all types of vehicle references in an internal table that is typed as LCL_VEHICLE. Therefore, what happens if there is no truck reference in the superclass reference R_VEHICLE at runtime but the down cast assignment operator tries to copy the reference to the now invalid reference R_TRUCK?

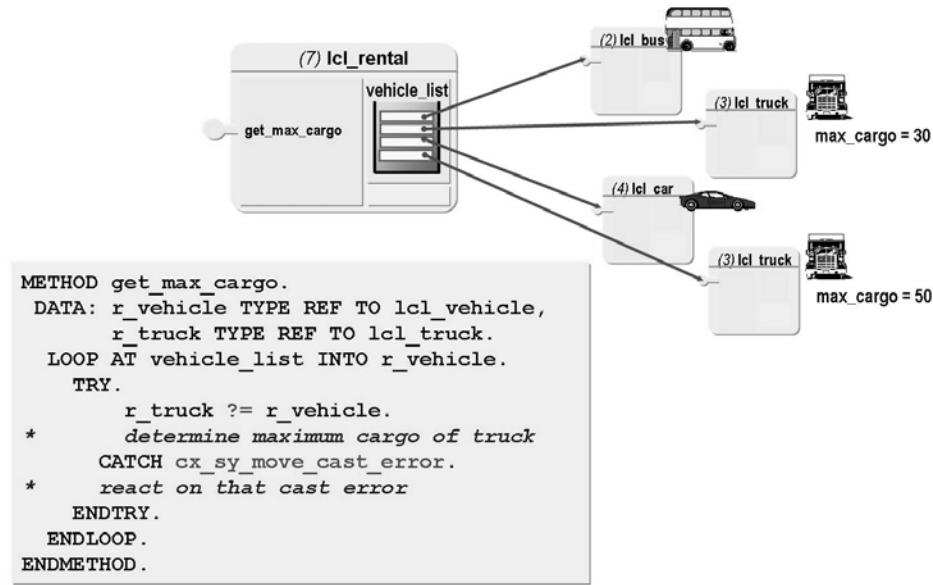


Figure 81: Down Cast and Exception Handling in the Application Example

In contrast to the up cast assignment, it is possible that the static type of the target variable (R_TRUCK) is neither more general than nor the same as the dynamic type of the source variable (R_VEHICLE), namely when R_VEHICLE contains bus or sports car references. That is why with this kind of cast, the runtime system checks, before the assignment, whether the current content of the source variable corresponds to the type requirements of the target variable. Otherwise, an exception that can be handled is triggered, and the original value of the target variable remains the same.

This exception of the error class CX_SY_MOVE_CAST_ERROR can be identified using TRY-ENDTRY and the CATCH statement. Another way of preventing this runtime error would be to use runtime type identification (RTTI) classes. They can be used to determine the dynamic type at runtime and to set a condition for the cast.

Input of Class Hierarchies

This final section does not examine **whether or not** inheritance should be used as a programming technique over a number of alternatives. As early as the modeling phase, you should be able to see whether there is a generalization/specialization relationship between certain classes. If there is, you can use inheritance to represent this in ABAP Objects.



If there is a suitable way to link classes in terms of inheritance, this results in the following advantages:



- Centralized maintenance
- Safe, generic method of access

All in all, this means that the entire software component can be extended very easily.

Figure 82: Input of Class Hierarchies

If this is the case, you need to adhere to the relevant concepts of inheritance. For example, the semantics must be preserved when you redefine methods. Furthermore, inherited components must be used as intended in the superclass.



Subclasses instead of another attribute

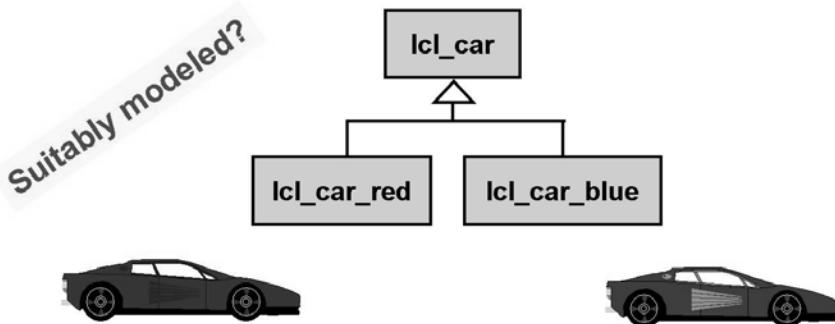


Figure 83: Examples: Misuse of Inheritance

If you do not have a correct understanding of the formulation “is a (specific)”, you run the risk of identifying the wrong places in which to use inheritance. Sometimes the need for another attribute for a class is incorrectly answered with a specialization.

Example: A superclass called Car contains the subclasses Red Car, Blue Car, and so on. The statement “A red car is a specific car” is only correct at first glance. However, there are no cars without a color. At most, there are some cars that have not been painted. Therefore, every car needs the attribute Color, assuming that it is relevant to the application. Therefore, the attribute should already have been defined in the

superclass, or there is no longer an authorization for subclasses of this type. There would also be contradictions with reality when you try to implement a method for painting the cars.

- **Note:** Such attributes are often also defined as reference variables to a superclass of role classes. There is one description class for each role. To change the role of an instance, you then exchange the references to the corresponding role description instances. Specialist literature also refers to this as a “role design pattern”.

In some cases, specialization relationships are identified that do not allow for the semantics to be preserved.

Example: The class Square inherits from the class Rectangle. If you try to define methods for the rectangle that change the width and height separately, these methods would not make sense when applied to the square. Even if the methods were redefined to make the lengths of the sides uniform, the semantics would be different.

For the same reasons, simple inheritance of the source text – that is, using inheritance only because some required functions were found in a (super)class – is also impossible.

Exercise 7: Class Hierarchies

Exercise Objectives

After completing this exercise, you will be able to:

- Define subclasses
- Redefine superclass methods in subclasses
- Effectively use visibility sections

Business Example

Your airplane management program is to be further refined. Put specific airplanes in relation to their general airplane class.

Task 1:

In the class LCL_AIRPLANE, define the local subclass LCL_PASSENGER_PLANE for passenger planes.

1. Complete your program ZBC401_##_MAIN (where ## is your two-digit group number) or copy the program SAPBC401_CASS_MAIN_E.)
If applicable, enter the source code for your new class in the include program.
2. The class must have a private instance attribute, MAX_SEATS, with the same type as the table field SFLIGHT-SEATSMAX.
3. Define and implement an instance constructor that assigns values to **all** instance attributes in the class.
4. Redefine DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.

Task 2:

In the class LCL_AIRPLANE, define the local subclass LCL_CARGO_PLANE for cargo planes.

1. The class must have a private instance attribute, MAX_CARGO, with the same type as the table field SCPLANE-CARGOMAX.
2. Define and implement an instance constructor that assigns values to **all** instance attributes in the class.

Continued on next page

3. Redefine DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.
4. Define the functional method GET_CARGO to return the cargo value to the calling program. Use the formula parameter RE_CARGO.

Task 3:

Create instances of your new classes and display their attributes.

1. In the main program, define a suitably typed reference variable for each of your new classes.
2. Call the static method DISPLAY_N_O_AIRPLANES (before instantiating any objects).
3. Use the two references to create an instance of each of the subclasses LCL_PASSENGER_PLANE and LCL_CARGO_PLANE. Decide for yourself how to fill the attributes.
4. Call the DISPLAY_ATTRIBUTES method for both instances.
5. Call the static method DISPLAY_ATTRIBUTES a second time.

Task 4:

Analyze your program.

1. Observe the program flow in the *ABAP Debugger*, paying special attention to the call of the DISPLAY_ATTRIBUTES method.
 2. Could the method GET_TECHNICAL_ATTRIBUTES be called directly from the **redefined** method DISPLAY_ATTRIBUTES of the subclasses?
-
-
-

Solution 7: Class Hierarchies

Task 1:

In the class LCL_AIRPLANE, define the local subclass LCL_PASSENGER_PLANE for passenger planes.

1. Complete your program ZBC401_##_MAIN (where ## is your two-digit group number) or copy the program SAPBC401_CASS_MAIN_E.)
If applicable, enter the source code for your new class in the include program.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
 - b) Model solution: SAPBC401_INHS_MAIN_A
2. The class must have a private instance attribute, MAX_SEATS, with the same type as the table field SFLIGHT-SEATSMAX.
 - a) Refer to the model solution.
3. Define and implement an instance constructor that assigns values to **all** instance attributes in the class.
 - a) Refer to the model solution.
4. Redefine DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.
 - a) Refer to the model solution.

Task 2:

In the class LCL_AIRPLANE, define the local subclass LCL_CARGO_PLANE for cargo planes.

1. The class must have a private instance attribute, MAX_CARGO, with the same type as the table field SCPLANE-CARGOMAX.
 - a) Refer to the model solution.
2. Define and implement an instance constructor that assigns values to **all** instance attributes in the class.
 - a) Refer to the model solution.

Continued on next page

3. Redefine DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.
 - a) Refer to the model solution.
4. Define the functional method GET_CARGO to return the cargo value to the calling program. Use the formula parameter RE_CARGO.
 - a) Refer to the model solution.

Task 3:

Create instances of your new classes and display their attributes.

1. In the main program, define a suitably typed reference variable for each of your new classes.
 - a) Refer to the model solution.
2. Call the static method DISPLAY_N_O_AIRPLANES (before instantiating any objects).
 - a) Refer to the model solution.
3. Use the two references to create an instance of each of the subclasses LCL_PASSENGER_PLANE and LCL_CARGO_PLANE. Decide for yourself how to fill the attributes.
 - a) Refer to the model solution.
4. Call the DISPLAY_ATTRIBUTES method for both instances.
 - a) Refer to the model solution.
5. Call the static method DISPLAY_ATTRIBUTES a second time.
 - a) Refer to the model solution.

Task 4:

Analyze your program.

1. Observe the program flow in the *ABAP Debugger*, paying special attention to the call of the DISPLAY_ATTRIBUTES method.
 - a) Carry out this step in the usual manner.
2. Could the method GET_TECHNICAL_ATTRIBUTES be called directly from the **redefined** method DISPLAY_ATTRIBUTES of the subclasses?

Answer: No, because the component of the superclass is private.

Continued on next page

Result

Model solution:

SAPBC401_INHS_MAIN_A

```

REPORT  sapbc401_inhs_main_a.

TYPE-POOLS icon.

INCLUDE sapbc401_inhs_a.

DATA: r_plane TYPE REF TO lcl_airplane.
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_passenger TYPE REF TO lcl_passenger_plane,
      plane_list TYPE TABLE OF REF TO lcl_airplane.

START-OF-SELECTION.
*#####
lcl_airplane=>display_n_o_airplanes( ).

CREATE OBJECT r_passenger EXPORTING
      im_name = 'LH BERLIN'
      im_planetype = '747-400'
      im_seats = 345.

CREATE OBJECT r_cargo EXPORTING
      im_name = 'US Hercules'
      im_planetype = '747-500'
      im_cargo = 533.

r_cargo->display_attributes( ).

r_passenger->display_attributes( ).

lcl_airplane=>display_n_o_airplanes( ).
```

Continued on next page

SAPBC401_INHS_A

```

*&-----*
*&   Include          SAPBC401_INHS_A           *
*&-----*
*-----*
*      CLASS lcl_airplane DEFINITION           *
*-----*
CLASS lcl_airplane DEFINITION.

PUBLIC SECTION.
"-----
CONSTANTS: pos_1 TYPE i VALUE 30.

METHODS: constructor IMPORTING
         im_name      TYPE string
         im_planetype TYPE saplane-planetype,
         display_attributes.

CLASS-METHODS: display_n_o_airplanes.

PRIVATE SECTION.
"-----
METHODS: get_technical_attributes
         IMPORTING im_type      TYPE saplane-planetype
         EXPORTING ex_weight    TYPE s_plan_wei
                     ex_tankcap  TYPE s_capacity

DATA: name      TYPE string,
      planetype TYPE saplane-planetype.

CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.          "lcl_airplane DEFINITION

*-----*
*      CLASS lcl_airplane IMPLEMENTATION        *
*-----*
CLASS lcl_airplane IMPLEMENTATION.

METHOD constructor.
  name      = im_name.

```

Continued on next page

```

planetype      = im_planetype.
n_o_airplanes = n_o_airplanes + 1.
ENDMETHOD.           "constructor

METHOD display_attributes.
DATA: weight TYPE saplane-weight,
      cap TYPE saplane-tankcap.
WRITE: / icon_ws_plane AS ICON,
      / 'Name of airplane'(001), AT pos_1 name,
      / 'Airplane type: '(002), AT pos_1 planetype.
get_technical_attributes( EXPORTING im_type = planetype
                           IMPORTING ex_weight = weight
                           ex_tankcap = cap ).
WRITE: / 'Weight:'(003), weight,
      'Tank capacity:'(004), cap.
ENDMETHOD.           "display_attributes

METHOD display_n_o_airplanes.
WRITE: /, / 'Number of airplanes: '(ca1),
      AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
ENDMETHOD.           "display_n_o_airplanes

METHOD get_technical_attributes.
SELECT SINGLE weight tankcap FROM saplane
      INTO (ex_weight, ex_tankcap)
      WHERE planetype = im_type.
IF sy-subrc <> 0.
  ex_weight = 100000.
  ex_tankcap = 10000.
ENDIF.

ENDMETHOD.           "get_technical_attributes

ENDCLASS.           "lcl_airplane IMPLEMENTATION

*-----*
*     CLASS lcl_cargo_plane DEFINITION
*-----*
*
*-----*
CLASS lcl_cargo_plane DEFINITION INHERITING FROM lcl_airplane.

```

Continued on next page

```

PUBLIC SECTION.

"-----
METHODS: constructor IMPORTING im_name TYPE string
         im_planetype TYPE saplane-planetype
         im_cargo TYPE scplane-cargomax.

METHODS: display_attributes REDEFINITION.

METHODS: get_cargo RETURNING VALUE(re_cargo) TYPE scplane-cargomax.

PRIVATE SECTION.

"-----
DATA: max_cargo TYPE scplane-cargomax.

ENDCLASS.                      "lcl_cargo_plane DEFINITION

*-----*
*      CLASS lcl_cargo_plane IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_cargo_plane IMPLEMENTATION.

METHOD constructor.

CALL METHOD super->constructor( im_name = im_name
                                 im_planetype = im_planetype ).
max_cargo = im_cargo.

ENDMETHOD.                      "constructor

METHOD display_attributes.

super->display_attributes( ).

WRITE: / 'Max Cargo = ', max_cargo.
ULINE.

ENDMETHOD.                      "display_attributes

METHOD get_cargo.

re_cargo = max_cargo.

ENDMETHOD.                      "get_cargo

ENDCLASS.                      "lcl_cargo_plane IMPLEMENTATION

```

Continued on next page

```

*      CLASS lcl_passenger_plane DEFINITION
*-----*
*
*-----*
CLASS lcl_passenger_plane DEFINITION INHERITING FROM lcl_airplane..

PUBLIC SECTION.

METHODS: constructor IMPORTING im_name TYPE string
          im_planetype TYPE saplane-planetype
          im_seats TYPE sflight-seatsmax.

METHODS: display_attributes REDEFINITION.

PRIVATE SECTION.

DATA: max_seats TYPE sflight-seatsmax.

ENDCLASS.                      "lcl_passenger_plane DEFINITION

*-----*
*      CLASS lcl_passenger_plane IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_passenger_plane IMPLEMENTATION.

METHOD constructor.
  CALL METHOD super->constructor( EXPORTING im_name      = im_name
                                    im_planetype = im_planetype ).
  max_seats = im_seats.
ENDMETHOD.                      "constructor

METHOD display_attributes.
  super->display_attributes( ).
  WRITE: / 'Max Seats = ', max_seats.
  ULINE.
ENDMETHOD.                      "display_attributes

ENDCLASS.                      "lcl_passenger_plane IMPLEMENTATION

```


Exercise 8: Polymorphism

Exercise Objectives

After completing this exercise, you will be able to:

- Program Casting Assignments
- Use inheritance relationships for polymorphic method calls

Business Example

Your airplane management program should display the attributes of the airplane objects generically, that is, it should be open to future extensions with additional airplane classes.

Task 1:

Buffer the airplane references in a suitable type of internal table.

1. Complete your program ZBC401_##_MAIN or copy the sample solution from the previous exercise. (## is your two-digit group number.)
2. In your main program, define an internal table for buffering airplane references if you do not already have one. The row type of the internal table should be REF TO LCL_AIRPLANE.

Task 2:

Display the attributes of **all** airplane types that were created so far.

1. Insert the references to your passenger and cargo airplanes into the internal table.
2. Program a loop through the contents of the internal table. Call the DISPLAY_ATTRIBUTES method every time the loop runs.

Task 3:

Analyze your program.

1. Follow the program flow in the *ABAP Debugger*, paying special attention to the call of the method DISPLAY_ATTRIBUTES.

Continued on next page

2. What would happen if the method DISPLAY_ATTRIBUTES had not been redefined in the subclasses?

Solution 8: Polymorphism

Task 1:

Buffer the airplane references in a suitable type of internal table.

1. Complete your program ZBC401_##_MAIN or copy the sample solution from the previous exercise. (## is your two-digit group number.)
 - a) Carry out this step as usual. Additional information is available in the SAP Library.
 - b) Model solution: SAPBC401_CASS_MAIN_A
2. In your main program, define an internal table for buffering airplane references if you do not already have one. The row type of the internal table should be REF TO LCL_AIRPLANE.
 - a) See the source code extract from the model solution.

Task 2:

Display the attributes of **all** airplane types that were created so far.

1. Insert the references to your passenger and cargo airplanes into the internal table.
 - a) See the source code extract from the model solution.
2. Program a loop through the contents of the internal table. Call the DISPLAY_ATTRIBUTES method every time the loop runs.
 - a) See the source code extract from the model solution.

Task 3:

Analyze your program.

1. Follow the program flow in the *ABAP Debugger*, paying special attention to the call of the method DISPLAY_ATTRIBUTES.
 - a) Carry out this step as usual. Additional information is available in the SAP Library.
2. What would happen if the method DISPLAY_ATTRIBUTES had not been redefined in the subclasses?

Answer: The implementation from the superclass would be executed. Your program would **not** contain polymorphism.

Continued on next page

Result

Source code:

SAPBC401_CASS_MAIN_A

```
REPORT  sapbc401_cass_main_a.

TYPE-POOLS  icon.

INCLUDE  sapbc401_inhs_a.

DATA: r_plane TYPE REF TO lcl_airplane,
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_passenger TYPE REF TO lcl_passenger_plane,
      plane_list TYPE TABLE OF REF TO lcl_airplane.

START-OF-SELECTION.
*#####
lcl_airplane=>display_n_o_airplanes( ).

CREATE OBJECT r_passenger EXPORTING
        im_name = 'LH BERLIN'
        im_planetype = '747-400'
        im_seats = 345.
APPEND r_passenger TO plane_list.

CREATE OBJECT r_cargo EXPORTING
        im_name = 'US HErcules'
        im_planetype = '747-500'
        im_cargo = 533.
APPEND r_cargo TO plane_list.

LOOP AT plane_list INTO r_plane.
  r_plane->display_attributes( ).
ENDLOOP.

lcl_airplane=>display_n_o_airplanes( ).
```

Exercise 9: Aggregation and Generic Calls

Exercise Objectives

After completing this exercise, you will be able to:

- Implement aggregation relationships between classes
- Program Casting Assignments
- Use inheritance relationships for polymorphic method calls

Business Example

Now, the management of airplane instances should no longer take place in the main program. Instead, it should be encapsulated in a new class for airlines.

Task 1:

Define a local class for airlines.

1. From the **include** program SAPBC401_CAST_B, copy the portions of the source text for the local class LCL_CARRIER into your include program ZBC401_##_AIRPLANE so that you can add to them **there**. (## is your two-digit group number.))
2. Extend the signature and implementation of the method ADD_AIRPLANE so that the airplane references can be added to the previously defined list AIRPLANE_LIST.
3. Extend the implementation of the method DISPLAY_AIRPLANES so that the attributes of all airplanes of the airline can be added to the list. Each time an airplane is added, its DISPLAY_ATTRIBUTES method should be called.
4. Extend the implementation of the method DISPLAY_ATTRIBUTES so that all of the attributes of the airline can be displayed and, consequently, the contents of the airplane list can also be displayed.

Task 2:

In the main program, create an airline instance. Transfer some airplane references to it and display the attributes.

1. Remove all the statements from the main program that define the global internal table for airplane references and their insertions.

Continued on next page

2. In the main program, define a suitably typed reference variable for your new airline class.
3. Using the reference, generate an instance of your class LCL_CARRIER. Decide for yourself how to fill the attributes.
4. Call the method ADD_AIRPLANE to transfer the airplane instances that have been created so far to the airline. You may also create and transfer additional airplanes.
5. Display the attributes of the airline by calling its method DISPLAY_ATTRIBUTES.

Task 3:

Optional exercise: Determining the highest cargo value.

1. Define and implement the functional method GET_HIGHEST_CARGO in class LCL_CARRIER to calculate the highest cargo value (load capacity) of all cargo planes. Use the down-cast technique to identify all relevant airplanes for this task.
2. For testing purposes, this new method can be called within DISPLAY_ATTRIBUTES of class LCL_CARRIER.

Solution 9: Aggregation and Generic Calls

Task 1:

Define a local class for airlines.

1. From the **include** program SAPBC401_CAST_B, copy the portions of the source text for the local class LCL_CARRIER into your include program ZBC401_##_AIRPLANE so that you can add to them **there**. (## is your two-digit group number.)
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
 - b) Model solution: SAPBC401_CASS_MAIN_B
2. Extend the signature and implementation of the method ADD_AIRPLANE so that the airplane references can be added to the previously defined list AIRPLANE_LIST.
 - a) See the source code extract from the model solution.
3. Extend the implementation of the method DISPLAY_AIRPLANES so that the attributes of all airplanes of the airline can be added to the list. Each time an airplane is added, its DISPLAY_ATTRIBUTES method should be called.
 - a) See the source code extract from the model solution.
4. Extend the implementation of the method DISPLAY_ATTRIBUTES so that all of the attributes of the airline can be displayed and, consequently, the contents of the airplane list can also be displayed.
 - a) See the source code extract from the model solution.

Task 2:

In the main program, create an airline instance. Transfer some airplane references to it and display the attributes.

1. Remove all the statements from the main program that define the global internal table for airplane references and their insertions.
 - a) See the source code extract from the model solution.
2. In the main program, define a suitably typed reference variable for your new airline class.
 - a) See the source code extract from the model solution.

Continued on next page

3. Using the reference, generate an instance of your class LCL_CARRIER. Decide for yourself how to fill the attributes.
 - a) See the source code extract from the model solution.
4. Call the method ADD_AIRPLANE to transfer the airplane instances that have been created so far to the airline. You may also create and transfer additional airplanes.
 - a) See the source code extract from the model solution.
5. Display the attributes of the airline by calling its method DISPLAY_ATTRIBUTES.
 - a) See the source code extract from the model solution.

Task 3:

Optional exercise: Determining the highest cargo value.

1. Define and implement the functional method GET_HIGHEST_CARGO in class LCL_CARRIER to calculate the highest cargo value (load capacity) of all cargo planes. Use the down-cast technique to identify all relevant airplanes for this task.
 - a) Refer to the model solution.
2. For testing purposes, this new method can be called within DISPLAY_ATTRIBUTES of class LCL_CARRIER.
 - a) Refer to the model solution.

Result

Source code extract:

SAPBC401_CASS_MAIN_C

```
REPORT  sapbc401_cass_main_c.

TYPE-POOLS  icon.

INCLUDE  sapbc401_cass_c.

DATA: r_plane TYPE REF TO lcl_airplane,
      r_cargo TYPE REF TO lcl_cargo_plane,
```

Continued on next page

```

r_passenger TYPE REF TO lcl_passenger_plane,
r_carrier TYPE REF TO lcl_carrier.

START-OF-SELECTION.
*#####
***** Create Carrier *****
CREATE OBJECT r_carrier EXPORTING im_name = 'Smile&Fly Travel'.

***** Passenger Plane *****
CREATE OBJECT r_passenger EXPORTING
      im_name = 'LH BERLIN'
      im_planetype = '747-400'
      im_seats = 345.

***** cargo Plane *****
CREATE OBJECT r_cargo EXPORTING
      im_name = 'US HERCULES'
      im_planetype = '747-500'
      im_cargo = 533.

***** insert planes into itab if client *****
r_carrier->add_airplane( r_passenger ).

r_carrier->add_airplane( r_cargo ).

***** show all airplanes inside carrier *****
r_carrier->display_attributes( ).
```

SAPBC401_CASS_C

```

*&-----*
*&   Include          SAPBC401_CASS_C
*&-----*

*-----*
*   CLASS lcl_airplane DEFINITION
*-----*

...
*-----*
*   CLASS lcl_carrier DEFINITION
*-----*
```

Continued on next page

```

*-----*
*
*-----*
CLASS lcl_carrier DEFINITION.

PUBLIC SECTION.
"-----
METHODS: constructor IMPORTING im_name TYPE string,
          get_name RETURNING value(ex_name) TYPE string,
          add_airplane IMPORTING
                  im_plane TYPE REF TO lcl_airplane,
          display_airplanes,
          display_attributes,
get_highest_cargo RETURNING value(re_cargo)
                  TYPE s_plan_car.

PRIVATE SECTION.
"-----
DATA: name           TYPE string,
      airplane_list TYPE TABLE OF REF TO lcl_airplane.
ENDCLASS.           "lcl_carrier DEFINITION

*-----*
*      CLASS lcl_carrier IMPLEMENTATION
*-----*
CLASS lcl_carrier IMPLEMENTATION.

METHOD add_airplane.
  APPEND im_plane TO airplane_list.
ENDMETHOD.           "add_airplane

METHOD display_attributes.
  DATA: highest_cargo TYPE s_plan_car.
  WRITE: icon_flight AS ICON, name . ULINE. ULINE.
         display_airplanes( ).

  highest_cargo = ME->get_highest_cargo( ).
  WRITE: / ' Highest Cargo = ', highest_cargo.
ENDMETHOD.           "display_attributes

METHOD display_airplanes.
  DATA: r_plane TYPE REF TO lcl_airplane.

```

Continued on next page

```

LOOP AT airplane_list INTO r_plane.
  r_plane->display_attributes( ).
ENDLOOP.

ENDMETHOD.                               "display_airplanes

METHOD constructor.
  name = im_name.
ENDMETHOD.                               "constructor

METHOD get_name.
  ex_name = name.
ENDMETHOD.                               "get_name

METHOD get_highest_cargo.
  DATA: r_cargo TYPE REF TO lcl_cargo_plane.
  DATA: r_plane TYPE REF TO lcl_airplane.
  DATA: cargo TYPE s_plan_car.
  DATA: r_exc TYPE REF TO cx_root.

  LOOP AT airplane_list INTO r_plane.
    TRY.
      ***** her comes the optimistic coding *****
      r_cargo ?= r_plane.
      cargo = r_cargo->get_cargo( ).

      IF re_cargo < cargo.
        re_cargo = cargo.
      ENDIF.
      CATCH cx_sy_move_cast_error INTO r_exc. **** no cargoplane
    ENDTRY.
  ENDLOOP.

ENDMETHOD.

ENDCLASS.                                "lcl_carrier IMPLEMENTATION

```



Lesson Summary

You should now be able to:

- Define inheritance relationships between classes
- Redefine methods
- Create up-cast assignments (Widening Cast)
- Create down-cast assignments (Narrowing Cast)
- Explain the concept of polymorphism with reference to inheritance
- Use cast assignments with inheritance to make generic calls

Related Information

For more information about this subject, refer to the SAP Library and the ABAP keyword documentation for the individual statements.

Lesson: Interfaces and Casting

Lesson Overview

The only real difference between interfaces and inheritance is the role they play. The programming advantages are thus the same as for inheritance. Therefore, the former will be the first focus of this lesson.



Lesson Objectives

After completing this lesson, you will be able to:

- Define and implement interfaces
- Implement interface methods
- Use interface references to make up-cast assignments
- Use interface references to make down-cast assignments
- Explain the term polymorphism with reference to interfaces
- Use cast assignments with interfaces to make generic calls

Business Example

You want to implement client/server relationships in combination with generic access from your model in ABAP Objects.

Areas of Use for Interfaces

Interfaces differ from regular inheritance in their area of use. In terms of programming, there are hardly any differences, however.

From a technical point of view, interfaces are simply superclasses that cannot be instantiated, do not have an implementation part, and only have public components. However, we will see that you can simulate multiple inheritance using interfaces.

In ABAP Objects, interfaces primarily serve to define uniform interfaces (protocols) for services. Various classes can thus offer – that is, implement – these services in different ways, but keep the same semantics. Interfaces therefore contain **no implementations**.

In ABAP Objects, the same components can generally be defined in interfaces and in classes. To recognize the semantic differences from regular inheritance, you can concentrate on the following, typical use cases:

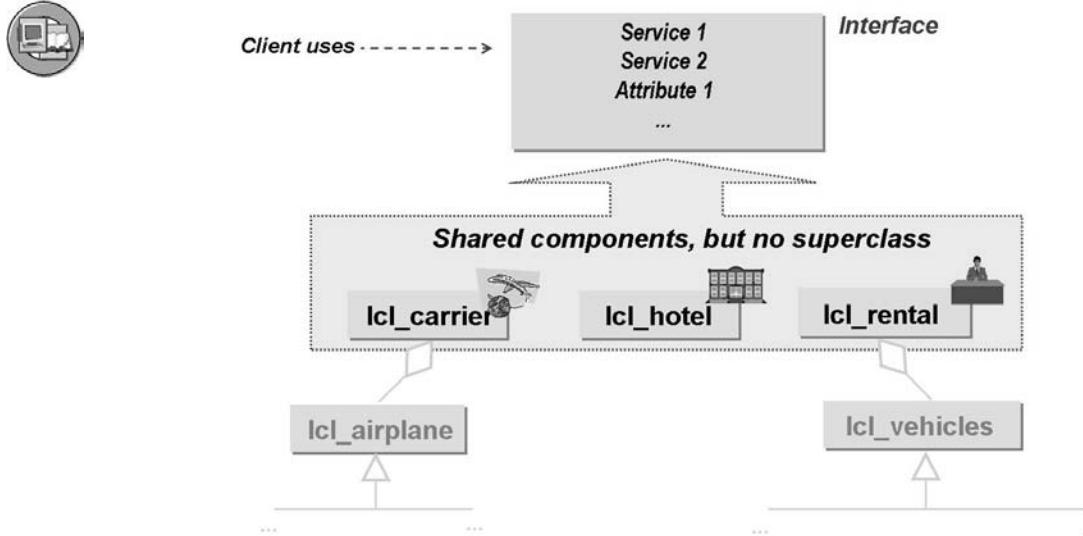


Figure 84: Central Definition of Shared Components

For example, you want to allow for the option of having multiple classes implementing a service in different ways, but using the same method names and with a uniform signature. With regular inheritance, you would define such a method in the shared superclass. However, if you cannot model a superclass suitably for inheritance, you need to define an interface and then define this method there. Therefore, you can compare this case with a generalization relationship with a superclass.

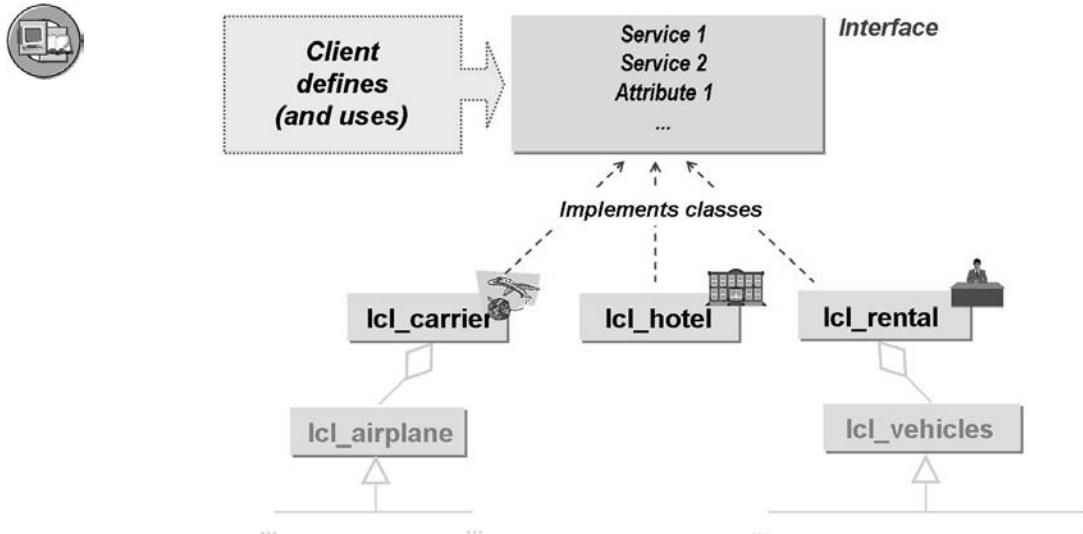


Figure 85: The Client Defines the Protocol

Compared to regular inheritance, the distribution of roles is sometimes different: The user generally defines the interfaces. In these, the user describes (both technically and semantically) the services that he/she wants the providers to offer. Each class can now decide for itself whether it serves the interface, that is, actually offers the services defined there. Therefore, this case is similar to a specialization relationship with a subclass.

As with regular inheritance, access to these services is then usually generic, that is, it uses a reference that is typed on the interface. As with regular inheritance, you can thus perform polymorphism with interfaces.

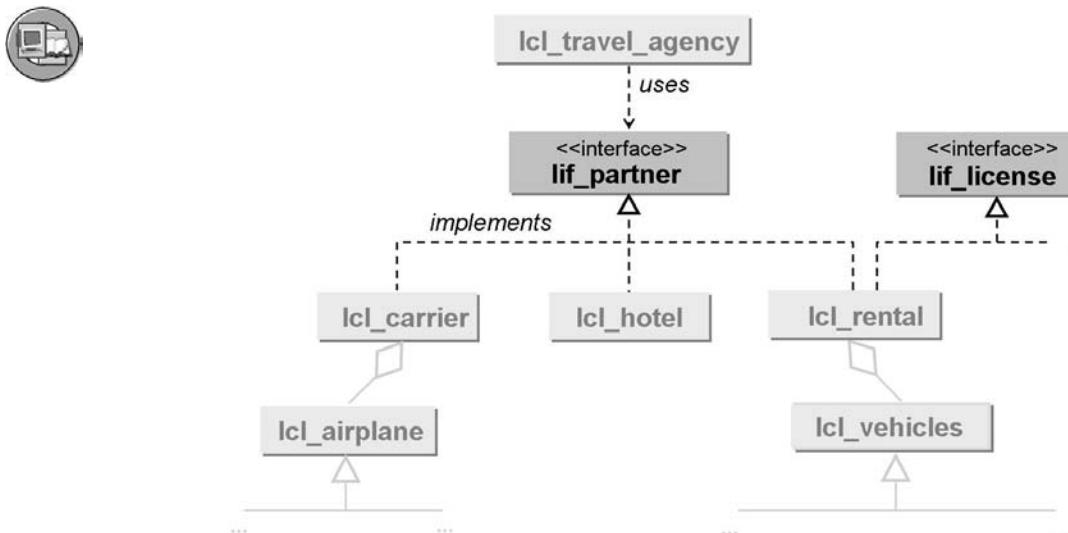


Figure 86: Interfaces in UML Notation

In UML, interfaces are represented in the same way as classes. However, in addition to their name, they have the stereotype «interface». The use of an interface is represented by a dotted line with a two-sided arrow from the user to the interface; the stereotype «uses» is optional. The fact that a class implements an interface is represented by a dotted arrow from the class to the interface.

The similarity with representing a generalization/specialization relationship is justified, as we have just described.

Creating Generalization/Specialization Relationships Using Interfaces

In ABAP Objects, the same components can be defined in an interface as in classes. However, interfaces do not know the visibility levels of components, that is, all components of an interface are public.

 **Note:** Note the areas of use for interfaces as the uniform interface for offering services.

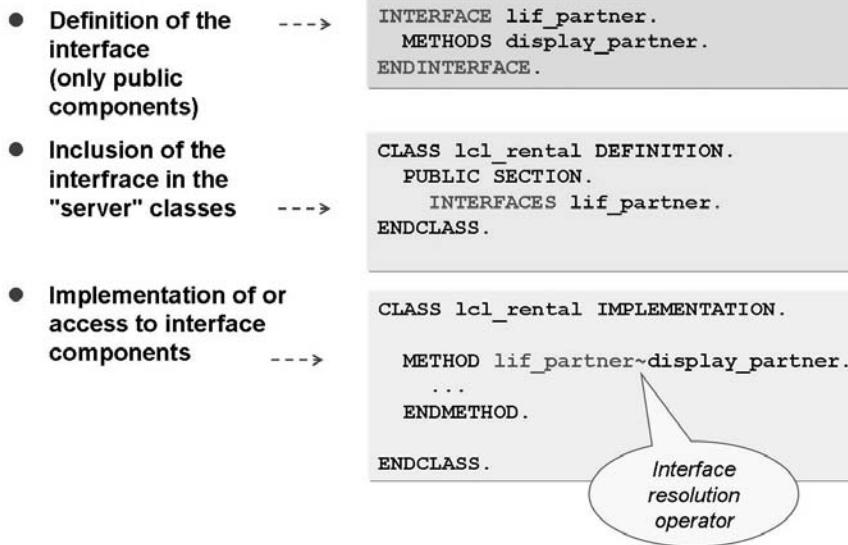


Figure 87: Defining and Implementing an Interface

Classes implement interfaces as follows:

- The interface name is listed in the definition part of the class with the INTERFACES statement. This must be in the PUBLIC SECTION, that is, interfaces can only be implemented publicly.
- The interface methods must be implemented in the implementation part of the class.
- The components defined in the interface can be addressed in the implementation part of the class.

Interface components are distinguished from the other components in the implementing class by prefixing the interface name followed by a tilde (~) (the interface resolution operator).

interface_name~component_name

To simplify access to interface components, you can use **alias names**. These can only appear in the definition part of a class or in the interface definition. Their use is subject to the visibility restriction of the defining class.

An alias for an interface method

```
ALIASES a_1 FOR lif_1~method_1.
```

The interface method `lif_1~method_1` can then be addressed with the shorter form `ref->a_1`.

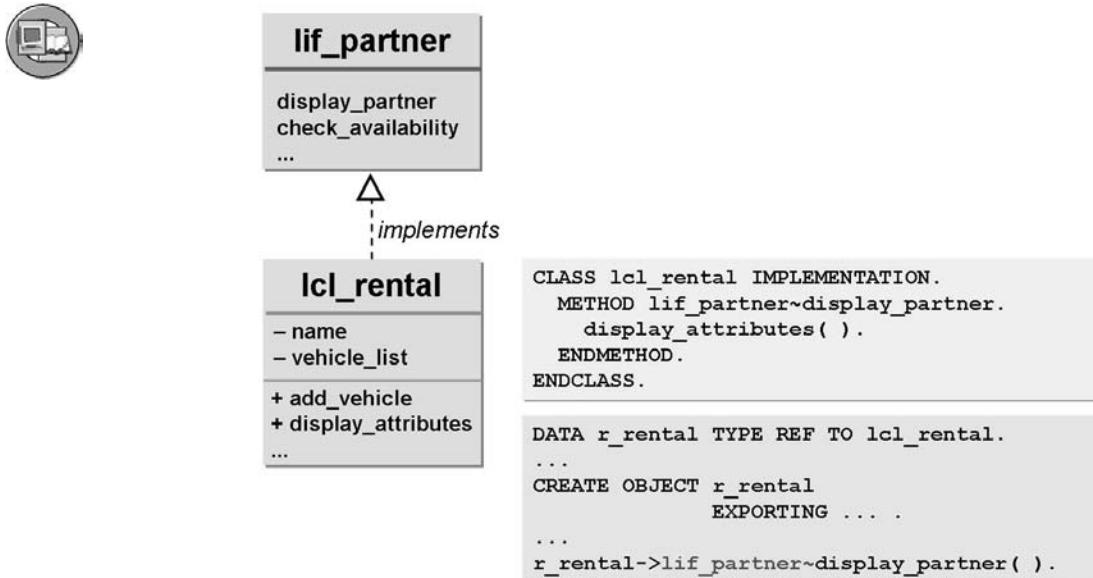


Figure 88: Addressing Interface Components Using Object References

Interface components can only be accessed by using an object reference whose class implements the interface. Syntactically, this takes place with the interface resolution operator `~`, just as with access to the interface components in the implementation part of the class.

Alternatively, you can use the alias names defined in the implementing class for the interface components. Even if shared components of the implementing classes are subsequently transferred to the interface, access to these components does not need to be adapted. However, the source code would then be less self-explanatory because you could conclude from the syntax that the components were defined in the class.

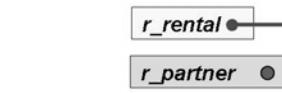
Polymorphism with Interfaces

Because interfaces cannot be instantiated, an interface reference can only refer to instances of classes that have implemented the interface. Therefore, if you want to perform polymorphism with interfaces, you must use up-cast to copy a reference to the reference variable that was typed with the help interface – as with regular inheritance.



```
DATA: r_rental  TYPE REF TO lcl_rental,
      r_partner TYPE REF TO lif_partner.
```

```
CREATE OBJECT r_rental.
```



* Up-Cast:
r_partner = r_rental.

```
r_partner
```

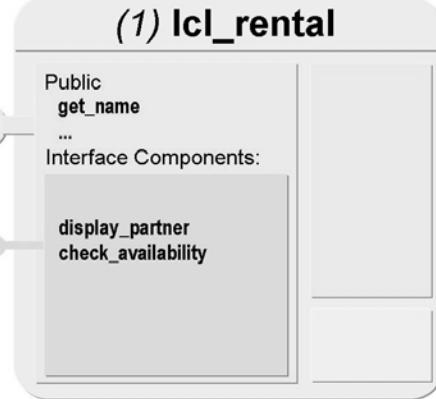


Figure 89: Up-Cast with Interface References

If the class has implemented the interface, it is certain that all components that can be accessed syntactically after the cast assignment are actually available in the instance. A user can thus address the instance of the implementing class using the interface. The prefixing of the interface name and the interface resolution operator is thus omitted. However, the user is restricted to using the components from the interface.

In the example shown here, the methods DISPLAY_PARTNER and CHECK_AVAILABILITY of the interface LIF_PARTNER can only be accessed after assigning the reference variable R_PARTNER. It is not possible to access the specific components of the instance from class LCL_RENTAL (GET_NAME in the above example) using the reference variable R_PARTNER.

The view is thus usually narrowed (or at least unchanged). That is why we describe this type of assignment of reference variables as **up-cast**. There is a switch from a view of several components to a view of a few components. The target reference can of course accept more dynamic types after the assignment than it could before. Therefore, the term **Widening Cast** is also suitable.

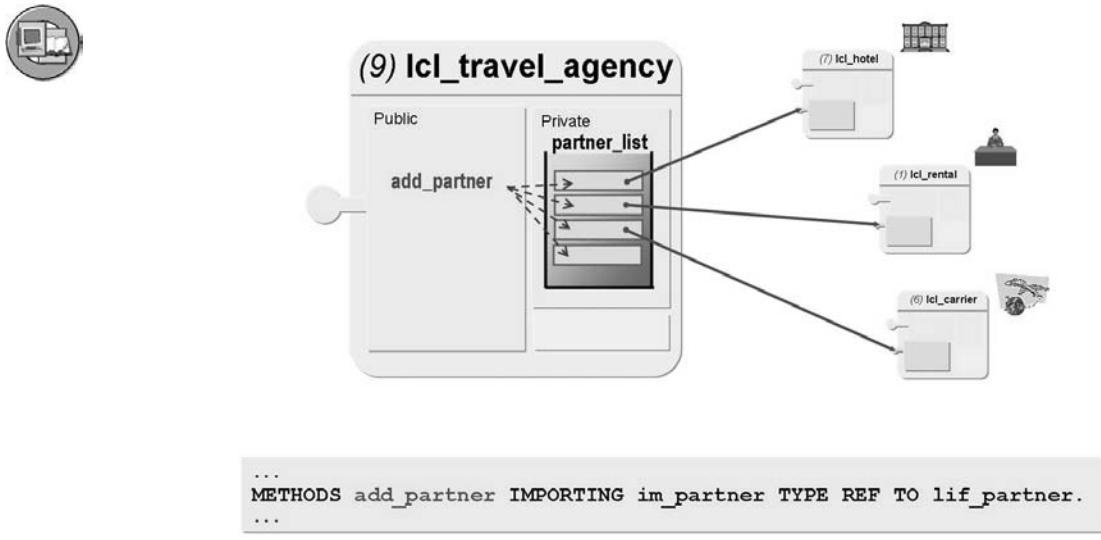


Figure 90: Row Type of the Internal Table in the Application Example

A typical area of use for up-cast assignments is preparation for generic access: A user who is not at all interested in the finer points of the instances of the classes that implement the interface but who simply wants to address the components defined in the interface, could use an interface reference for this access.

In the example shown here, a travel agency (LCL_TRAVEL_AGENCY) needs to manage all the various kinds of business partner in one list. The row type of the internal table must therefore be typed as the reference to the interface LIF_PARTNER.

The travel agency only wants to request the services in order to display their attributes and to check availability. The relevant methods DISPLAY_PARTNER and CHECK_AVAILABILITY are defined in the interface LIF_PARTNER and implemented in all business partner classes.

Objects of different classes (LCL_HOTEL, LCL_RENTAL, and LCL_CARRIER in the above example) can be kept in an internal table, typed with interface references (LIF_PARTNER in the above example). The components defined in the interface can then be accessed uniformly.

For this example, method ADD_PARTNER is therefore needed. This copies the references to all the kinds of business partner in this internal table. Its import parameter is already typed as the reference to the interface.



```
METHOD display_agency_partners.
DATA r_partner TYPE REF TO lif_partner.
LOOP AT partner_list INTO r_partner.
  r_partner->display_partner( ).
ENDLOOP.
ENDMETHOD.
```

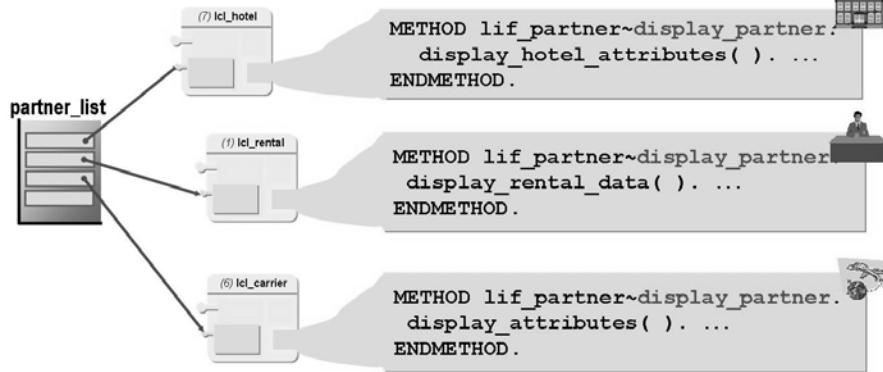


Figure 91: Polymorphism – Generic Access Using the Interface Reference

Polymorphism can also be performed for interfaces: Interface references can be used to call methods, whereby different implementations can be executed depending on the object of the reference.

The dynamic type and not the static type of the reference variable is used to search for the implementation of a method. In the above example, `r_partner->display_partner()` therefore uses the class of the instance to which `r_partner` actually refers in order to search for the implementation of `display_partner`. It does not, for example, use the static type for `r_partner` (which is always **REF TO lif_partner**).

The implementation that is now executed when `DISPLAY_PARTNER` is called therefore depends on the object to which the interface reference `R_PARTNER` currently refers. When objects from different classes react differently to the same method calls, this is known as **polymorphism**.

The option of performing polymorphism is one of the main strengths of interfaces: A client can handle different classes uniformly, irrespective of their implementation. The runtime system searches for the right implementation of a method on behalf of the client.

Polymorphism can be used to write programs that are highly generic, that is, they do not need to be changed significantly if use cases are added.

In the example given here, it would therefore be very easy to realize a boat rental addition: The relevant class – for example, with the name LCL_SHIPPING – would simply have to implement the interface LIF_PARTNER and thus the method DISPLAY_PARTNER defined there. Business partner management could then easily include shipowning companies and also request them to display their attributes.

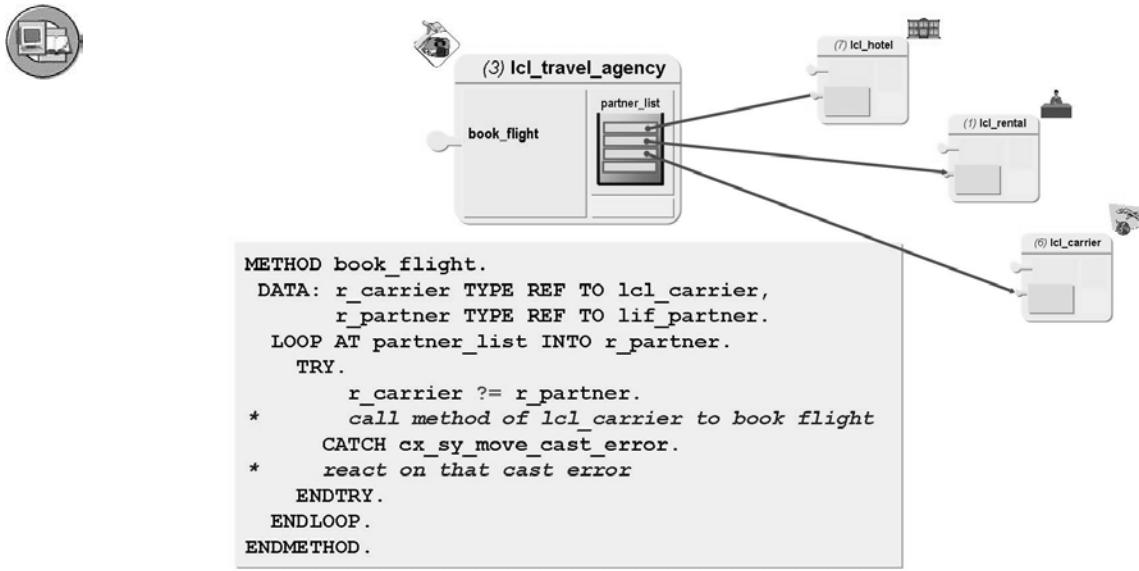


Figure 92: Down-Cast Assignment and Exception Handling in the Application Example

If you want to assign an interface reference to a class reference where the class has implemented the interface, you must use the down-cast assignment operator **MOVE ... ?TO ...** or its short form **?=**. Otherwise, the system would return a message stating that it is not certain that all components that can be accessed syntactically after the cast assignment are actually available in the instance. As a rule, the implementing class contains more components than the interface.

Interface reference variables can contain references to instances of the implementing class at runtime. After assigning this type of reference (back) to a reference to the implementing class, clients are no longer limited to interface components: In the example given here, all components of the LCL_CARRIER instance can be accessed (again) after the assignment using reference R_CARRIER.

The view is thus usually widened (or at least unchanged). That is why we describe this type of assignment of reference variables as **down-cast**. There is a switch from a view of a few components to a view of more components. As the target variable cannot accept as many dynamic types after this, the designation **Narrowing Cast** is also common.

A typical area of use for down-cast assignments is when specific components of instances need to be addressed whose references are kept in variables that are typed on the interface. A user who is interested in the **finer points** of instances of implementing classes cannot use the interface reference for this access because this only allows access to the interface components. In the example shown here, a travel agency (LCL_TRAVEL_AGENCY) needs to book a flight but keeps all the various business partner references in an internal table that was typed on the interface LIF_PARTNER.

Therefore, what happens if there is no airline reference in the interface reference R_PARTNER at runtime but the down-cast assignment operator is used to copy the reference to the then invalid reference R_CARRIER? In contrast to the up-cast assignment, it is possible that the static type of the target variable (R_CARRIER) is neither more general than nor the same as the dynamic type of the source variables (R_PARTNER), specifically if R_PARTNER contains hotel or car rental references.

That is why with this kind of cast, the runtime system checks, before the assignment, whether the current content of the source variable corresponds to the type requirements of the target variable. Otherwise, an exception that can be handled is triggered, and the original value of the target variable remains the same. This exception of error class CX_SY_MOVE_CAST_ERROR can be identified using TRY-ENDTRY and the CATCH statement.

Another way of preventing this runtime error would be to use runtime type identification (RTTI) classes. They can be used to determine the dynamic type at runtime and to set a condition for the cast.

Assignments between interface reference variables, whose typing interfaces are not related to each other, cannot be checked statically either and must therefore be performed using down-cast. With such an assignment, the system checks at runtime whether the class of the instance to which the source reference refers also supports the interface with which the target reference is typed.

Interface Hierarchies

We have already seen several times that an interface implementation strongly resembles regular inheritance. We therefore need to ask what the interface counterpart of hierarchical inheritance looks like. We want to explain interface hierarchies using an application example.

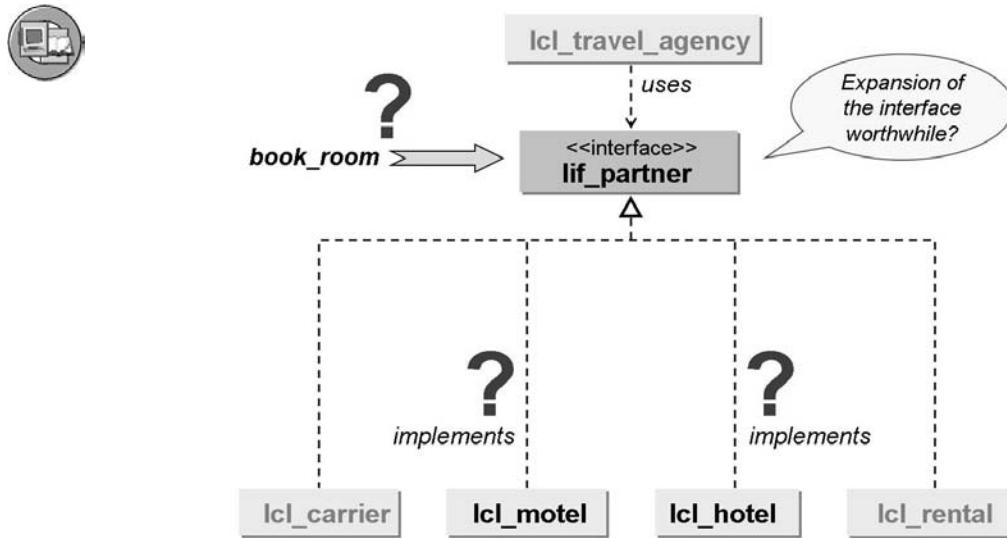


Figure 93: Interface Hierarchy in the Application Example

In this example, we need to know whether it would be useful to define a further service “room reservation” in the interface LIF_PARTNER. In this case, the classes LCL_CARRIER and LCL_RENTAL would have to implement the appropriate method because they have integrated the interface LIF_PARTNER. However, implementation of a room reservation – keeping the **same semantics** – is not conceivable for airlines or car rental companies.

However, because there are several other business partner types for which an implementation would be useful (for example, motels and hotels), the method needs to be centrally defined and not individually for motels and hotels. On the other hand, the option of easily extending the model with other accommodation provider types (for example, guesthouse) must be retained.

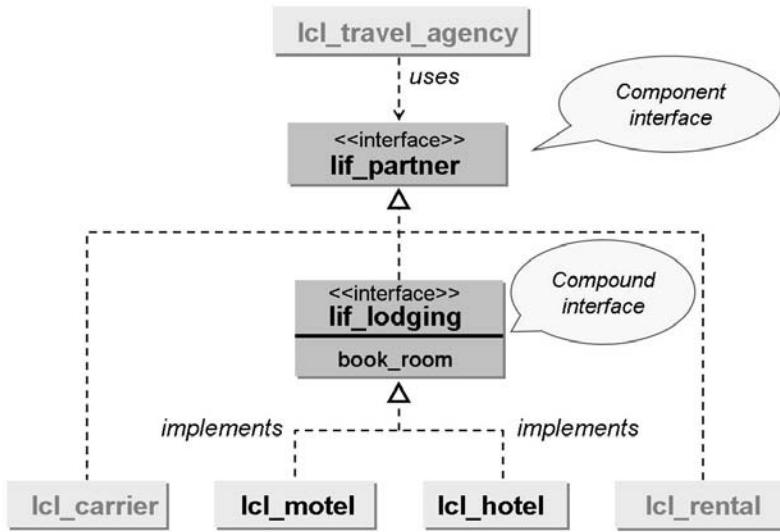


Figure 94: Compound Interface in UML Notation

In ABAP Objects, interfaces – like regular superclasses – can include other interfaces. As with regular inheritance, the result is interface hierarchies of any depth.

You can thus regard the including interface as a specialization of the included interface. It represents an extension of the included interface.

The including interface is known as a **compound interface**. An included interface represents a component of another interface and is therefore known as a **component interface**. An **elementary interface** does not itself contain other interfaces.

The UML notation corresponds to the implementation of an elementary interface by a class.

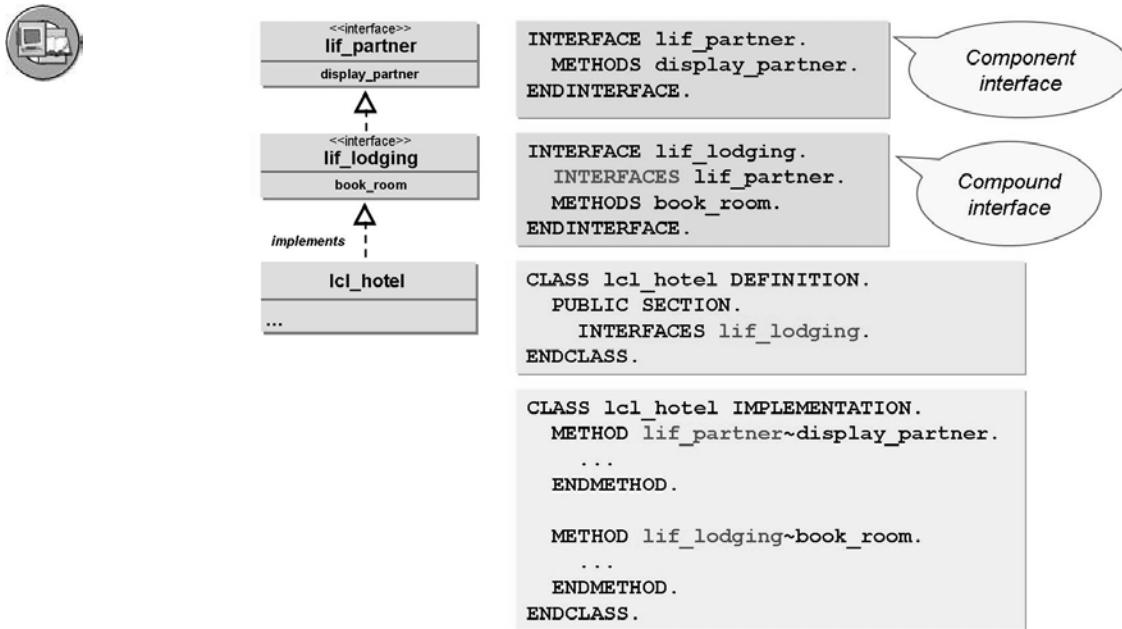


Figure 95: Definition and Implementation of Compound Interfaces: Syntax

As with regular inheritance, the implementing class only needs to list the compound interface in order to integrate all components. Nevertheless, the components of the component interfaces keep their **original** names:

`component_interface_name~component_name`

They are therefore **not** prefixed by the name of the compound interface.

All implementations of methods from all higher-level interfaces must take place in the first implementing class. Alias names are suitable for short-form syntax when accessing components from different interfaces. At the same time, you get a central documentary view of all components with the definition of these alias names in the implementing class.



```
DATA: r_hotel    TYPE REF TO lcl_hotel,
      r_partner TYPE REF TO lif_partner,
      r_lodging TYPE REF TO lif_lodging.
```

```
r_hotel->lif_partner~display_partner.
r_hotel->lif_lodging~book_room.

* Up-Casts for generic access:
r_lodging = r_hotel.
r_lodging->lif_partner~display_partner( ).
r_lodging->book_room( ).

r_partner = r_hotel.
r_partner->display_partner( ).

* Down-Casts for specific access again:
r_lodging ?= r_partner.
* or:
r_hotel ?= r_partner.
```

Figure 96: Addressing Components in Compound Interfaces – Syntax

The same possibilities apply for accessing its components from a compound interface and for cast assignments.



If there is no suitable way to link classes in terms of inheritance, creating generalization or specialization relationships using interfaces can have the following advantages:



- Separation of the protocol (interface – often defined by user) and the service (implementing class)
- Safe, generic method of access
- Ability to simulate multiple inheritance

All in all, this means that the entire software component can be extended very easily.

Figure 97: Using Interfaces

Interfaces are used to describe protocols for using components without connecting a kind of implementation. An **intermediate** layer is introduced between the client and the server to protect the client from the explicit server, thereby making the client independent.

Interfaces enable different classes to be handled uniformly, providing they implemented the interfaces. As with inheritance, you can also perform polymorphism using interface reference variables.

As with regular inheritance, the definition of an interface means an abstraction of the implementing classes to a specific partial aspect.

Multiple inheritance can be **simulated** using interfaces: If several interfaces are included, all components are available to all of these interfaces. All methods must be implemented.

Exercise 10: Interface Definition und Implementation

Exercise Objectives

After completing this exercise, you will be able to:

- Define interfaces
- Implement interfaces

Business Example

You now need to add a car rental company to your airplane management program. The airlines and car rental companies rental companies then need to be abstracted to business partners; that is, they need to be able to offer general services using an abstract interface. These services will be used in later exercises for travel agencies.

Task 1:

Define an interface with the service **DISPLAY_PARTNER** “Display of Business Partner Data” in order to later offer generic access options to potential clients (in our example a travel agency). The interface will be implemented in the class for airlines as well as in the class for car rentals.

1. Complete your program **ZBC401_##_MAIN** or copy the sample solution from the previous exercise **SAPBC401_CASS_MAIN_C** (## is your two-digit group number.))
2. Create the include **ZBC401_##_INTERFACE** and define an interface **LIF_PARTNERS** here.
3. The interface should contain the instance method **DISPLAY_PARTNER**. This method should not possess a signature! Later on, your task will be to display business partner attributes.
4. Integrate the include in your main program to produce a reasonable and correct order with the remaining class definitions. Keep in mind that other classes will use this interface definition later!

Continued on next page

Now check and activate your finished application.



Hint: The integration of the include with the new interface at this point in the exercise will not yet extend the application functionality of course; this will only happen in the next few steps of the task.

Task 2:

The class LCL_CARRIER should make the “service” DISPLAY_PARTNER from the interface available to potential clients. In addition, the implementation of the interface method DISPLAY_PARTNER is necessary.

1. Now present the newly created interface in your class LCL_CARRIER. It should therefore also be implemented there. This means that LCL_CARRIER acts as a “server class” in this context and will offer the service DISPLAY_PARTNER to a later client that has yet to be programmed.

Make sure you make yourself very familiar with the corresponding parts of the UML graphic here and, if necessary, supplement your own UML graphics in the right places! (An arrow should be dragged between the INTERFACE and LCL_CARRIER).

2. Implement the coding of the interface method DISPLAY_PARTNER in your class LCL_CARRIER As is evident from the name of this method, the data or attributes of this business partner should be displayed here. Consider possible solutions and execute the most suitable approach.

Check your entire application again for syntactical correctness and activate your application.



Hint: The performance of the entire application will still not change at this point; the results will only be seen in the later tasks.

Task 3:

The interface implemented in the class LCL_CARRIER should now be tested by calling from the main program.

1. Now navigate to your main program.

Here, remove all calls from DISPLAY_ATTRIBUTES that were used in the previous exercises, including those used for test purposes.

Continued on next page

2. In order to test the functioning of the services, that is, to test the interface methods, you now temporarily call the interface method DISPLAY_PARTNER of your class LCL_CARRIER as the next interim step.
How should this call from the main program be encoded?
3. What would the aim and objective of such a call at this point of the exercise be, or to put it differently, what does the option of this interface method call bring out of the main program in contrast with a conventional method call?
4. After a successful test turn this method call into a comment, so that it becomes ineffective.



Hint: The generic use of the interface methods using another class takes place in the next steps of the exercise.

The performance of the entire application will still not change at this point; the results will only be seen in the later tasks.

Task 4: Integration of the Training Model

In order to structure our application sensibly, the instructor's part model with the class LCL_RENTAL and the other vehicle classes (LCL_VEHICLE, LCL_TRUCK and LCL_BUS) should be integrated.

The first step is comprised of copying a template include: Copy the include **SAPBC401_VEHT_OPT** to your own, new include **ZSAPBC401_##_VEHT** and integrate this include into your main program. It contains the entire submodel for the classes LCL_RENTAL, LCL_VEHICLE, LCL_TRUCK and LCL_BUS.

For LCL_RENTAL implement the interface LIF_PARTNERS again (like in your own class LCL_CARRIER before).

The second step is comprised of generating instances for the classes of the RENTAL submodel, in order to give the application life. In order to make your work easier, a template program with the instantiation of these classes (LCL_TRUCK, LCL_BUS, LCL_RENTAL) is available (**SAPBC401_INTS_MAIN_OPT**). You can copy corresponding parts of the program to your main program using CTRL+C, CTRL+V.

Visualizing the objects is not yet necessary here. This will be done later. For the time being, you can deactivate or delete all calls for the DISPLAY methods in your main program!

1. The solution include is called: SAPBC401_VEHS_OPT
2. The solution main program is called: SAPBC401_INTS_MAIN_D.

Solution 10: Interface Definition und Implementation

Task 1:

Define an interface with the service DISPLAY_PARTNER “Display of Business Partner Data” in order to later offer generic access options to potential clients (in our example a travel agency). The interface will be implemented in the class for airlines as well as in the class for car rentals.

1. Complete your program **ZBC401_##_MAIN** or copy the sample solution from the previous exercise **SAPBC401_CASS_MAIN_C** (## is your two-digit group number.)
 - a) Carry out this step in the usual manner. Additional information on the copying of programs is available in the SAP Library.
2. Create the include **ZBC401_##_INTERFACE** and define an interface **LIF_PARTNERS** here.
 - a) See source text excerpt from the model solution.
 - b) The solution is called: **SAPBC401_INTS_INTERFACE**
3. The interface should contain the instance method **DISPLAY_PARTNER**. This method should not possess a signature! Later on, your task will be to display business partner attributes.
 - a) See source text excerpt from the model solution.
4. Integrate the include in your main program to produce a reasonable and correct order with the remaining class definitions. Keep in mind that other classes will use this interface definition later!

Now check and activate your finished application.



Hint: The integration of the include with the new interface at this point in the exercise will not yet extend the application functionality of course; this will only happen in the next few steps of the task.

- a) See source text excerpt from the model solution.
- b) Main program solution **SAPBC401_INTS_MAIN_A**

Continued on next page

Task 2:

The class LCL_CARRIER should make the “service” DISPLAY_PARTNER from the interface available to potential clients. In addition, the implementation of the interface method DISPLAY_PARTNER is necessary.

1. Now present the newly created interface in your class LCL_CARRIER. It should therefore also be implemented there. This means that LCL_CARRIER acts as a “server class” in this context and will offer the service DISPLAY_PARTNER to a later client that has yet to be programmed.

Make sure you make yourself very familiar with the corresponding parts of the UML graphic here and, if necessary, supplement your own UML graphics in the right places! (An arrow should be dragged between the INTERFACE and LCL_CARRIER).

- a) See source text excerpt from the model solution.
2. Implement the coding of the interface method DISPLAY_PARTNER in your class LCL_CARRIER As is evident from the name of this method, the data or attributes of this business partner should be displayed here. Consider possible solutions and execute the most suitable approach.

Check your entire application again for syntactical correctness and activate your application.



Hint: The performance of the entire application will still not change at this point; the results will only be seen in the later tasks.

- a) See source text excerpt from the model solution.
- b) The solution include is called: SAPBC401_INTS_B
- c) The corresponding main program is called: SAPBC401_INTS_MAIN_B

Task 3:

The interface implemented in the class LCL_CARRIER should now be tested by calling from the main program.

1. Now navigate to your main program.

Here, remove all calls from DISPLAY_ATTRIBUTES that were used in the previous exercises, including those used for test purposes.

- a) Carry out this step in the usual manner.

Continued on next page

2. In order to test the functioning of the services, that is, to test the interface methods, you now temporarily call the interface method DISPLAY_PARTNER of your class LCL_CARRIER as the next interim step.
How should this call from the main program be encoded?
 - a) See source text excerpt from the model solution.
 - b) The solution program is called: SAPBC401_INTS_MAIN_C
3. What would the aim and objective of such a call at this point of the exercise be, or to put it differently, what does the option of this interface method call bring out of the main program in contrast with a conventional method call?
 - a) Apart from this call's detailed and "longer" syntax, the interface method's call could be replaced by an ordinary instance method call from, for example, DISPLAY_ATTRIBUTES . At this point in the exercise this call does not seem altogether sensible.
4. After a successful test turn this method call into a comment, so that it becomes ineffective.



Hint: The generic use of the interface methods using another class takes place in the next steps of the exercise.

The performance of the entire application will still not change at this point; the results will only be seen in the later tasks.

- a) See the source code extract from the model solution.

Continued on next page

Task 4: Integration of the Training Model

In order to structure our application sensibly, the instructor's part model with the class LCL_RENTAL and the other vehicle classes (LCL_VEHICLE, LCL_TRUCK and LCL_BUS) should be integrated.

The first step is comprised of copying a template include: Copy the include **SAPBC401_VEHT_OPT** to your own, new include **ZSAPBC401_##_VEHT** and integrate this include into your main program. It contains the entire submodel for the classes LCL_RENTAL, LCL_VEHICLE, LCL_TRUCK and LCL_BUS.

For LCL_RENTAL implement the interface LIF_PARTNERS again (like in your own class LCL_CARRIER before).

The second step is comprised of generating instances for the classes of the RENTAL submodel, in order to give the application life. In order to make your work easier, a template program with the instantiation of these classes (LCL_TRUCK, LCL_BUS, LCL_RENTAL) is available (**SAPBC401_INTS_MAIN_OPT**). You can copy corresponding parts of the program to your main program using CTRL+C, CTRL+V.

Visualizing the objects is not yet necessary here. This will be done later. For the time being, you can deactivate or delete all calls for the DISPLAY methods in your main program!

1. The solution include is called: SAPBC401_VEHS_OPT
 - a)
2. The solution main program is called: SAPBC401_INTS_MAIN_D.
 - a)

Result

Source text extracts for the individual tasks

Task 1

```
Include SAPBC401_INTS_INTERFACE

*&-----
*&   Include          SAPBC401_INTS_INTERFACE
*&-----*



*-----*
* define interface lif_partners in this includefile
*-----*
```

Continued on next page

```

*-----*
INTERFACE lif_partners.
METHODS display_partner.
ENDINTERFACE.

Programm SAPBC401_INTS_MAIN_A

REPORT sapbc401_ints_main_a.
*** This is the MAIN program ! ****
TYPE-POOLS icon.

*** include the new includefile with the interface definition !
INCLUDE sapbc401_ints_interface.
INCLUDE sapbc401_ints_a.

DATA: r_plane TYPE REF TO lcl_airplane,
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_passenger TYPE REF TO lcl_passenger_plane,
      r_carrier TYPE REF TO lcl_carrier,

START-OF-SELECTION.
*#####
*   nothing has to be changed here...****
```

Task 2

```

Include SAPBC401_INTS_B

*&-----
*& Include          SAPBC401_INTS_B
*&-----
```

*...definitions of lcl_airplane and other classes

- * now the class LCL_CARRIER has to be extended
- * this class implements the interface LIF_PARTNERS !

Continued on next page

```

*-----*
*      CLASS lcl_carrier DEFINITION
*-----*
*-----*
CLASS lcl_carrier DEFINITION.

PUBLIC SECTION.

"-----
INTERFACES lif_partners.
METHODS: constructor IMPORTING im_name TYPE string,
          get_name RETURNING value(ex_name) TYPE string,
          add_airplane IMPORTING
                  im_plane TYPE REF TO lcl_airplane,
          display_airplanes,
          display_attributes.

PRIVATE SECTION.

"-----
DATA: name           TYPE string,
      airplane_list TYPE TABLE OF REF TO lcl_airplane.
ENDCLASS.           "lcl_carrier DEFINITION

*-----*
*      CLASS lcl_carrier IMPLEMENTATION
*-----*
*-----*
CLASS lcl_carrier IMPLEMENTATION.

*   implement the interfacemethod !
METHOD lif_partners~display_partner.
  display_attributes( ).           "lif_partners~display_partner
ENDMETHOD.

*..... nothing has to be changed here...

ENDCLASS.           "lcl_carrier IMPLEMENTATION

```

Task 3

```

Programm SAPBC401_INTS_MAIN_C

REPORT sapbc401_ints_main_c.
*   change the main programm to show how the

```

Continued on next page

```

*     interfacemethod DISPLAY_PARTNER can be called !

TYPE-POOLS icon.

INCLUDE sapbc401_ints_interface.
INCLUDE sapbc401_ints_b.

DATA: r_plane TYPE REF TO lcl_airplane,
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_passenger TYPE REF TO lcl_passenger_plane,
      r_carrier TYPE REF TO lcl_carrier,
      r_lif TYPE REF TO lif_partners.

START-OF-SELECTION.
*#####
***** Create CARRIER *****
CREATE OBJECT r_carrier EXPORTING im_name = 'Smile&Fly Travel'.

* ...nothing has to be changed...
*... creating several other classes like LCL_AIRPLANE...

*** call the interface method DISPLAY_PARTNER
*** that is implemented in the class LCL_CARRIER
*** please think about the purpose of this! Could this be achieved easier ?
r_carrier->lif_partners-display_partner( ).
```

Task 4

Include SAPBC401_VEHS_OPT

```

*&-----
*& Include          SAPBC401_VEHS_OPT
*& Solution which contains all classes of submodell RENTAL / VEHICLE
*& and the interface LIF_PARTNERS is implemented in class LCL_RENTAL
*&
*&-----*
```

Continued on next page

```

*      CLASS lcl_vehicle DEFINITION
*-----
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION.

"-----
METHODS: get_average_fuel IMPORTING im_distance TYPE s_distance
          im_fuel TYPE s_capacity
          RETURNING value(re_avgfuel) TYPE s_consum.

METHODS      constructor IMPORTING im_make TYPE string.
METHODS      display_attributes.
METHODS      set_make IMPORTING im_make TYPE string.
METHODS      get_make EXPORTING ex_make TYPE string.
CLASS-METHODS get_count EXPORTING re_count TYPE i.

PRIVATE SECTION.

"-----
DATA: make      TYPE string.

METHODS      init_make.

CLASS-DATA: n_o_vehicles TYPE i.

ENDCLASS.           "lcl_vehicle DEFINITION

*-----
*      CLASS lcl_vehicle IMPLEMENTATION
*-----


CLASS lcl_vehicle IMPLEMENTATION.

METHOD constructor.
  make = im_make.
  ADD 1 TO n_o_vehicles.
ENDMETHOD.

METHOD get_average_fuel.
  re_avgfuel = im_distance / im_fuel.
ENDMETHOD.

METHOD set_make.

```

Continued on next page

```

        IF im_make IS INITIAL.
            init_make( ).      " me->init_make( ).
        ELSE.
            make = im_make.
        ENDIF.
    ENDMETHOD.

METHOD init_make.
    make = 'default make'.
ENDMETHOD.                      "init_make

METHOD get_make.
    ex_make = make.
METHOD get_make.                  "get_make

METHOD display_attributes.
    WRITE: make.
ENDMETHOD.                      "display_attributes

METHOD get_count.
    re_count = n_o_vehicles.
ENDMETHOD.                      "get_count
ENDCLASS.                         "lcl_vehicle IMPLEMENTATION

*-----*
*      CLASS lcl_truck DEFINITION
*-----*
CLASS lcl_truck DEFINITION INHERITING FROM lcl_vehicle.

PUBLIC SECTION.
    -----
METHODS:      constructor IMPORTING im_make TYPE string
                im_cargo TYPE s_plan_car.

METHODS      display_attributes REDEFINITION.
METHODS      get_cargo RETURNING value(re_cargo) TYPE s_plan_car.

PRIVATE SECTION.
    -----
DATA: max_cargo TYPE s_plan_car.

```

Continued on next page

```

ENDCLASS

*-----*
*      CLASS lcl_truck IMPLEMENTATION
*-----*

CLASS lcl_truck IMPLEMENTATION.

METHOD constructor.
    super->constructor( im_make ).
    max_cargo = im_cargo.
ENDMETHOD.                      "constructor

METHOD display_attributes.
    WRITE: / icon_ws_truck AS ICON.
    super->display_attributes( ).  

    WRITE: 20 ' Cargo = ', max_cargo.
    ULINE.
ENDMETHOD.                      "display_attributes

METHOD get_cargo.
    re_cargo = max_cargo.
ENDMETHOD.                      "get_cargo

ENDCLASS.

*-----*
*      CLASS lcl_bus DEFINITION
*-----*

CLASS lcl_bus DEFINITION INHERITING FROM lcl_vehicle.

PUBLIC SECTION.
"-----
METHODS:     constructor IMPORTING im_make TYPE string
             im_passengers TYPE i.

METHODS      display_attributes REDEFINITION.

PRIVATE SECTION.
"-----
DATA: max_passengers TYPE i.

```

Continued on next page

```

ENDCLASS.

*-----*
*      CLASS lcl_bus IMPLEMENTATION
*-----*
CLASS lcl_bus IMPLEMENTATION.

METHOD constructor.
    super->constructor( im_make ).
    max_passengers = im_passengers.
ENDMETHOD.                      "constructor

METHOD display_attributes.
    WRITE: / icon_transportation_mode AS ICON.
    super->display_attributes( ).
    WRITE: 20 ' Passengers = ', max_passengers.
    ULINE.
ENDMETHOD.                      "display_attributes

ENDCLASS.

*-----*
*      CLASS lcl_rental DEFINITION
*-----*
CLASS lcl_rental DEFINITION.

PUBLIC SECTION.
    "-----
    INTERFACES lif_partners.
METHODS:   constructor IMPORTING im_name TYPE string.
METHODS:   add_vehicle IMPORTING im_vehicle
            TYPE REF TO lcl_vehicle.
METHODS:   display_attributes.

PRIVATE SECTION.
    "-----
    DATA: name TYPE string,
          vehicle_list TYPE TABLE OF REF TO lcl_vehicle.
ENDCLASS.

*-----*

```

Continued on next page

```

*      CLASS lcl_rental IMPLEMENTATION
*-----
CLASS lcl_rental IMPLEMENTATION.

METHOD lif_partners-display_partner.
    display_attributes( ).
ENDMETHOD.

METHOD constructor.
    name = im_name.
ENDMETHOD.                      "constructor

METHOD add_vehicle.
    APPEND im_vehicle TO vehicle_list.
ENDMETHOD.                      "add_vehicle

METHOD display_attributes.
DATA: r_vehicle TYPE REF TO lcl_vehicle.
    WRITE: / icon_transport_proposal AS ICON, name.
    WRITE: ' Here comes the vehicle list: '. ULINE. ULINE.
    LOOP AT vehicle_list INTO r_vehicle.
        r_vehicle->display_attributes( ).
    ENDLOOP.
ENDMETHOD.                      "display_attributes

ENDCLASS.                      "lcl_rental IMPLEMENTATION

```

```

Programm SAPBC401_INTS_MAIN_D

REPORT sapbc401_ints_main_d.
*   this is the final main-program showing instances of both submodels!

TYPE-POOLS icon.
*   include file with the interfacedefinition.
INCLUDE sapbc401_ints_interface
*   now the submodel of the trainer is included with rental and all vehicle classes
INCLUDE sapbc401_vehs_opt.
*   submodel with carrier and all the airplane classes
INCLUDE sapbc401_ints_b.

```

Continued on next page

```
DATA: r_plane TYPE REF TO lcl_airplane,
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_passenger TYPE REF TO lcl_passenger_plane,
      r_carrier TYPE REF TO lcl_carrier,
      r_rental TYPE REF TO lcl_rental,
      r_truck TYPE REF TO lcl_truck,
      r_bus TYPE REF TO lcl_bus,
      r_lif TYPE REF TO lif_partners.

START-OF-SELECTION.
*#####
***** Create CARRIER *****
CREATE OBJECT r_carrier EXPORTING im_name = 'Smile&Fly Travel'.

* ...creating instances of airplane classes and add them to LCL_CARRIER
*... this has already been shown in many excercises before

***** Create Rental and other objects of vehicle submodel *****
CREATE OBJECT r_rental EXPORTING im_name = 'Happy Car Rental'.

* create some vehicles and add them to the rentalclass
CREATE OBJECT r_truck EXPORTING im_make = 'MAN'
          im_cargo = 45.
r_rental->add_vehicle( r_truck ).

CREATE OBJECT r_bus EXPOTRING im_make = 'VOLVO'
          im_max_passengers = 56.
r_rental->add_vehicle( r_bus ).

*** just to show the call of the interfacemethod for testing
*r_carrier->lif_partners~display_partner( ).
*r_rental->lif_partners~display_partner( ).
```

Exercise 11: Use of Interfaces

Exercise Objectives

After completing this exercise, you will be able to:

- Work with interfaces
- Use the services of interfaces as a client

Business Example

You now need to add a user class to your program: Travel agencies need to manage various business partners using the interface and access the business partners' general services.

Task 1:

Define a local class for travel agencies.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise. (## is your two-digit group number.))
2. If necessary, add a class **LCL_TRAVEL_AGENCY** that uses the interface LIF_PARTNERS to your UML diagram.
3. To lighten your typing load, you can copy sections of source text from the template **SAPBC401_VEHT_B** for the local class LCL_TRAVEL_AGENCY to your main- or one of your include programs in order to supplement them. If required, the class LCL_TRAVEL_AGENCY can also be created from scratch, without using templates.
4. As a private class attribute, add an internal table **PARTNER_LIST** for the buffering of references to the business partners who have implemented the interface LIF_PARTNERS.
5. Add to the signature and implementation of method **ADD_PARTNER** so that the business partner references can be added to the list PARTNER_LIST.
6. Implement the method **DISPLAY_AGENCY_PARTNERS** so that the attributes of all business partners of the travel agency can be added to the list. The centrally provided method DISPLAY_PARTNER needs to be called for each business partner.

Continued on next page

Task 2:

In the main program, generate a travel agency instance, transfer the references to the airline and car rental company to this instance, and add the attributes.

1. In the main program, define a suitably typed reference variable for your new travel agency class.
2. Using the reference, generate an instance of your class LCL_TRAVEL_AGENCY. Decide for yourself how to fill the attributes.
3. Call the method ADD_PARTNER to transfer the references to the generated airline and car rental company instances to the travel agency.
4. Add the attributes of the travel agency by calling its method **DISPLAY_AGENCY_PARTNERS**.

The method DISPLAY_ATTRIBUTES could also be (optionally) created in this class, and would output the names of travel agencies and all business partners together. In this method it is necessary to call the method DISPLAY_AGENCY_PARTNERS after the output of the travel agency names.

Solution 11: Use of Interfaces

Task 1:

Define a local class for travel agencies.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise. (## is your two-digit group number.)
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
 - b) Model solution: **SAPBC401_INTS_MAIN_E**
2. If necessary, add a class **LCL_TRAVEL_AGENCY** that uses the interface **LIF_PARTNERS** to your UML diagram.
 - a) Speak to your instructor if you have any questions.
3. To lighten your typing load, you can copy sections of source text from the template **SAPBC401_VEHT_B** for the local class **LCL_TRAVEL_AGENCY** to your main- or one of your include programs in order to supplement them. If required, the class **LCL_TRAVEL_AGENCY** can also be created from scratch, without using templates.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
4. As a private class attribute, add an internal table **PARTNER_LIST** for the buffering of references to the business partners who have implemented the interface **LIF_PARTNERS**.
 - a) See the source code extract from the model solution.
5. Add to the signature and implementation of method **ADD_PARTNER** so that the business partner references can be added to the list **PARTNER_LIST**.
 - a) See the source code extract from the model solution.
6. Implement the method **DISPLAY_AGENCY_PARTNERS** so that the attributes of all business partners of the travel agency can be added to the list. The centrally provided method **DISPLAY_PARTNER** needs to be called for each business partner.
 - a) See the source code extract from the model solution.

Continued on next page

Task 2:

In the main program, generate a travel agency instance, transfer the references to the airline and car rental company to this instance, and add the attributes.

1. In the main program, define a suitably typed reference variable for your new travel agency class.
 - a) See the source code extract from the model solution.
2. Using the reference, generate an instance of your class LCL_TRAVEL_AGENCY. Decide for yourself how to fill the attributes.
 - a) See the source code extract from the model solution.
3. Call the method ADD_PARTNER to transfer the references to the generated airline and car rental company instances to the travel agency.
 - a) See the source code extract from the model solution.
4. Add the attributes of the travel agency by calling its method **DISPLAY_AGENCY_PARTNERS**.

The method DISPLAY_ATTRIBUTES could also be (optionally) created in this class, and would output the names of travel agencies and all business partners together. In this method it is necessary to call the method DISPLAY_AGENCY_PARTNERS after the output of the travel agency names.

- a) Solution main program: SAPBC401_INTS_MAIN_E.
- b) Solution include: SAPBC401_INTS_AGENCY.

Result

Source text of the model solution:

SAPBC401_INTS_AGENCY

```
* - this is the includefile with the classdefinition of LCL_TRAVEL_AGENCY -
*-----*
*      CLASS lcl_travel_agency DEFINITION
*-----*
CLASS lcl_travel_agency DEFINITION.

PUBLIC SECTION.
"-----"
METHODS:      constructor IMPORTING im_name TYPE string.
METHODS       add_partner IMPORTING im_partner
```

Continued on next page

```

        TYPE REF TO lif_partners.

METHODS      display_agency_partners.
METHODS:    display_attributes.

PRIVATE SECTION.

"-----
DATA: name TYPE string,
      partner_list TYPE TABLE OF REF TO lif_partners.

ENDCLASS.          "lcl_travel_agency DEFINITION

*-----*
*      CLASS lcl_travel_agency IMPLEMENTATION
*-----*
CLASS lcl_travel_agency IMPLEMENTATION.

METHOD display_attributes.

  write: / 'Name of the agency: ', name.  ULINE.
  write: / 'Here are the partners of the agency :'.
  display_agency_partners( ).

ENDMETHOD.

METHOD display_agency_partners.

  DATA: r_partner TYPE REF TO lif_partners.
  WRITE: icon_dependents AS ICON, name.
  WRITE: ' Here are the partners of the travel agency: '.  ULINE.
  LOOP AT partner_list INTO r_partner.
    r_partner->display_partner( ).
  ENDLOOP.

ENDMETHOD.          "display_agency_partners

METHOD constructor.

  name = im_name.

ENDMETHOD.          "constructor

METHOD add_partner.

  APPEND im_partner TO partner_list.

ENDMETHOD.          "add_partner

ENDCLASS.          "lcl_travel_agency IMPLEMENTATION

```

Continued on next page

SAPBC401_INTS_MAIN_E

```

* - this is the main program -
REPORT    sapbc401_ints_main_e.

TYPE-POOLS icon.
INCLUDE sapbc401_ints_interface.
INCLUDE sapbc401_ints_agency.
INCLUDE sapbc401_vehs_opt.
INCLUDE sapbc401_ints_b.

DATA: r_plane TYPE REF TO lcl_airplane,
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_passenger TYPE REF TO lcl_passenger_plane,
      r_carrier TYPE REF TO lcl_carrier,
      r_agency TYPE REF TO lcl_travel_agency,
      r_rental TYPE REF TO lcl_rental,
      r_truck TYPE REF TO lcl_truck,
      r_bus TYPE REF TO lcl_bus.

START-OF-SELECTION.
*#####
***** Create TRAVEL_AGENCY *****
CREATE OBJECT r_agency EXPORTING im_name = 'Fly&Smile Travel'.

***** Create CARRIER *****
CREATE OBJECT r_carrier EXPORTING im_name = 'Smile&Fly Travel'.
..
***** insert business-parnter of agency into partner_list*****
r_agency->add_partner( r_carrier ).

***** create RENTAL *****
CREATE OBJECT r_rental EXPORTING im_name = 'HAPPY CAR RENTAL'.

*** creating all the VEHICLE and AIRPLANE objects is left out here
*** it has been shown several times before

***** insert business-parnter of agency into partner_list*****
r_agency->add_partner( r_rental ).

***** show attributes of agency including all businesspartners *****
r_agency->display_attributes( ).
```



Lesson Summary

You should now be able to:

- Define and implement interfaces
- Implement interface methods
- Use interface references to make up-cast assignments
- Use interface references to make down-cast assignments
- Explain the term polymorphism with reference to interfaces
- Use cast assignments with interfaces to make generic calls

Related Information

For more information, refer to the SAP Library.

Lesson: Events

Lesson Overview

This lesson provides a general introduction to the concept of events in object orientation, followed by an explanation of all related modeling aspects and syntax elements. The section entitled 'Registration' contains the information that is probably most important for understanding the topic.



Lesson Objectives

After completing this lesson, you will be able to:

- Define and trigger events
- Handle events
- Register and deregister event handling
- Explain the key differences between explicit method calls and event-controlled method calls

Business Example

You want to implement event-controlled behavior from your model in ABAP Objects.

Event-Controlled Method Calls

Besides attributes and methods, classes – and their instances – can contain a third type of component: **events**. Instance events can be **triggered** by the instances of the class, while static instance events can be triggered by the class itself.

Events can also be defined as interface components.

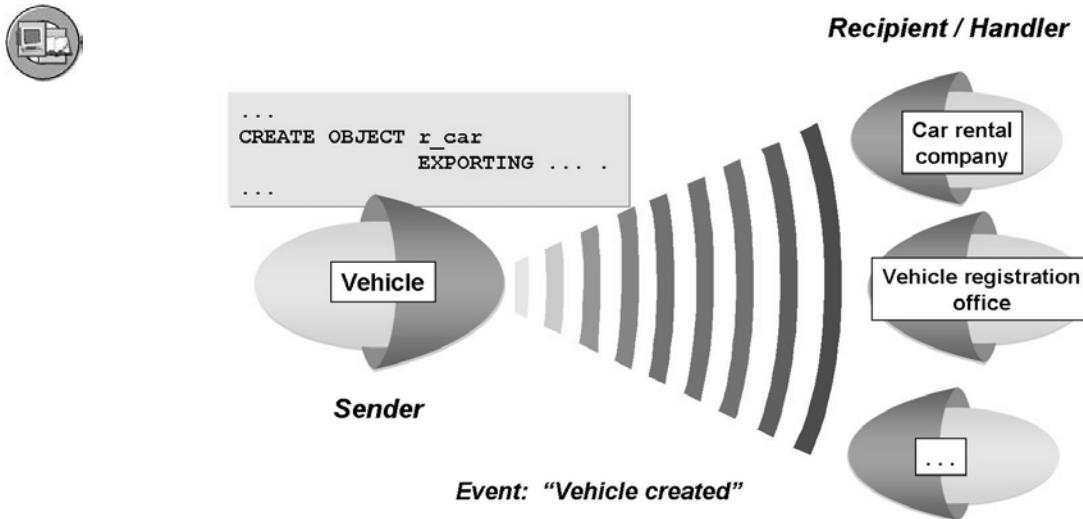


Figure 98: Event-Controlled Method Calls

Given the right circumstances, handler methods can now react to the triggering of this event. This means that the **runtime system** may call these handler methods after the event has been triggered. In other words, the handler method is not usually called by the client directly.

This results in a completely different modeling concept: While you are developing the class that triggers the event, you do not need to know anything about the class that is handling it. The triggering class sends a specific message to all classes (and, if need be, their instances) of the running program. At the time of development, it is completely unclear what type of handlers there will be and how many will be used.

Because of the definition of the handler method, the range of possible results can be narrowed down, but which, if any, of these results actually will occur can only be determined after the event has been triggered.

An event can have exporting parameters, which means that, in contrast to the explicit method call, the **calling program** determines the protocol in this case.

In this application example, after an instance in the “vehicle” class is created, it triggers the event “vehicle created.” This event is received by different instances and processed differently by each one. The rental car company considers purchasing a vehicle, while the vehicle registration office registers the car, and so on.



Caution: Do not confuse this concept of events in object-oriented programming with events in the ABAP runtime system (LOAD-OF-PROGRAM, START-OF-SELECTION, and so on). Also, do not confuse it with background processing or workflow control.

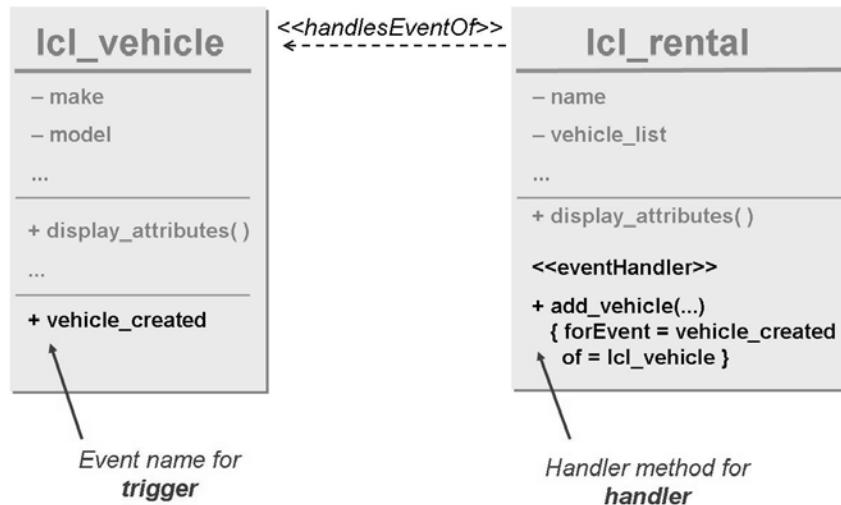


Figure 99: Event Handling in a UML Class Diagram

In UML class diagrams, a dotted arrow with the stereotype «handlesEventOf» points from the handling class to the triggering class. The event definition and signature only appear implicitly in the handling class within the handler method. The handler methods are separated from the other methods using the stereotype «eventHandler».

Triggering and Handling Events

The following summarizes all of the required programming steps for event-controlling. They will be explained in more detail later.



- **Trigger events**
 - 1 ▶
 - Class defines an event
(EVENTS, CLASS-EVENTS)
 - 2 ▶
 - Object or class triggers the event
(RAISE EVENT)

- **Handle events**
 - 3 ▶
 - Handler class defines and implements the handler method
([CLASS-]METHODS... FOR EVENT ... OF ...)
 - 4 ▶
 - “Handler object” or handler class is registered to events at runtime
(SET HANDLER)

Figure 100: Triggering and Handling Events – Overview

Keep in mind that, depending on the status of your application, you may not need to program all steps. The separation of cause and effect in your programming should be reflected in the way you construct complex applications. Often, the event is already triggered and all you have to do is create another event handler.

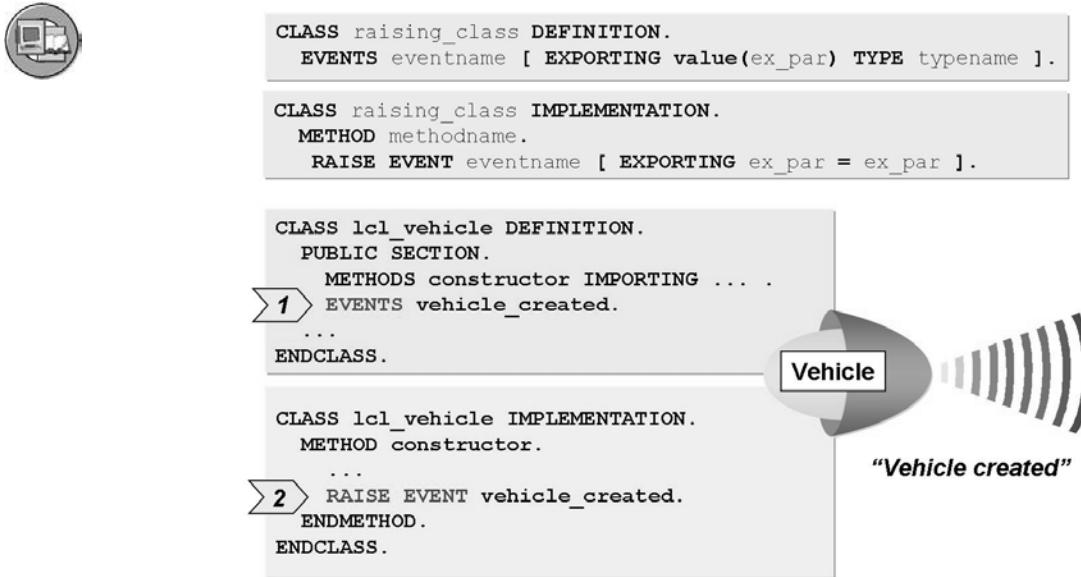


Figure 101: Defining and Triggering Events – Syntax

Within a class, instance events are defined using the EVENTS statement, while static events are defined using the CLASS-EVENTS statement.

Events can only have exporting parameters, which **must** be passed by **value as a copy**.

A class or instance can trigger an event at runtime using the RAISE EVENT statement. Both instance events and static events can be triggered in instance methods. Only static events can be triggered in static methods.

When an event is triggered, the handler methods that are registered to this event are called **in sequence**. Of course, these can trigger more events of their own.



```
CLASS handling_class DEFINITION.
METHODS on_eventname FOR EVENT eventname
    OF raising_class | interface
    [ IMPORTING ex_par_1 .. ex_par_n [sender] ].
```

```
CLASS lcl_rental DEFINITION.
...
3> METHODS add_vehicle FOR EVENT vehicle_created OF lcl_vehicle
    IMPORTING sender.
ENDCLASS.
```

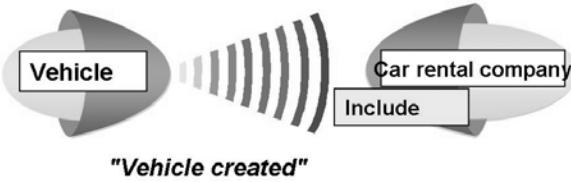


Figure 102: Handling Events – Syntax

Instance events or static methods can be defined within a class to handle events. To do so, you must specify the event (FOR EVENT) and the class or interface in which the event was defined (OF).

If the event contains exporting parameters and you want to be able to address these syntactically, you **must** have specified the exporting parameters immediately after IMPORTING in the definition of the method. The handler method's signature can consist of no more than the exporting parameters of the associated event. The parameters are typed by the handler method during the definition of the event. (The object that triggers the event determines the protocol.)

In addition to the explicitly defined exporting parameters, the predefined importing parameter SENDER can **always** be listed. By using that parameter, you can place a reference to the event-trigger object into the handler method.

Therefore, handler methods are usually called by triggered events (RAISE EVENT): However, they can also be called explicitly (CALL METHOD).

Registering for Events

The definition of the handler method only specifies **how** and to **which event** of which class the method will react. At **runtime**, it needs to be determined **which** possible reactions will actually take place and **when** each of these will happen.

When triggering instance events, you also have to specify what event the reaction will trigger. If instance methods are set to carry out the reaction, you also have to specify **which instance(s)** will perform the reaction.

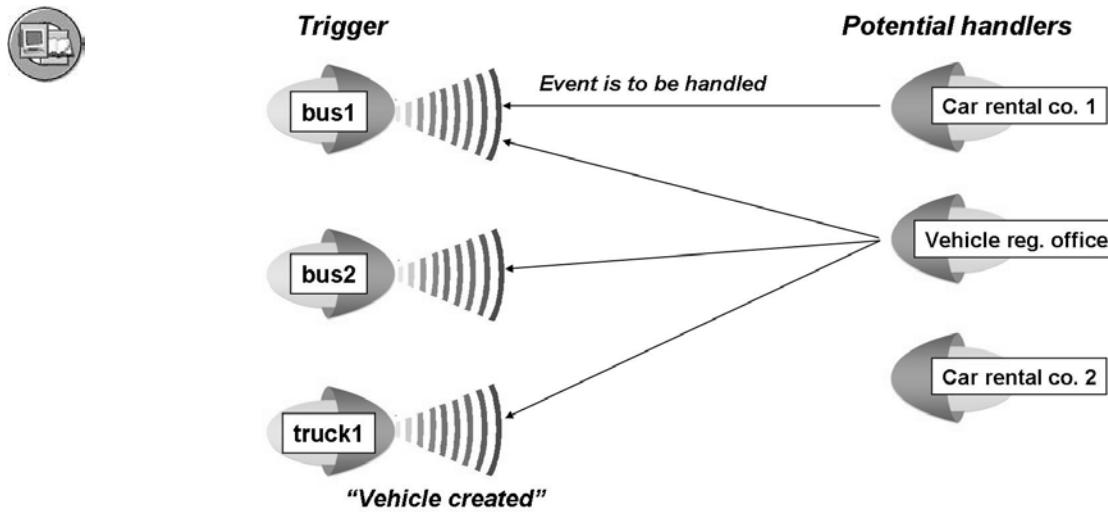


Figure 103: Registering Event Handling

These specifications are collectively known as **registration**. The registration is **always** carried out **using the trigger**. When the event is triggered, the runtime uses the registrations of the trigger to determine which event handler methods need to be called.

In this example, handler methods are defined for the event of the vehicle class, the car rental class, and the vehicle registration class. However, you can only predetermine which car rental instances and vehicle registration instances will react to which vehicle instance, and when they will do so.

Registrations can also be revoked.



```
SET HANDLER ref_handler->on_eventname
[ FOR ref_sender | FOR ALL INSTANCES ]
[ ACTIVATION flag ].
```

```
CLASS lcl_rental DEFINITION.
...
PUBLIC SECTION.
METHODS add_vehicle FOR EVENT vehicle_created OF ....
ENDCLASS.
```

```
CLASS lcl_rental IMPLEMENTATION.
METHOD constructor.
...
```

4 SET HANDLER add_vehicle FOR ALL INSTANCES.
ENDMETHOD.
ENDCLASS.

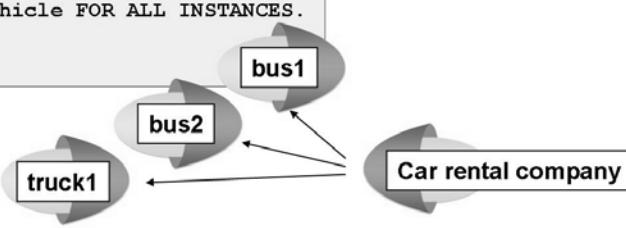


Figure 104: Registering Event Handling – Syntax

Events are registered using the SET HANDLER statement. Registration is only active at program runtime.

With instance events, FOR is followed by the reference to the object that triggers the event. Alternatively, you can also use the addition ALL INSTANCES. That way, you can also register objects that have not yet been created.

The addition ACTIVATION 'X' is optional during registration. To undo the registration, use ACTIVATION ''.

You can register several methods with one SET-HANDLER statement:

```
SET HANDLER
ref_handler_1->on_eventname_1
...
ref_handler_n->on_eventname_n
FOR ....
```

If several methods were registered to one event, the **sequence in which the event handler methods are called is not defined**, that is, there is no guaranteed sequence in which the event handler methods are called.

If a new event handler is registered within an event handler method for an event that has just been triggered, then this event handler is added to the end of the sequence and is then also executed when its turn comes. If an existing event handler is deregistered in an event handler method, then this handler is deleted from the event handler method sequence.

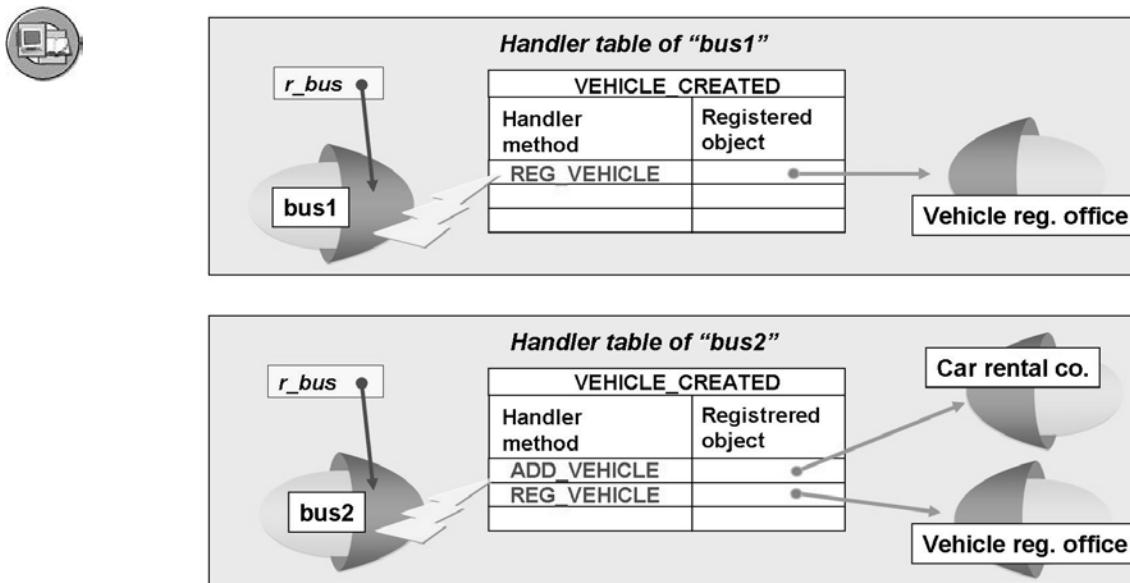


Figure 105: Registration/Deregistration: Handler Tables

Every object or class that has defined events has an internal table: the handler table. All handler methods that are registered to the various events are listed within the table. For instance methods, the handler table also contains references to the registered objects.

Objects that are registered for event handling are **not** deleted by the Garbage Collector if there are no remaining references to them.

Visibility Sections in Event Handling

Events are also subject to the visibility concept and can therefore be either public, protected, or private. Event handler methods also have visibility attributes.



- The **visibility of an event** determines **where** the event can be handled:

PUBLIC

All

PROTECTED

Can only be handled by users within that class or its subclasses

PRIVATE

Can only be handled within its class

- The **visibility of a handler method** controls **where** the registration of the method can be performed, that is, the locations where the SET HANDLER statement may be programmed.

PUBLIC

Anywhere in the program

PROTECTED

Can be handled by users within that class or its subclasses

PRIVATE

Can only be handled within its class

Event handler methods can only have the same visibility or more restricted visibility than the events they refer to.

Exercise 12: Events in Superclasses

Exercise Objectives

After completing this exercise, you will be able to:

- Define and trigger events
- Handle events
- Register event handling

Business Example

Airplane and vehicle references are to be included in the airline and the car rental company's lists. This process is to be event-controlled.

Task 1:

Define an event for the creation of an airplane. Trigger and handle it so that the reference to the airplane is entered into the airline's list of airplanes.

1. Complete your program ZBC401_##_MAIN or copy the program SAPBC401_INTS_MAIN_E (where ## is your two-digit group number)
2. Remove the calls of the method ADD_AIRPLANE from your main program.



Note: The entry of an airplane reference into the airline's list is to be event-controlled.

3. Identify the class that is suitable for triggering the event and the classes that should be used for handling the event. Use your UML diagram if necessary.
If applicable, illustrate the relationships in your UML diagram.
4. Define the public event AIRPLANE_CREATED and trigger it in the class LCL_AIRPLANE using a suitable method.
5. Change the method ADD_AIRPLANE in class LCL_CARRIER into a handler method for the event that you just defined. You will need to change both the signature and the implementation.
6. Register the new handler method, so that the airline enters **every airplane that was created after itself** in the list.



Note: Although this model is unrealistic, we will use it **for the time being**. A different rule for entering the airplane can be created later.

Continued on next page

7. Observe the execution of the program in the *ABAP Debugger*.
8. Where could the SET-HANDLER statement be executed in the example given? Could this also be done from the main program? What would the syntax have to be in this case?

Task 2:

Define an event for the creation of a vehicle. Trigger and handle it so that the reference to the vehicle is entered into the car rental company's list of vehicles.

1. Remove the calls of the method ADD_VEHICLE from your main program.
2. Define the event VEHICLE_CREATED and proceed as you did in the previous exercise.

Solution 12: Events in Superclasses

Task 1:

Define an event for the creation of an airplane. Trigger and handle it so that the reference to the airplane is entered into the airline's list of airplanes.

1. Complete your program ZBC401_##_MAIN or copy the program SAPBC401_INTS_MAIN_E (where ## is your two-digit group number)
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
 - b) Model solution: SAPBC401_EVES_MAIN_A
2. Remove the calls of the method ADD_AIRPLANE from your main program.



Note: The entry of an airplane reference into the airline's list is to be event-controlled.

- a) Speak to your instructor if you have any questions.
3. Identify the class that is suitable for triggering the event and the classes that should be used for handling the event. Use your UML diagram if necessary.
If applicable, illustrate the relationships in your UML diagram.
 - a) Speak to your instructor if you have any questions.
4. Define the public event AIRPLANE_CREATED and trigger it in the class LCL_AIRPLANE using a suitable method.
 - a) See the source code extract from the model solution.
5. Change the method ADD_AIRPLANE in class LCL_CARRIER into a handler method for the event that you just defined. You will need to change both the signature and the implementation.
 - a) See the source code extract from the model solution.
6. Register the new handler method, so that the airline enters **every airplane that was created after itself** in the list.



Note: Although this model is unrealistic, we will use it **for the time being**. A different rule for entering the airplane can be created later.

- a) See the source code extract from the model solution.

Continued on next page

7. Observe the execution of the program in the *ABAP Debugger*.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
8. Where could the SET-HANDLER statement be executed in the example given? Could this also be done from the main program? What would the syntax have to be in this case?
 - a) SET HANDLER r_vehicle->add_vehicle FOR ALL INSTANCES

Task 2:

Define an event for the creation of a vehicle. Trigger and handle it so that the reference to the vehicle is entered into the car rental company's list of vehicles.

1. Remove the calls of the method ADD_VEHICLE from your main program.
 - a) See the source code extract from the model solution.
2. Define the event VEHICLE_CREATED and proceed as you did in the previous exercise.
 - a) Speak to your instructor if you have any questions.
 - b) See the source code extract from the model solution.

Result

Source code extract:

SAPBC401_EVES_MAIN_A

```
REPORT  sapbc401_eves_main_a.

TYPE-POOLS icon.

INCLUDE sapbc401_ints_interface.
INCLUDE sapbc401_vehs_opt_event.
INCLUDE sapbc401_eves_a.
INCLUDE sapbc401_ints_agency.

DATA: r_vehicle TYPE REF TO lcl_vehicle,
      r_truck  TYPE REF TO lcl_truck,
      r_bus    TYPE REF TO lcl_bus,
```

Continued on next page

```

r_passenger TYPE REF TO lcl_passenger_plane,
r_cargo TYPE REF TO lcl_cargo_plane,
r_carrier TYPE REF TO lcl_carrier,
r_rental TYPE REF TO lcl_rental,
r_agency TYPE REF TO lcl_travel_agency.

START-OF-SELECTION.
*#####
***** Create TRAVEL_AGENCY *****
CREATE OBJECT r_agency EXPORTING im_name = 'Fly&Smile Travel'.

***** Create CARRIER *****
CREATE OBJECT r_carrier EXPORTING im_name = 'Smile&Fly Travel'.

***** Passenger Plane *****
* in the constructor of lcl_airplane an event is raised now !
* the passengerairplane is added automatically to lcl_carrier !
CREATE OBJECT r_passenger EXPORTING
    im_name = 'LH BERLIN'
    im_planetype = '747-400'
    im_seats = 345.

***** cargo Plane *****
* in the constructor of lcl_airplane an event is raised now !
* the cargoairplane is added automatically to lcl_carrier !

CREATE OBJECT r_cargo EXPORTING
    im_name = 'US Hercules'
    im_planetype = '747-500'
    im_cargo = 533.

***** insert business-parnter of agency into partner_list*****
r_agency->add_partner( r_carrier ).

***** create RENTAL *****
CREATE OBJECT r_rental EXPORTING im_name = 'HAPPY CAR RENTAL'.

***** create truck *****
CREATE OBJECT r_truck EXPORTING im_make = 'MAN'
    im_cargo = 45.

***** create truck *****

```

Continued on next page

```

CREATE OBJECT r_bus EXPORTING im_make = 'Mercedes'
          im_passengers = 80.

***** create truck *****
CREATE OBJECT r_truck EXPORTING im_make = 'VOLVO'
          im_cargo = 48.

***** insert business-parnter of agency into partner_list*****
r_agency->add_partner( r_rental ).

***** show attributes of all partners of travel_agency *****
r_agency->display_agency_partners( ).
```

SAPBC401_VEHs_OPT_EVENT

```

*-----*
* all classes of vehicle-submodel
* the class lcl_vehicle raises event vehicle_created
* *-----*
*-----*
*      CLASS lcl_vehicle DEFINITION
*-----*
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION.

"-----
METHODS: get_average_fuel IMPORTING im_distance TYPE s_distance
          im_fuel TYPE s_capacity
          RETURNING value(re_avgfuel) TYPE s_consum.

METHODS      constructor IMPORTING im_make TYPE string.
METHODS      display_attributes.
METHODS      set_make IMPORTING im_make TYPE string.
METHODS      get_make EXPORTING ex_make TYPE string.
CLASS-METHODS get_count EXPORTING re_count TYPE i.

EVENTS: vehicle_created.

PRIVATE SECTION.

"-----
DATA: make    TYPE string.

METHODS      init_make.
```

Continued on next page

```

CLASS-DATA:  n_o_vehicles TYPE i.

ENDCLASS.          "lcl_vehicle DEFINITION

*-----*
*      CLASS lcl_vehicle IMPLEMENTATION
*-----*

CLASS lcl_vehicle IMPLEMENTATION.

METHOD get_average_fuel.
    re_avgfuel = im_distance / im_fuel.
ENDMETHOD.          "get_average_fuel

METHOD constructor.
    make = im_make.
    ADD 1 TO n_o_vehicles.
    RAISE EVENT vehicle_created.
ENDMETHOD.          "constructor

METHOD set_make.
    IF im_make IS INITIAL.
        init_make( ).   " me->init_make( ). also possible
    ELSE.
        make = im_make.
    ENDIF.
ENDMETHOD.          "set_make

METHOD init_make.
    make = 'default make'.
ENDMETHOD.          "init_make

METHOD get_make.
    ex_make = make.
ENDMETHOD.          "get_make

METHOD display_attributes.
    WRITE: make.
ENDMETHOD.          "display_attributes

METHOD get_count.
    re_count = n_o_vehicles.
ENDMETHOD.          "get_count

```

Continued on next page

```

ENDCLASS.                                "lcl_vehicle IMPLEMENTATION

*-----*
*      CLASS lcl_truck DEFINITION
*-----*
...
*-----*
*      CLASS lcl_bus DEFINITION
*-----*
*
*-----*
...
*-----*
*      CLASS lcl_rental DEFINITION
*-----*
CLASS lcl_rental DEFINITION.

PUBLIC SECTION.
  "
METHODS:      constructor IMPORTING im_name TYPE string.
METHODS       add_vehicle FOR EVENT vehicle_created OF lcl_vehicle
              IMPORTING sender.
METHODS       display_attributes.
INTERFACES:  lif_partners.

PRIVATE SECTION.
  "
DATA: name TYPE string,
      vehicle_list TYPE TABLE OF REF TO lcl_vehicle.
ENDCLASS.                                "lcl_rental DEFINITION

*-----*
*      CLASS lcl_rental IMPLEMENTATION
*-----*
CLASS lcl_rental IMPLEMENTATION.

METHOD lif_partners~display_partner.
  display_attributes( ).
ENDMETHOD.                               "lif_partners~display_partner

METHOD  constructor.

```

Continued on next page

```

        name = im_name.
      SET HANDLER add_vehicle FOR ALL INSTANCES.
ENDMETHOD.                      "constructor

METHOD add_vehicle.
  APPEND sender TO vehicle_list.
ENDMETHOD.                      "add_vehicle

METHOD display_attributes.
  DATA: r_vehicle TYPE REF TO lcl_vehicle.
  WRITE: / icon_transport_proposal AS ICON, name.
  WRITE: ' Here comes the vehicle list: '. ULINE. ULINE.
  LOOP AT vehicle_list INTO r_vehicle.
    r_vehicle->display_attributes( ).
  ENDLOOP.
ENDMETHOD.                      "display_attributes

ENDCLASS.                      "lcl_rental IMPLEMENTATION

*-----*
*   CLASS lcl_travel_agency DEFINITION
*-----*
...

```

SAPBC401_EVES_A

```

*&-----*
*&   Include          SAPBC401_EVES_A           *
*&-----*
*-----*
*   CLASS lcl_airplane DEFINITION           *
*-----*
CLASS lcl_airplane DEFINITION.

PUBLIC SECTION.
"-----
CONSTANTS: pos_1 TYPE i VALUE 30.

METHODS: constructor IMPORTING
         im_name      TYPE string
         im_planetype TYPE saplane-planetype,

```

Continued on next page

```

        display_attributes.

CLASS-METHODS: class_constructor.
CLASS-METHODS: display_n_o_airplanes.
EVENTS airplane_created.

PRIVATE SECTION.
-----+
CLASS-METHODS: get_technical_attributes
    IMPORTING im_type      TYPE saplane-planetype
    EXPORTING ex_weight    TYPE s_plan_wei
                  ex_tankcap  TYPE s_capacity.

DATA: name          TYPE string,
      planetype   TYPE saplane-planetype.

CLASS-DATA: n_o_airplanes TYPE i.
CLASS-DATA: list_of_planetypes TYPE z_00_planetype.

ENDCLASS.           "lcl_airplane DEFINITION
*-----+
*      CLASS lcl_airplane IMPLEMENTATION
*-----+
CLASS lcl_airplane IMPLEMENTATION.

METHOD class_constructor.
    SELECT * FROM saplane INTO TABLE list_of_planetypes.
ENDMETHOD.          "constructor

METHOD constructor.
    name          = im_name.
    planetype    = im_planetype.
    n_o_airplanes = n_o_airplanes + 1.
    RAISE EVENT airplane_created.
ENDMETHOD.          "constructor

METHOD display_attributes.
    DATA: weight TYPE saplane-weight,
          cap   TYPE saplane-tankcap.
    WRITE: / icon_ws_plane AS ICON,
           / 'Name of Airplane:'(001), AT pos_1 name,

```

Continued on next page

```

        / 'Type of airplane: '(002), AT pos_1 planetype.
get_technical_attributes( EXPORTING im_type = planetype
                           IMPORTING ex_weight = weight
                           ex_tankcap = cap ).
WRITE: / 'Weight:'(003), weight,
       'Tankcap:'(004), cap.
ENDMETHOD.                      "display_attributes

METHOD display_n_o_airplanes.
WRITE: /, / 'Number of airplanes: '(ca1),
       AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
ENDMETHOD.                      "display_n_o_airplanes

METHOD get_technical_attributes.
DATA: wa TYPE saplane.

READ TABLE list_of_planetypes INTO wa
      WITH TABLE KEY planetype = im_type
            TRANSPORTING weight tankcap.
ex_weight = wa-weight.
ex_tankcap = wa-tankcap.
ENDMETHOD.                      "get_technical_attributes

ENDCLASS.                         "lcl_airplane IMPLEMENTATION

*-----*
*   CLASS lcl_cargo_plane DEFINITION
*-----*
...
*-----*
*   CLASS lcl_passenger_plane DEFINITION
*-----*
*
*-----*
...
*-----*
*   CLASS lcl_carrier DEFINITION
*-----*
CLASS lcl_carrier DEFINITION.

PUBLIC SECTION.
"-----"

```

Continued on next page

```

INTERFACES lif_partners.

METHODS: constructor IMPORTING im_name TYPE string,
          get_name RETURNING value(ex_name) TYPE string,
          add_airplane FOR EVENT airplane_created OF lcl_airplane
                         IMPORTING sender,
          display_airplanes,
          display_attributes.

PRIVATE SECTION.

"-----
DATA: name           TYPE string,
      airplane_list TYPE TABLE OF REF TO lcl_airplane.
ENDCLASS.           "lcl_carrier DEFINITION

*-----*
*      CLASS lcl_carrier IMPLEMENTATION
*-----*

CLASS lcl_carrier IMPLEMENTATION.

METHOD lif_partners~display_partner.
  display_airplanes( ).
ENDMETHOD.           "lif_partners~display_partner

METHOD add_airplane.
  APPEND sender TO airplane_list.
ENDMETHOD.           "add_airplane

METHOD display_attributes.
  WRITE: icon_flight AS ICON, name . ULINE. ULINE.
  display_airplanes( ).
ENDMETHOD.           "display_attributes

METHOD display_airplanes.
  DATA: r_plane TYPE REF TO lcl_airplane.
  LOOP AT airplane_list INTO r_plane.
    r_plane->display_attributes( ).
  ENDLOOP.
ENDMETHOD.           "display_airplanes

METHOD constructor.
  SET HANDLER add_airplane FOR ALL INSTANCES.
  name = im_name.
ENDMETHOD.           "constructor

```

Continued on next page

```
METHOD get_name.  
    ex_name = name.  
ENDMETHOD.  
                                "get_name  
  
ENDCLASS.                      "lcl_carrier IMPLEMENTATION
```

SAPBC401_INTS_INTERFACE

```
*&-----*  
*&   Include          SAPBC401_INTS_INTERFACE  
*&-----*  
  
* This INTERFACE will be used to access services of classes  
* LCL_CARRIER, LCL_RENTAL, LCL_HOTEL, ...  
* The client to use these services will be class LCL_TRAVEL_AGENCY  
*  
INTERFACE lif_partners.  
METHODS: display_partner.  
ENDINTERFACE.                  "lif_partners
```


Exercise 13: Events in Interfaces (Optional)

Exercise Objectives

After completing this exercise, you will be able to:

- Define events in interfaces
- Trigger interface events in implementing classes
- Handle interface events
- Register event handling

Business Example

Airline and car rental company references are to be entered into the travel agent's list. This process is to be event-controlled. Be sure to create your program so that it can be easily extended to manage additional business partners of the travel agent in the future.

Task:

Define an event for the creation of a business partner. Trigger and handle it so that the reference to the business partner is entered into the travel agent's list of partners.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise (where ## is your two-digit group number).)
2. Remove the calls of the method ADD_PARTNER from your main program.



Note: The entry of a business partner reference into the travel agent's list is to be event-controlled.

3. If necessary, examine your UML diagram. Which class or interface should define the event? Which class or interface should trigger it? Which class or interface should handle it?
If applicable, illustrate the relationships in your UML diagram.
4. Define the event PARTNER_CREATED and trigger it using a suitable method in the class that you selected.
5. Change the method ADD_PARTNER in the class LCL_TRAVEL_AGENCY into a handler method for the event that you just defined. You will need to change both the signature and the implementation.

Continued on next page

6. Register the new handler method, so that the travel agency enters **every business partner that was created after itself** into the list.

→ **Note:** Although this model is unrealistic, we will use it **for the time being**. A different rule for entering the business partner can be created later.
7. Observe the execution of the program in the *ABAP Debugger*.

Solution 13: Events in Interfaces (Optional)

Task:

Define an event for the creation of a business partner. Trigger and handle it so that the reference to the business partner is entered into the travel agent's list of partners.

1. Complete your ZBC401_##_MAIN program or copy the model solution from the previous exercise (where ## is your two-digit group number).
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
 - b) Model solution: SAPBC401_EVES_MAIN_B
2. Remove the calls of the method ADD_PARTNER from your main program.



Note: The entry of a business partner reference into the travel agent's list is to be event-controlled.

- a) See the source code extract from the model solution
3. If necessary, examine your UML diagram. Which class or interface should define the event? Which class or interface should trigger it? Which class or interface should handle it?

If applicable, illustrate the relationships in your UML diagram.

 - a) Speak to your instructor if you have any questions.
4. Define the event PARTNER_CREATED and trigger it using a suitable method in the class that you selected.
 - a) See the source code extract from the model solution.
5. Change the method ADD_PARTNER in the class LCL_TRAVEL_AGENCY into a handler method for the event that you just defined. You will need to change both the signature and the implementation.
 - a) See the source code extract from the model solution.

Continued on next page

6. Register the new handler method, so that the travel agency enters **every business partner that was created after itself** into the list.

→ **Note:** Although this model is unrealistic, we will use it **for the time being**. A different rule for entering the business partner can be created later.

 - a) See the source code extract from the model solution.
7. Observe the execution of the program in the *ABAP Debugger*.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.

Result

Source code extract:

SAPBC401_EVES_MAIN_B

```

REPORT  sapbc401_eves_main_b.
*&-----*
*&     Implement events in lcl_vehicle and lcl_airplane      *
*&     Implement events in lcl_carrier and lcl_rental        *
*&     No "add" methods are needed anymore in the main program !   *
*&     Vehicles and Airplanes are added automatically          *
*&-----*
TYPE-POOLS icon.

INCLUDE sapbc401_ints_interface_event.
INCLUDE sapbc401_vehs_event.
INCLUDE sapbc401_eves_b.
INCLUDE sapbc401_ints_agency_handler.

DATA: r_vehicle TYPE REF TO lcl_vehicle,
      r_truck TYPE REF TO lcl_truck,
      r_bus    TYPE REF TO lcl_bus,
      r_passenger TYPE REF TO lcl_passenger_plane,
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_carrier TYPE REF TO lcl_carrier,
      r_rental TYPE REF TO lcl_rental,
      r_agency TYPE REF TO lcl_travel_agency.

```

Continued on next page

```

START-OF-SELECTION.

***** create travel_agency *****
CREATE OBJECT r_agency EXPORTING im_name = 'Fly&Smile Travel'.


***** create rental *****
CREATE OBJECT r_rental EXPORTING im_name = 'HAPPY CAR RENTAL'.


***** create truck *****
CREATE OBJECT r_truck EXPORTING im_make = 'MAN'
                      im_cargo = 45.

***** create truck *****
CREATE OBJECT r_bus EXPORTING im_make = 'Mercedes'
                      im_passengers = 80.

***** create truck *****
CREATE OBJECT r_truck EXPORTING im_make = 'VOLVO'
                      im_cargo = 48.

***** Create CARRIER *****
CREATE OBJECT r_carrier EXPORTING im_name = 'Smile&Fly Travel'.


***** Passenger Plane *****
CREATE OBJECT r_passenger EXPORTING
                      im_name = 'LH BERLIN'
                      im_planetype = '747-400'
                      im_seats = 345.

***** cargo Plane *****
CREATE OBJECT r_cargo EXPORTING
                      im_name = 'US Hercules'
                      im_planetype = '747-500'
                      im_cargo = 533.

***** show attributes of all partners of travel_agency *****
r_agency->display_agency_partners( ).
```

SAPBC401_INTS_INTERFACE_EVENT

```

*&-----*
*&   Include      SAPBC401_INTS_INTERFACE_EVENT      *
*&-----*
*-----*
```

Continued on next page

```

* an event is defined inside he interface
*-----
INTERFACE lif_partners.
METHODS display_partner.
*** the event partner_created can be raised by classes that implement the interface!
*** this also shows the solution of the optional events excercise
*** if you solved this additional optional excercise you do not need to
*** call the add_partner methods anymore - congratulation!
EVENTS: partner_created.
ENDINTERFACE.          "lif_partners

```

SAPBC401_VEHS_EVENT

```

*&-----
*&  Include      SAPBC401_VEHS_EVENT
*&-----
* the definition of LCL_VEHICLE, LCL_TRUCK, LCL_BUS is left out here
* this has been shown several times before

*-----
*      CLASS lcl_rental DEFINITION
*-----
*
*-----
CLASS lcl_rental DEFINITION.

PUBLIC SECTION.
"-----
METHODS:   constructor IMPORTING im_name TYPE string.
METHODS     add_vehicle for event vehicle_created of lcl_vehicle
           importing sender.
METHODS     display_attributes.
INTERFACES: lif_partners.
PRIVATE SECTION.
"-----
DATA: name TYPE string,
      vehicle_list TYPE TABLE OF REF TO lcl_vehicle.
ENDCLASS.          "lcl_rental DEFINITION

*-----
*      CLASS lcl_rental IMPLEMENTATION

```

Continued on next page

```

*-----*
*-----*
*-----*
CLASS lcl_rental IMPLEMENTATION.
METHOD lif_partners~display_partner.
    display_attributes( ).
ENDMETHOD.                      "lif_partners~display_partner

METHOD constructor.
    name = im_name.
    SET HANDLER add_vehicle FOR ALL INSTANCES.
    RAISE EVENT lif_partners~partner_created.
ENDMETHOD.                      "constructor

METHOD add_vehicle.
    APPEND sender TO vehicle_list.
ENDMETHOD.                      "add_vehicle

METHOD display_attributes.
    DATA: r_vehicle TYPE REF TO lcl_vehicle.
    skip 2.
    WRITE: / icon_transport_proposal AS ICON, name.
    WRITE: ' Here comes the vehicle list: '. ULINE. ULINE.
    LOOP AT vehicle_list INTO r_vehicle.
        r_vehicle->display_attributes( ).
    ENDLOOP.
ENDMETHOD.                      "display_attributes

ENDCLASS.                      "lcl_rental IMPLEMENTATION

```

SAPBC401_EVES_B

```

*&-----*
*&   Include      SAPBC401_EVES_B
*&-----*
*-----*
*      CLASS lcl_carrier DEFINITION
*-----*
CLASS lcl_carrier DEFINITION.

```

Continued on next page

```

PUBLIC SECTION.

"-----
INTERFACES lif_partners.
METHODS: constructor IMPORTING im_name TYPE string,
          get_name RETURNING value(ex_name) TYPE string,
          add_airplane FOR EVENT airplane_created OF lcl_airplane
                      IMPORTING sender,
          display_airplanes,
          display_attributes.

PRIVATE SECTION.

"-----
DATA: name           TYPE string,
      airplane_list TYPE TABLE OF REF TO lcl_airplane.
ENDCLASS.           "lcl_carrier DEFINITION

*-----*
*      CLASS lcl_carrier IMPLEMENTATION
*-----*
CLASS lcl_carrier IMPLEMENTATION.

METHOD lif_partners~display_partner.
  display_attributes( ).
ENDMETHOD.           "lif_partners~display_partner

METHOD add_airplane.
  APPEND sender TO airplane_list.
ENDMETHOD.           "add_airplane

METHOD display_attributes.
  SKIP 2.
  WRITE: icon_flight AS ICON, name . ULINE. ULINE.
  display_airplanes( ).
ENDMETHOD.           "display_attributes

METHOD display_airplanes.
  DATA: r_plane TYPE REF TO lcl_airplane.
  LOOP AT airplane_list INTO r_plane.
    r_plane->display_attributes( ).
  ENDLOOP.
ENDMETHOD.           "display_airplanes

METHOD constructor.

```

Continued on next page

```

        name = im_name.
        SET HANDLER add_airplane FOR ALL INSTANCES.
        RAISE EVENT lif_partners~partner_created.
ENDMETHOD.                                     "constructor

METHOD get_name.
    ex_name = name.
ENDMETHOD.                                     "get_name

ENDCLASS.                                       "lcl_carrier IMPLEMENTATION

```

SAPBC401_INTS_AGENCY_HANDLER

```

*&-----*
*&   Include          SAPBC401_INTS_AGENCY_HANDLER      *
*&-----*
*-----*
*      CLASS lcl_travel_agency DEFINITION
*-----*
CLASS lcl_travel_agency DEFINITION.

PUBLIC SECTION.
"-----
METHODS:   constructor IMPORTING im_name TYPE string.
METHODS     add_partner FOR EVENT partner_created OF lif_partners
           IMPORTING sender.
METHODS     display_agency_partners.

PRIVATE SECTION.
"-----
DATA: name TYPE string,
      partner_list TYPE TABLE OF REF TO lif_partners.
ENDCLASS.                                     "lcl_travel_agency DEFINITION

*-----*
*      CLASS lcl_travel_agency IMPLEMENTATION
*-----*
*-----*
*-----*
CLASS lcl_travel_agency IMPLEMENTATION.
METHOD display_agency_partners.
DATA: r_partner TYPE REF TO lif_partners.

```

Continued on next page

```
        WRITE: icon_dependents AS ICON, name.  
        WRITE: ' Here are the partners of the travel agency: '.ULINE.ULINE.  
        LOOP AT partner_list INTO r_partner.  
          r_partner->display_partner( ).  
        ENDLOOP.  
      ENDMETHOD.                      "display_agency_partners  
  
METHOD constructor.  
  name = im_name.  
  SET HANDLER add_partner FOR ALL INSTANCES.  
ENDMETHOD.                      "constructor  
  
METHOD add_partner.  
  APPEND sender TO partner_list.  
ENDMETHOD.                      "add_partner  
  
ENDCLASS.                        "lcl_travel_agency IMPLEMENTATION
```



Lesson Summary

You should now be able to:

- Define and trigger events
- Handle events
- Register and deregister event handling
- Explain the key differences between explicit method calls and event-controlled method calls

Related Information

For more information, refer to the SAP Library.



Unit Summary

You should now be able to:

- Define inheritance relationships between classes
- Redefine methods
- Create up-cast assignments (Widening Cast)
- Create down-cast assignments (Narrowing Cast)
- Explain the concept of polymorphism with reference to inheritance
- Use cast assignments with inheritance to make generic calls
- Define and implement interfaces
- Implement interface methods
- Use interface references to make up-cast assignments
- Use interface references to make down-cast assignments
- Explain the term polymorphism with reference to interfaces
- Use cast assignments with interfaces to make generic calls
- Define and trigger events
- Handle events
- Register and deregister event handling
- Explain the key differences between explicit method calls and event-controlled method calls



Test Your Knowledge

1. What is the ABAP syntax required to make a local subclass inherit from a superclass?

2. What is the ABAP syntax required to redefine an inherited method in a local class?

3. Suppose you are copying a subclass reference to a reference variable that is typed to the superclass (up-cast). What components can you access with this reference variable?

Choose the correct answer(s).

- A Redefined components of the superclass
- B Newly defined components of the subclass
- C Inherited components of the superclass
- D Redefined components of the subclass

4. Assume that a reference variable typed on a superclass contains a subclass reference and you copy this to a reference variable that is typed on the class (down-cast). Which of the following components can you access with this reference variable?

Choose the correct answer(s).

- A Redefined components of the superclass
- B Newly defined components of the subclass
- C Inherited components of the superclass
- D Redefined components of the subclass

5. For what purpose do you use inheritance?

6. Does an interface have an implementation part?

7. Suppose that you copy an instance reference of a class that implements an interface to a reference variable that is typed on the interface (up-cast). What components can you access with this reference variable?

Choose the correct answer(s).

- A The components of the interface
- B The components of the class that are not defined in the interface
- C All components of the class
- D The components of the interface for which alias names have been defined

8. Suppose that a reference variable that is typed on an interface contains an instance reference of a class that implements this interface and you copy this to a reference variable that is typed on the class (down cast). Which of the following components can you access with this reference variable?

Choose the correct answer(s).

- A The components of the interface
- B The components from the class that are not defined on the interface
- C All components of the class
- D The components of the interface for which alias names have been defined

9. What is the statement for defining events?

10. What is the statement for triggering events?

11. With which statement would you define a handler method M_H for event E of class C?

12. With which statement would you register handler method M_H of the reacting instance REF_H with the triggering instance REF_R?

13. Can events be defined in interfaces?

14. Can events be triggered in interfaces?



Answers

1. What is the ABAP syntax required to make a local subclass inherit from a superclass?

Answer: the INHERITING FROM addition to the CLASS-DEFINITION statement.

2. What is the ABAP syntax required to redefine an inherited method in a local class?

Answer: the REDEFINITION addition to the METHODS statement.

3. Suppose you are copying a subclass reference to a reference variable that is typed to the superclass (up-cast). What components can you access with this reference variable?

Answer: A, C

4. Assume that a reference variable typed on a superclass contains a subclass reference and you copy this to a reference variable that is typed on the class (down-cast). Which of the following components can you access with this reference variable?

Answer: A, B, C

5. For what purpose do you use inheritance?

Answer: Inheritance is how generalization/specialization relationships between classes are implemented in a program.

6. Does an interface have an implementation part?

Answer: No

Refer to the relevant section of the lesson.

7. Suppose that you copy an instance reference of a class that implements an interface to a reference variable that is typed on the interface (up-cast). What components can you access with this reference variable?

Answer: A

Refer to the relevant section of the lesson.

8. Suppose that a reference variable that is typed on an interface contains an instance reference of a class that implements this interface and you copy this to a reference variable that is typed on the class (down cast). Which of the following components can you access with this reference variable?

Answer: A, B, C, D

Refer to the relevant section of the lesson.

9. What is the statement for defining events?

Answer: EVENTS

Refer to the relevant section of the lesson.

10. What is the statement for triggering events?

Answer: RAISE EVENT

Refer to the relevant section of the lesson.

11. With which statement would you define a handler method M_H for event E of class C?

Answer: METHODS m_h FOR EVENT e OF c

Refer to the relevant section of the lesson.

12. With which statement would you register handler method M_H of the reacting instance REF_H with the triggering instance REF_R?

Answer: `SET HANDLER ref_h->m_h FOR ref_r.`

Refer to the relevant section of the lesson.

13. Can events be defined in interfaces?

Answer: Yes

14. Can events be triggered in interfaces?

Answer: No

Refer to the relevant section of the lesson.

Unit 3

Object-Oriented Repository Objects

Unit Overview

The first lesson will introduce you to the *Class Builder*, a tool from the *ABAP Workbench* that you have not used before. You will focus on learning how to apply the techniques that you learned for local classes and interfaces to global classes and interfaces. A focal point in this lesson is the description of how object-oriented ABAP programming can be used in various ways, as illustrated using the example of SAP Grid Control and BAdIs.

We will add a few more details (which were deliberately left out earlier) to your understanding of object-oriented programming concepts. These apply equally to local and global classes.

The third lesson is about possibilities for generating very specific global classes. In turn, this generation tool uses the *Class Builder*.



Unit Objectives

After completing this unit, you will be able to:

- Describe the functions of the Class Builder
- Create global classes using the Class Builder
- Create interfaces using the Class Builder
- Reference global classes and interfaces in other Repository objects
- Define abstract classes
- Define abstract methods
- Define final classes
- Define final methods
- Limit the visibility of the constructor
- Define friendship relationships between classes
- Explain the “singleton pattern”

Unit Contents

Lesson: Global Classes and Interfaces	227
Procedure: Importing Local Classes and Interfaces	235
Procedure: Moving the Method Definition of a Global Class to an Implemented Interface	244
Exercise 14: Global Classes	245
Exercise 15: Global Interfaces	249
Exercise 16: (Optional) Refactoring Assistant	261
Lesson: Special Object-Oriented Programming Techniques	266
Exercise 17: Singleton Classes (Optional)	273
Exercise 18: Friendship Relationships (Optional)	277

Lesson: Global Classes and Interfaces

Lesson Overview

This lesson introduces you to another tool in the *ABAP Workbench*: The *Class Builder*, which was introduced into SAP R/3 4.6A at the same time as ABAP Objects. We will examine its main functions and its integration into the *ABAP Workbench*, and in particular, the Object Navigator.



Lesson Objectives

After completing this lesson, you will be able to:

- Describe the functions of the Class Builder
- Create global classes using the Class Builder
- Create interfaces using the Class Builder
- Reference global classes and interfaces in other Repository objects

Business Example

You want to develop your classes and interfaces as actively integrated Repository objects.

Creating Global Classes and Interfaces

As with subroutines, **local** classes or interfaces can only be used within the program in which they are defined and implemented. The CLASS statement is a local, declarative statement in the program. Just as the TYPES statement defines local data types, the CLASS statement defines local object types.

In both cases, whether or not the source text is stored separately in include programs is irrelevant.

On the other hand, global classes or global interfaces are individual Repository objects with all of the normal attributes (active integration, versioning, transport system, and so on). The namespace convention (Y*, Z*, or a special customer namespace) is the same as that used for the namespace of other Repository objects.

Therefore, a special maintenance tool is available in the *ABAP Workbench* as of SAP R/3 4.6A: the **Class Builder**.

→ **Note:** A large number of screenshots and screen illustrations will be used in the following lesson. Note that the appearance of some of the icons or menus depends on the release level.

Where doubts arise, you are advised to use the ScreenTip (explanatory text that appears when you place the cursor over an icon and leave it there for a short time).

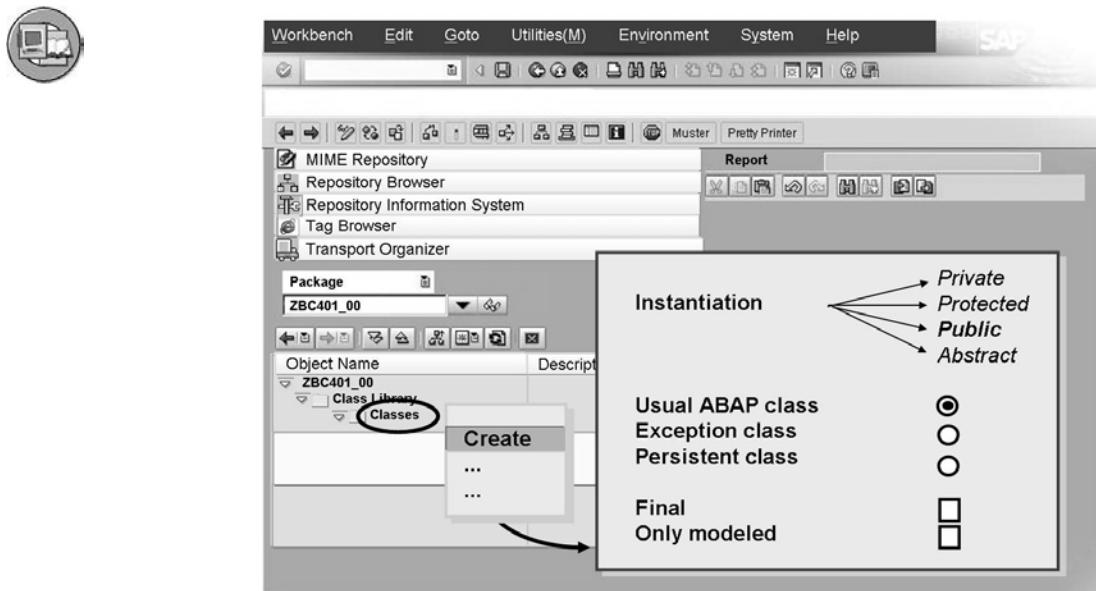


Figure 106: Creating Global Classes in the Object Navigator

As with other Repository objects, the separate navigation area of the **Object Navigator** makes it the ideal development tool for all Repository objects. It also supports the Class Builder. As with other Repository objects, the easiest way to create a new global class is to use the context menu in the navigation area: First select the package node or select the class node itself within a package.

A dialog box asks you to make further attributes for the new class: Do not change the default attributes at this point. You will learn about the different settings and their effects later.

Do the same when creating global classes.

The global class or global interface is then displayed in the Class Builder table in the editor area of the Object Navigator.

As has already been mentioned, some of the icons and menus could be different depending on the release level. When you are learning to work in the Class Builder, you are therefore advised to use the ScreenTip.

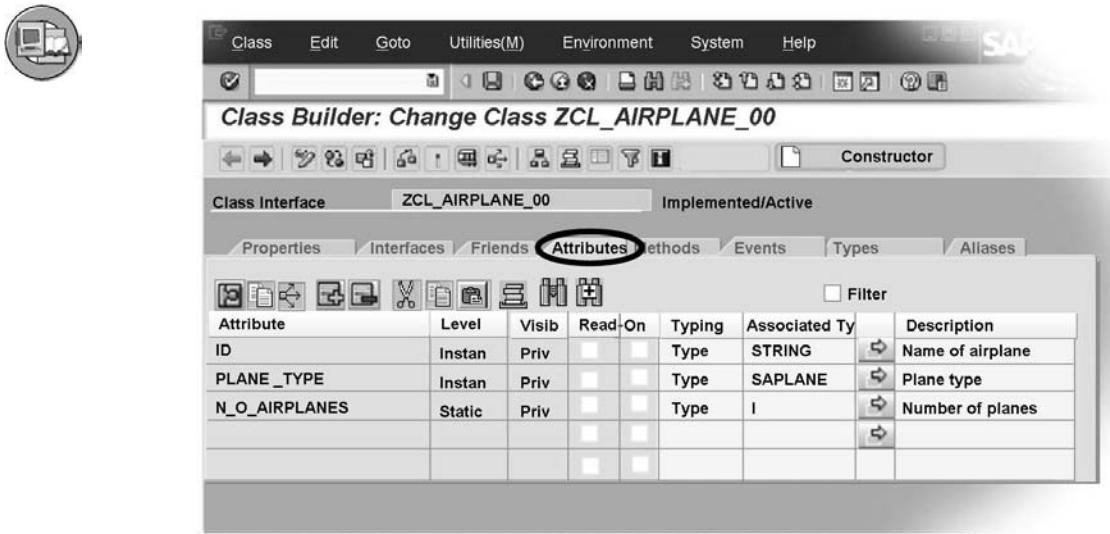


Figure 107: Definition of Attributes

Choose the *Attributes* tab to open the list of all attribute definitions in the class. You can define new attributes here.

You can use the *input help* when you are defining the type attributes. Remember to use meaningful short descriptions.

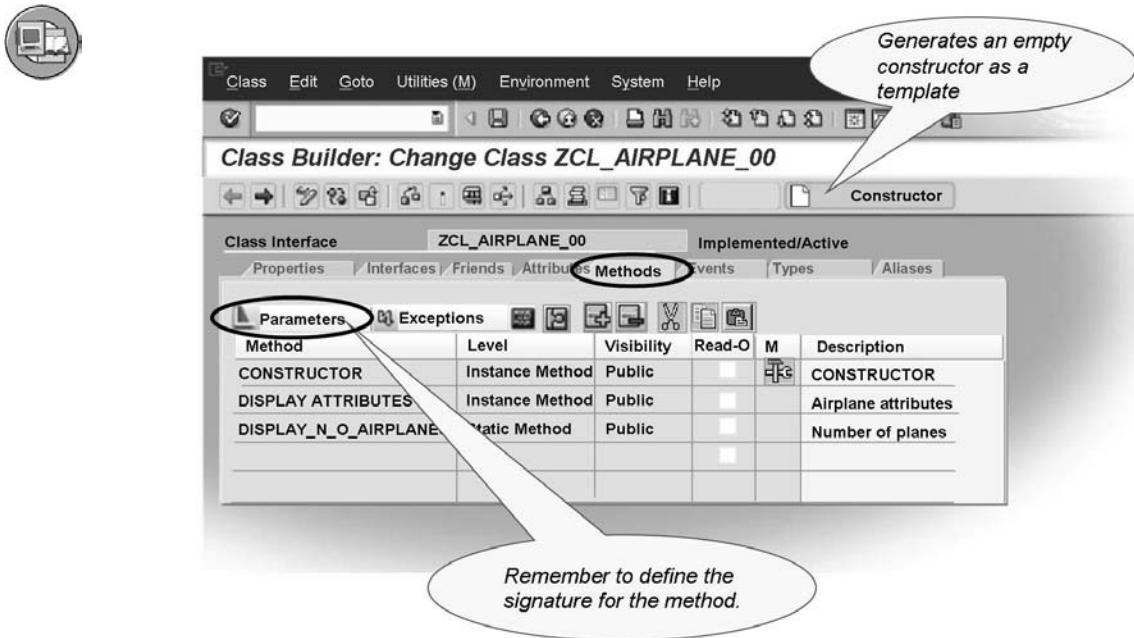


Figure 108: Definition of Methods

Choose the *Methods* tab to open the list of all method definitions in the class. You can define new methods here. You can use the *input help* when defining the attributes. Remember to use meaningful short descriptions.

There are separate editor windows for the signature and the implementation.

Choose the *Constructor* button to define an instance constructor. The constructor name is chosen automatically and the selection possibilities in the editor window for the signature are restricted accordingly.

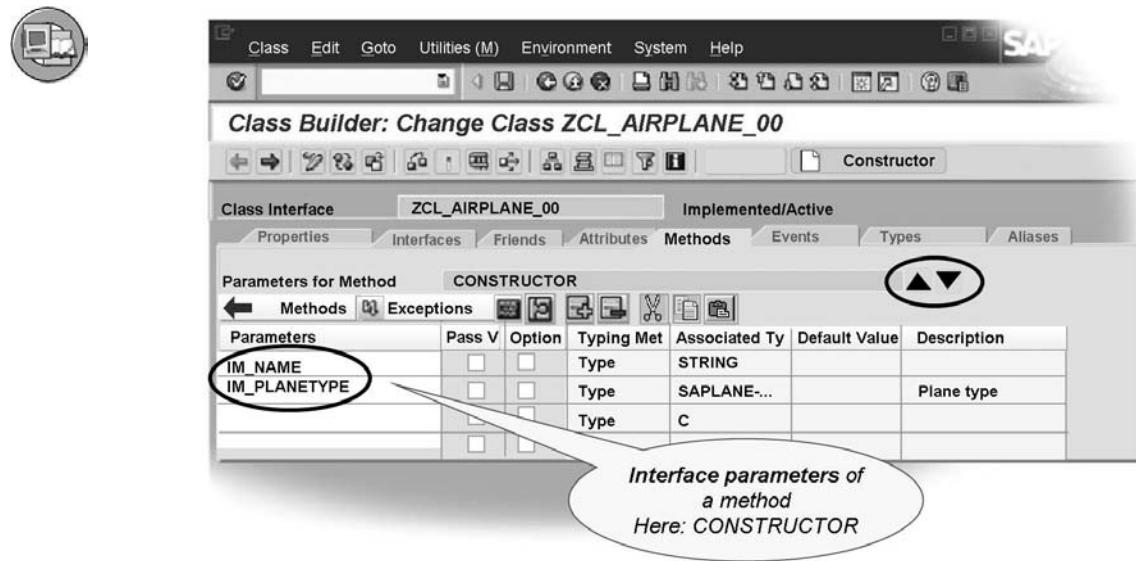


Figure 109: Definition of Method Signatures

In the method list, select a method and choose the *Parameter* button to go to signature maintenance. You can define new formal parameters here.

You can use the *input help* when defining the attributes. Remember to use meaningful short descriptions.

You can scroll between the signatures using the *Previous Method* or *Next Method* buttons. Choose the *Methods* button to return to the method list.

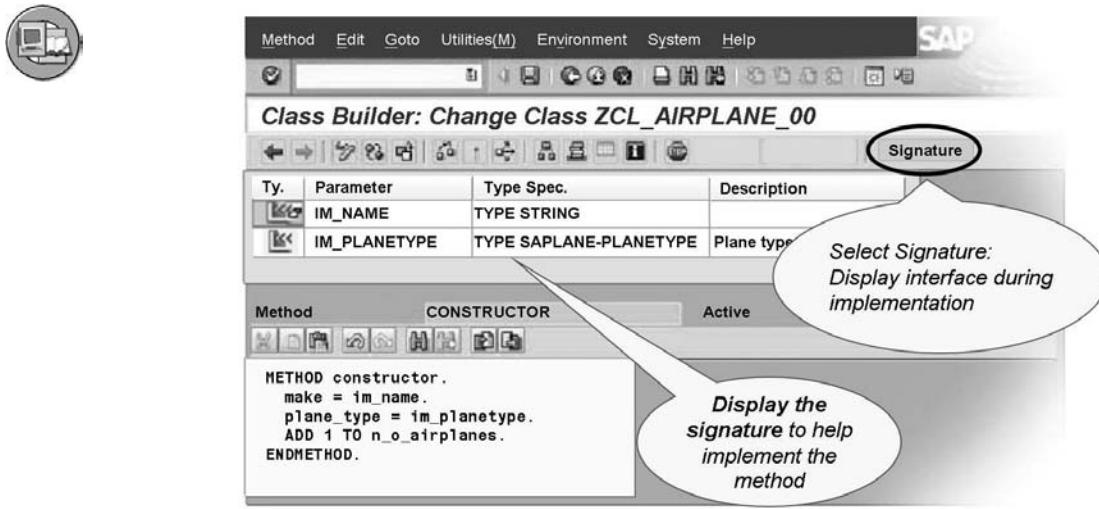


Figure 110: Implementation of Methods

In the method list, select a method (double-click) or, if a method is already selected, choose the *Source code* button to go to the source text maintenance. You can implement the modifications here.

Hint: You can display the method's signature by choosing the relevant button.

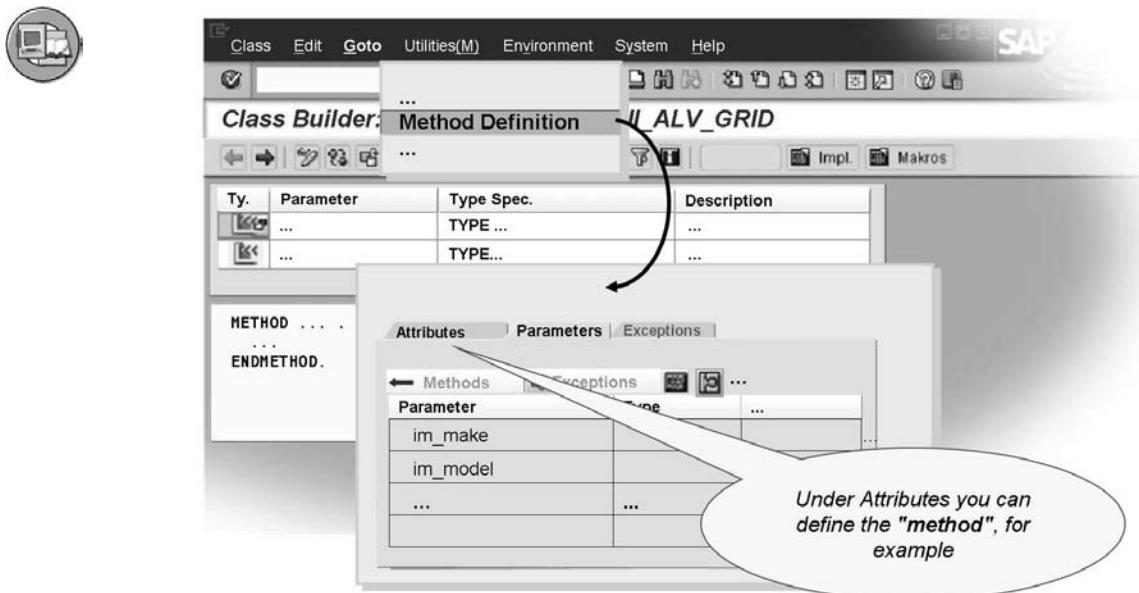


Figure 111: Displaying the Method Definition

Choose *Goto → Method definition* if you want to change the attributes of your method during the implementation. You can define a handler method here, for example. You do this using the “Properties” tab in the dialog box.



Figure 112: Definition of Components Using the Navigation Area

You can also define attributes, methods, or events in the context menu in the Object Navigator's navigation area. The properties are then maintained in a dialog box, and not in the table we saw previously.



Hint: Print selected portions of the source text using *Class → Print* or *Method → Print*.

You create global interfaces in a similar way to creating global classes. Proceed as normal, using the right mouse button in the Object Navigator. The naming convention is: “IF_” for SAP interfaces and “ZIF_ or YIF_” for user-defined interfaces.

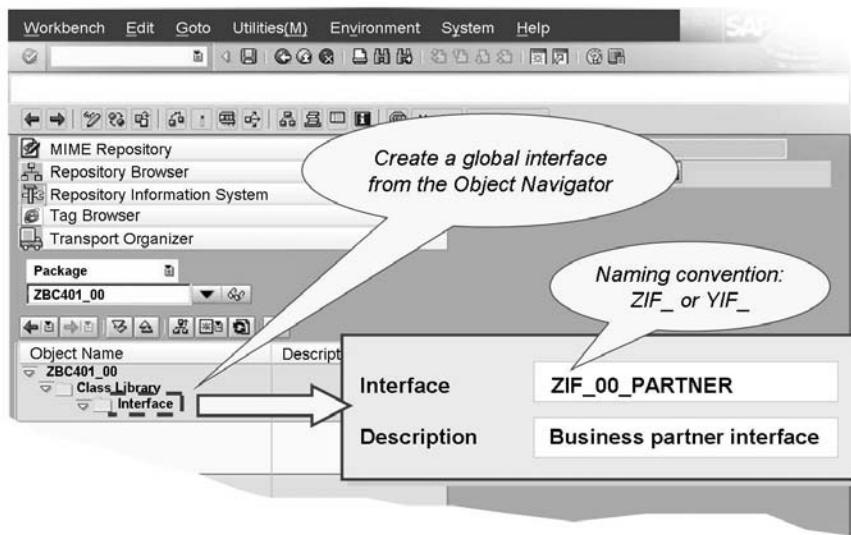


Figure 113: Defining Global Interfaces

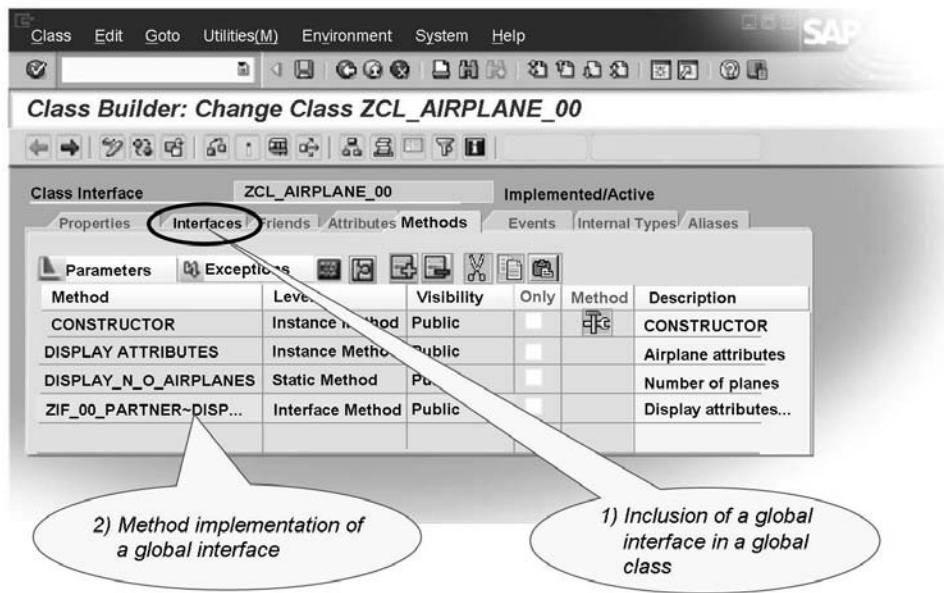


Figure 114: Including Global Interfaces

If you want to include a global interface in your global class, you must enter the name of the interface in the *Interfaces* tab. Once you have done that, all of the interface's components automatically appear under the relevant tabs according to the naming

convention and the interface resolution operator. In the example, the global interface ZIF_00_PARTNER is incorporated using the DISPLAY_PARTNER method. You can implement the method simply by double-clicking on the method name.

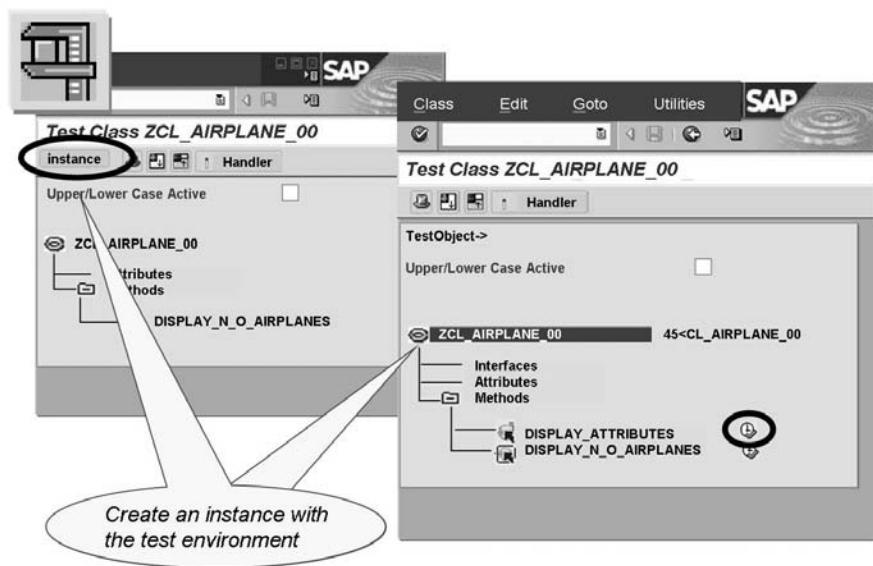


Figure 115: The Class Builder Testing Environment

You can test active global classes:

The class's templates are allocated temporarily. Static ones are allocated immediately, while instance components are allocated when you choose the *Instance* button.

The system only lists the public components. Methods can be tested using the *Execute Method* icon.

You can test the triggering of events in a class as follows:

1. Select an event.
2. Choose *Handler*. This registers a standard method for the event.
3. Call a method in which the event trigger was implemented.

The event that was triggered and all of the exported actual parameters are displayed in a list.



Importing Local Classes and Interfaces

Use

The procedure is an easy way to make **global copies** of local classes and local interfaces.



Hint: This function **cannot** be used from within OBJECT-NAVIGATOR.

Procedure

1. In the *SAP Easy Access Menu*, choose *Tools* → *ABAP Workbench* → *Development* → *Class Builder* or call transaction SE24.
2. From the initial screen of SE24, select *Object type* → *Import* → *Local program classes*.
3. Enter the name of the main program and, if the local classes and interfaces were defined within include programs, select *Expand Includes*.
4. Choose *Display Classes/Interfaces*.
5. Enter names for the global classes and interfaces that you want to create. Remember the customer namespace, if applicable.
6. Select the global classes and interfaces that you want to create and press the *Import* button.

Continued on next page

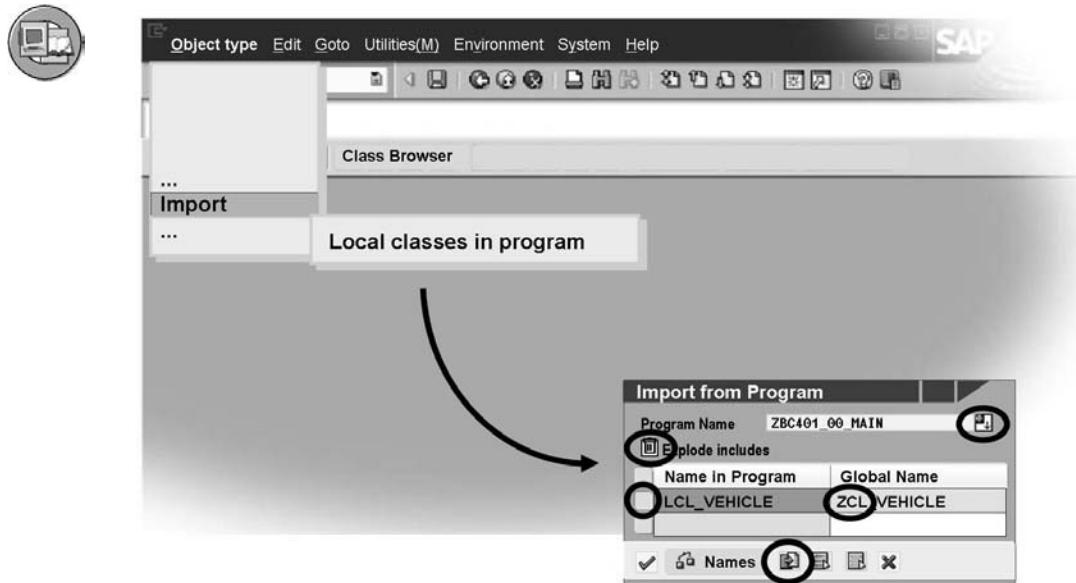


Figure 116: Importing a Local Program Class

Other Functions of the Class Builder

You set out the inheritance relationships between global classes on the *Properties* tab.

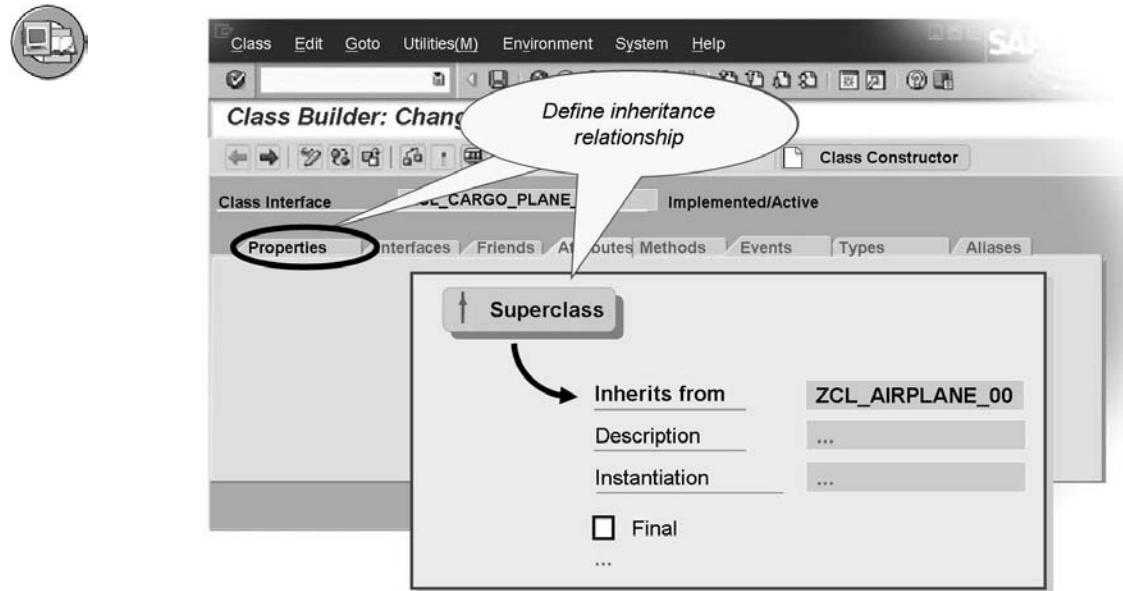


Figure 117: Defining an Inheritance Relationship

You can specify a superclass after you choose *Superclass*. In this example, the subclass ZCL_CARGO_PLANE_00 will inherit from superclass ZCL_AIRPLANE_00.

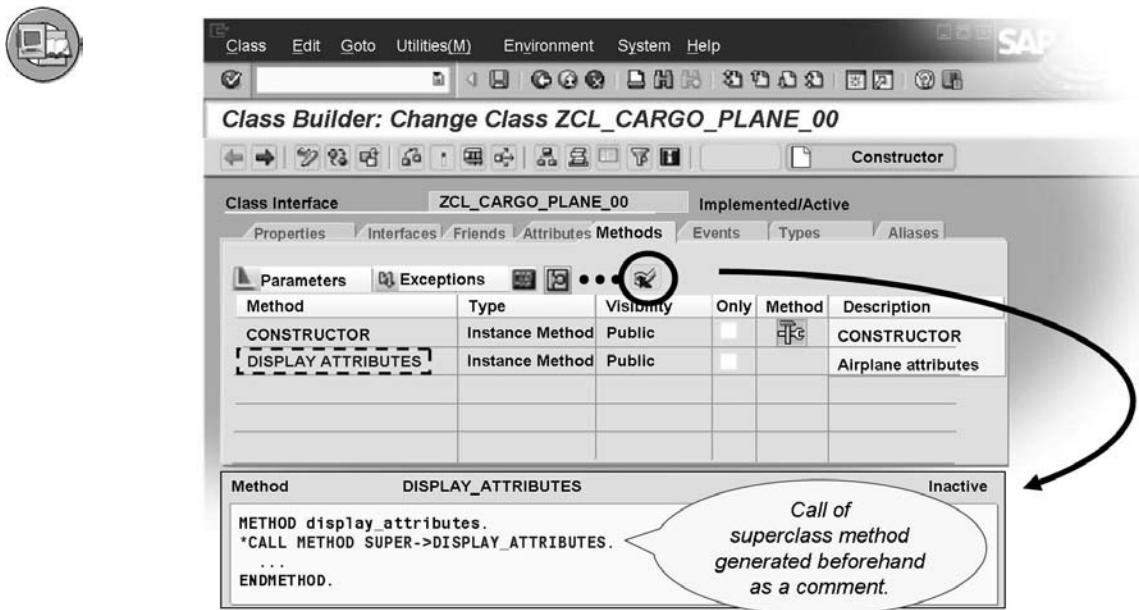


Figure 118: Redefining an Inherited Method

To redefine an inherited method, select the relevant method in the list and choose the *Redefine* button. Alternatively, you can use the context menu in the navigation area.

→ **Note:** Note that the appearance of some of the icons or menus depends on the release level. The icon for redefining methods is an example of this.

💡 **Hint: Creating the constructor in the subclass:** To define the constructor in the subclass, click on the **CONSTRUCTOR** button in the application toolbar. The system then proposes transferring the signature of the superclass constructor. This is helpful when you want to create the subclass constructor. You may have to add some parameters here. Likewise you will already find the call for the superclass constructor in the implementation for the subclass constructor.

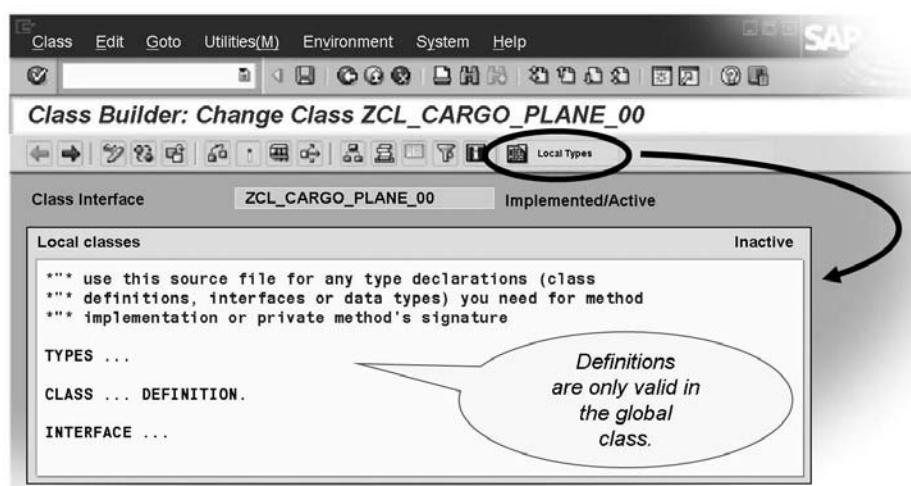


Figure 119: Defining a Local Type

You can define local types in global classes. This includes **local** classes in particular.

Technically, you are not defining a class within a class, but a class that is local in the Repository object of the global class.

All components of the global class have access to these local types, but they are encapsulated if you try to access them from outside.

The same applies for local interfaces in global classes.

To edit the implementation parts of these local classes, choose the *Impl.* button.(for *Local Class Implementations*).

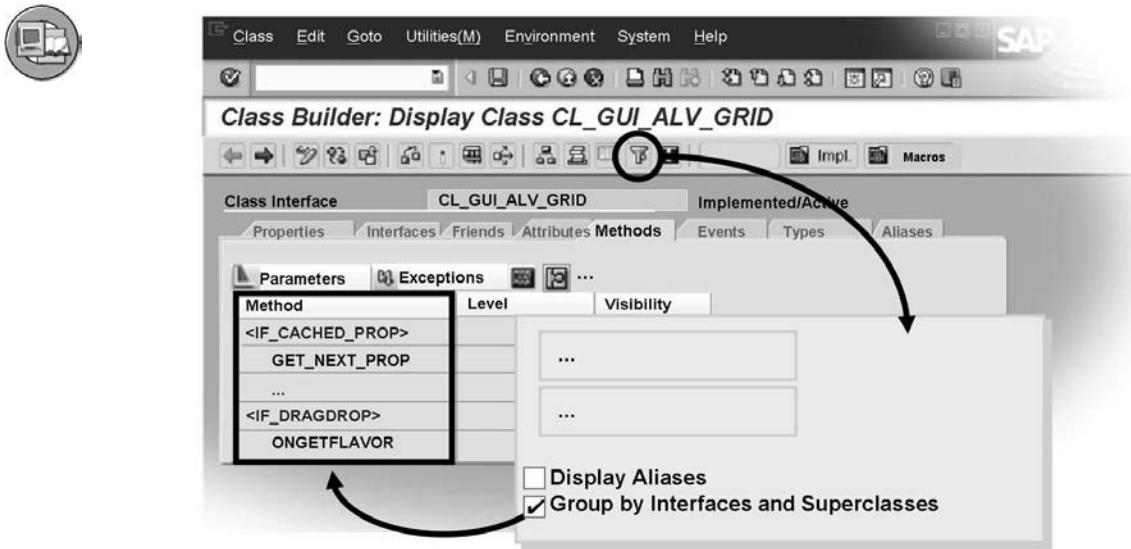


Figure 120: Structured Display of Inherited Components

To improve your understanding of inheritance and interface components, you can set the *Group by Classes and Interfaces* flag in the *User-Specific Settings* for the Class Builder. The system will then display the components of the global class in a group.

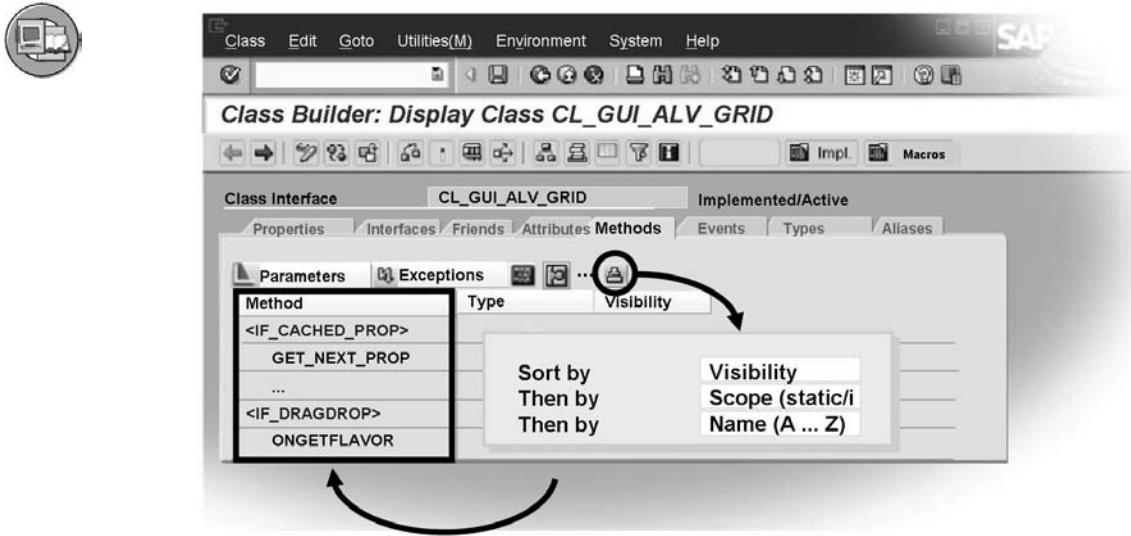


Figure 121: Sorting the Component Display of Global Classes

You can also sort all components by five criteria in three levels. To do this, display the appropriate dialog box by choosing the *Sort* button.

Addition of Global Classes Using the ABAP Editor

Like other repository objects, global classes and interfaces are added in the navigation area of the OBJECT-NAVIGATOR.

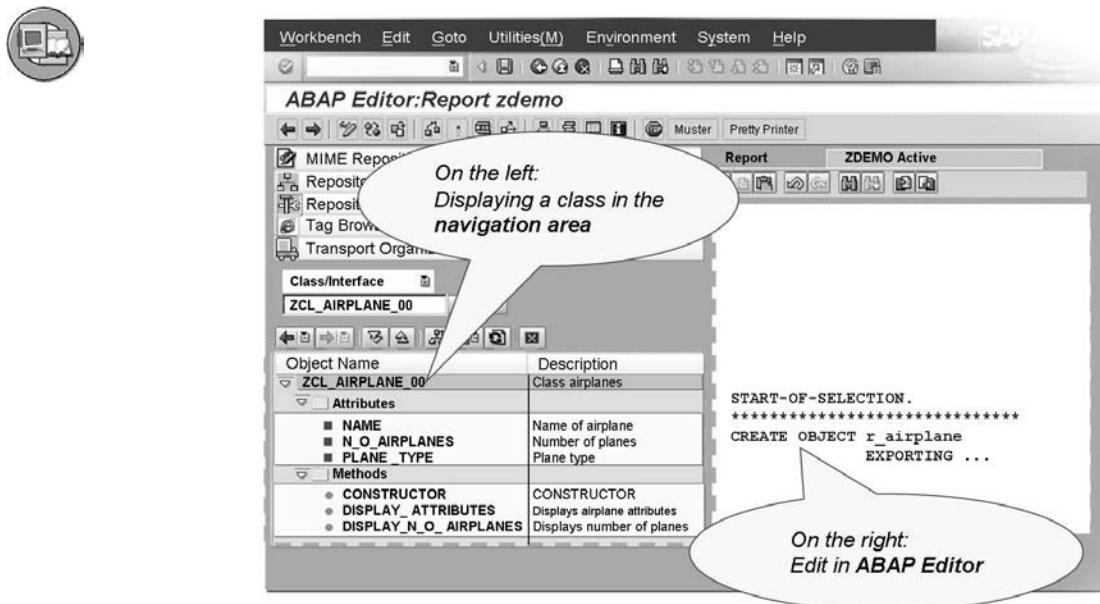


Figure 122: Separating the Navigation and Editing Areas of the Object Navigator.

This way, the previously discussed advantages also apply to working with global classes and interfaces.

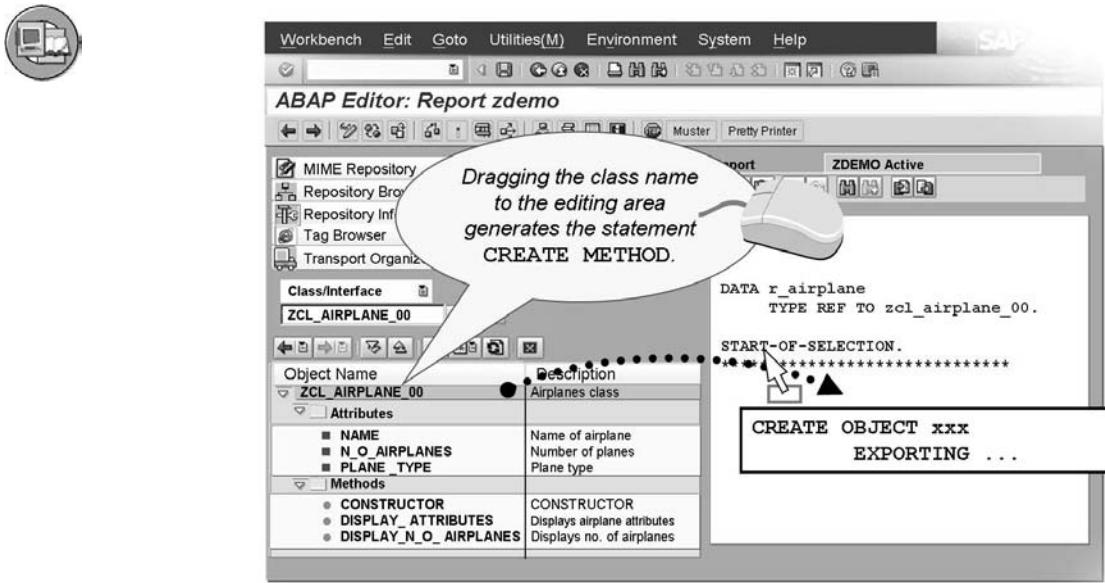


Figure 123: Object Instantiation Using Drag and Drop

In the navigation area, select a **class** name and drag it to the editing area by holding down the left mouse button. This creates a **CREATE OBJECT** statement. You must then add the reference variable and the actual parameters, if applicable, to the statement.

Alternatively, you can also press the *Pattern* button. The **CREATE-OBJECT** statement is under *ABAP Objects Pattern*. You can generate the statement using the input help.

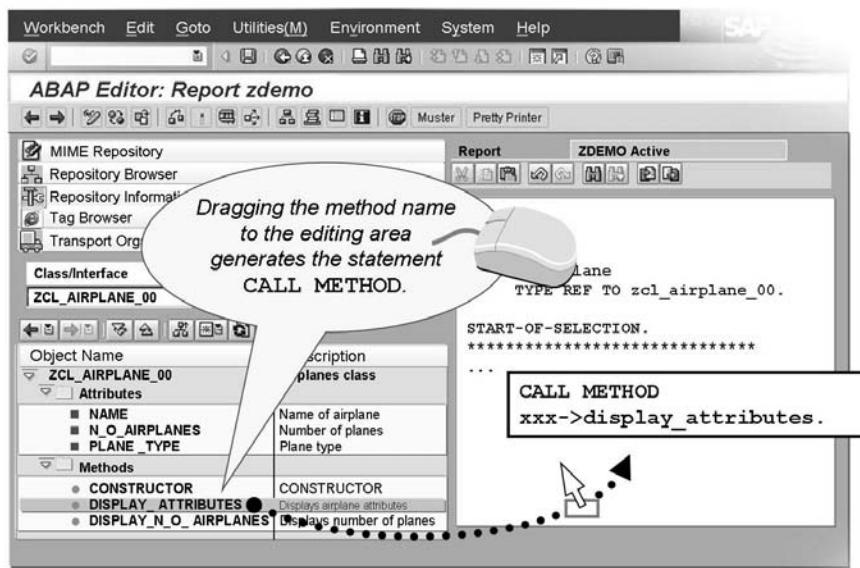


Figure 124: Method Calls Using Drag and Drop

In the navigation area, select a **method name** and drag it to the editing area by holding down the left mouse button. This creates a CREATE METHOD statement. You must then add the reference variable and the actual parameters, if applicable, to the statement.

Alternatively, you can also press the *Pattern* button. The CALL-METHOD statement is under *ABAP Objects Pattern*. You can generate the statement using the input help.

→ **Note:** As of SAP NW AS 7.0, you can convert to functional modern writing style when generating method calls. Follow the path *Utilities -> Settings -> Pattern* and select the checkbox “Functional Writing Style for Call Method”.

The Refactoring Assistant

In an ideal scenario, all classes, interfaces, and the associations between them would be modeled completely using UML diagrams before developers begin implementing them. However, in some cases, the model needs to be adapted during the implementation phase. The **Refactoring Assistant** offers a range of user-friendly options for you to change Repository objects that you created previously. For example, you can use the **Refactoring Assistant** to move the components of a class within the inheritance hierarchy. For a complete list of the features in this tool, refer to the *SAP Library*.

Working with the tool is simple, since it is based on drag and drop dialogs.

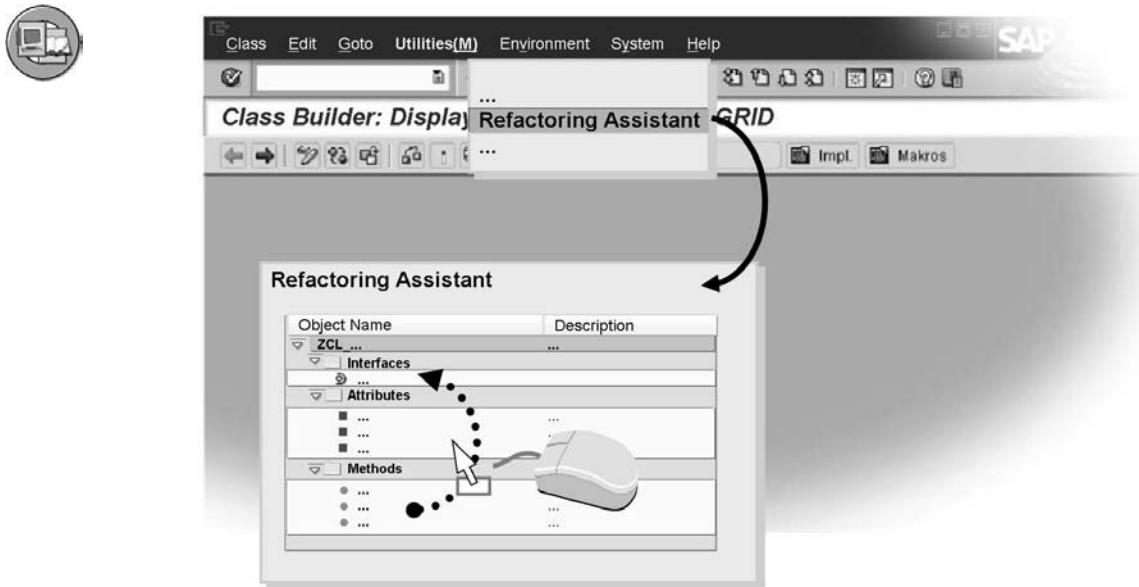


Figure 125: Working with the Refactoring Assistant



Hint: The *Tool Info* button opens the *SAP Library* article about the Refactoring Assistant. Here you will also find the descriptions of the other options provided by the tool.

Generally, you do **not** adjust the **implementation** of methods, since you cannot know how much you will need to alter object references after making these changes.



Moving the Method Definition of a Global Class to an Implemented Interface

1. Change to the editing mode of a global class that implements a global interface.
2. Select *Tools* → *Refactoring Assistant*.
3. In the tree structure that appears, open the folder for the method you want to move and for the target – in this case, the implemented interface.
4. Move the method to the interface.
5. Save your work.
6. You might also need to adapt statements that call the method in the source code.
7. Activate both the class **and** the interface.

Exercise 14: Global Classes

Exercise Objectives

After completing this exercise, you will be able to:

- Describe the functions of the *Class Builder*
- Create global classes using the *Class Builder*

Business Example

Create a global class to represent hotels.

Task 1:

Create a global class for hotels.

1. Create the global class ZCL_##_HOTEL. (## is your two-digit group number.))
2. Define the following attributes in the class:

NAME	of the type STRING	as a private instance attribute
MAX_BEDS	of the type I	as a private instance attribute
N_O_HOTELS	of the type I	as a private static attribute

Maintain the short texts.

3. Define the following methods in the class:

CONSTRUCTOR	Instance constructor for setting the private attributes with the import parameters IM_NAME and IM_BEDS
DISPLAY_ATTRIBUTES	Instance method for displaying attributes in an ABAP list
DISPLAY_N_O_HOTELS	Static method to display the number of created hotel instances in an ABAP list

Maintain the short texts.

Continued on next page

Task 2:

Check your work.

1. Activate your class.
2. Test your class in the *Class Builder's* testing environment.

Solution 14: Global Classes

Task 1:

Create a global class for hotels.

1. Create the global class ZCL_##_HOTEL. (## is your two-digit group number.))
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Model solution: CL_HOTEL



Caution: If you copy from this model solution, you must remove the inheritance relationship to the global class CL_HOUSE and the global interface IF_PARTNERS.

2. Define the following attributes in the class:

NAME	of the type STRING	as a private instance attribute
MAX_BEDS	of the type I	as a private instance attribute
N_O_HOTELS	of the type I	as a private static attribute

Maintain the short texts.

- a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.
3. Define the following methods in the class:

CONSTRUCTOR	Instance constructor for setting the private attributes with the import parameters IM_NAME and IM_BEDS
DISPLAY_ATTRIBUTES	Instance method for displaying attributes in an ABAP list
DISPLAY_N_O_HOTELS	Static method to display the number of created hotel instances in an ABAP list

Continued on next page

Maintain the short texts.

- a) Follow the processes as outlined in the relevant section of this lesson.
- b) Speak to your instructor if you have any questions.

Task 2:

Check your work.

1. Activate your class.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
2. Test your class in the *Class Builder's* testing environment.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.

Exercise 15: Global Interfaces

Exercise Objectives

After completing this exercise, you will be able to:

- Describe the functions of the *Class Builder*
- Create interfaces using the *Class Builder*
- Reference global classes/interfaces in other Repository objects

Business Example

Add a hotel as a new business partner in your program for managing a travel agency's business partners. Replace the local interface that you have been using with a global one, so that it can also be implemented by the global hotel class.

Task 1:

Create a global interface for generalized access to business partner instances.

1. If applicable, change the interface name in your UML diagram to ZIF_##_PARTNERS. (## is your two-digit group number.))
2. Create the global interface ZIF_##_PARTNERS.

Define the instance method DISPLAY_PARTNER and the instance event PARTNER_CREATED in the interface.



Hint: Import the local interface from your program ZBC401_##_MAIN or from the previous lesson's model solution SAPBC401_EVES_MAIN_B.

Task 2:

Have your hotel class implement the interface.

1. If necessary, add the class ZCL_##_HOTEL to your UML diagram. It should implement the interface method DISPLAY_PARTNER and trigger the instance event PARTNER_CREATED. Draw the relationships in your diagram.
2. Include the interface in your hotel class.
3. Implement the interface method in such a way that the hotel's DISPLAY_ATTRIBUTES instance method is called.

Continued on next page

4. a) If you have not performed the optional exercise on events, include the reference to your hotel object in the partner list for the travel agency with the ADD_PARTNER method. (You had already done this with the airline and the car rental company). b) If you have carried out the optional exercise on events, trigger the interface event in the constructor for the hotel class. Doing this means you do not have to explicitly call the method ADD_PARTNER.

Task 3:

Add an instance of your global hotel class to your main program.

1. Complete your program ZBC401_##_MAIN program or copy the program SAPBC401_EVES_MAIN_B (where ## is your two-digit group number).)
2. Remove the definition of the local interface and adapt all places where it was used to suit the new global interface.
3. Define a reference variable, specify your global hotel class as the type, and create an instance.
4. Run your program. If you have done everything correctly, the hotel attributes should now also be displayed in the list. Otherwise, debugging is the best way of analyzing any errors that may have occurred here.

Solution 15: Global Interfaces

Task 1:

Create a global interface for generalized access to business partner instances.

1. If applicable, change the interface name in your UML diagram to ZIF_##_PARTNERS. (## is your two-digit group number.)
 - a) Speak to your instructor if you have any questions.
2. Create the global interface ZIF_##_PARTNERS.

Define the instance method DISPLAY_PARTNER and the instance event PARTNER_CREATED in the interface.



Hint: Import the local interface from your program ZBC401_##_MAIN or from the previous lesson's model solution SAPBC401_EVES_MAIN_B.

- a) Follow the processes as outlined in the relevant section of this lesson.
- b) Speak to your instructor if you have any questions.
- c) Suggested name: IF_PARTNERS

Task 2:

Have your hotel class implement the interface.

1. If necessary, add the class ZCL_##_HOTEL to your UML diagram. It should implement the interface method DISPLAY_PARTNER and trigger the instance event PARTNER_CREATED. Draw the relationships in your diagram.
 - a) Speak to your instructor if you have any questions.
2. Include the interface in your hotel class.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.
3. Implement the interface method in such a way that the hotel's DISPLAY_ATTRIBUTES instance method is called.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.

Continued on next page

4.
 - a) If you have not performed the optional exercise on events, include the reference to your hotel object in the partner list for the travel agency with the ADD_PARTNER method. (You had already done this with the airline and the car rental company). b) If you have carried out the optional exercise on events, trigger the interface event in the constructor for the hotel class. Doing this means you do not have to explicitly call the method ADD_PARTNER.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.

Task 3:

Add an instance of your global hotel class to your main program.

1. Complete your program ZBC401_##_MAIN program or copy the program SAPBC401_EVES_MAIN_B (where ## is your two-digit group number).
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
 - b) Model solution: SAPBC401_CLSS_MAIN_A
2. Remove the definition of the local interface and adapt all places where it was used to suit the new global interface.
 - a) See the source code excerpt from the model solution.
3. Define a reference variable, specify your global hotel class as the type, and create an instance.
 - a) See the source code excerpt from the model solution.
4. Run your program. If you have done everything correctly, the hotel attributes should now also be displayed in the list. Otherwise, debugging is the best way of analyzing any errors that may have occurred here.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.

Result

Source code:

SAPBC401_CLSS_MAIN_A

```
REPORT  sapbc401_clss_main_a.
```

Continued on next page

```

TYPES: ty_fuel TYPE p DECIMALS 2,
       ty_cargo TYPE p DECIMALS 2.

TYPE-POOLS icon.

INCLUDE sapbc401_vehd_j.
INCLUDE sapbc401_clss_a.

DATA: r_vehicle TYPE REF TO lcl_vehicle,
      r_truck TYPE REF TO lcl_truck,
      r_bus    TYPE REF TO lcl_bus,
      r_passenger TYPE REF TO lcl_passenger_plane,
      r_cargo TYPE REF TO lcl_cargo_plane,
      r_carrier TYPE REF TO lcl_carrier,
      r_rental TYPE REF TO lcl_rental,
      r_agency TYPE REF TO lcl_travel_agency,
      r_hotel TYPE REF TO cl_hotel.

START-OF-SELECTION.
*#####
***** create travel_agency *****
CREATE OBJECT r_agency EXPORTING im_name = 'Fly&Smile Travel'.

***** create rental *****
CREATE OBJECT r_rental EXPORTING im_name = 'HAPPY CAR RENTAL'.
* fires event PARTNER_CREATED in rental to add partner to partnerlist

***** create truck *****
CREATE OBJECT r_truck EXPORTING im_make = 'MAN'
           im_cargo = 45.
***** create truck *****
CREATE OBJECT r_bus EXPORTING im_make = 'Mercedes'
           im_passengers = 80.

***** create truck *****
CREATE OBJECT r_truck EXPORTING im_make = 'VOLVO'
           im_cargo = 48.

***** Create CARRIER *****

```

Continued on next page

```

CREATE OBJECT r_carrier EXPORTING im_name = 'Smile&Fly Travel'.
* fires event PARTNER_CREATED in carrier to add partner to partnerlist

***** Passenger Plane *****
CREATE OBJECT r_passenger EXPORTING
    im_name = 'LH BERLIN'
    im_planetype = '747-400'
    im_seats = 345.

***** cargo Plane *****
CREATE OBJECT r_cargo EXPORTING
    im_name = 'US Hercules'
    im_planetype = '747-500'
    im_cargo = 533.

***** create hotel *****
CREATE OBJECT r_hotel EXPORTING im_name = 'Holiday Inn'
    im_beds = 345.
* fires event PARTNER_CREATED in hotel to add partner to partnerlist

***** show attributes of all partners of travel_agency *****
r_agency->display_agency_partners( ).
```

SAPBC401_VEHJ

```

*&-----
*&   Include          SAPBC401_VEHJ
*&-----
*-----
* work with the global interface if_hotel
*-----
* -----
*       CLASS lcl_vehicle DEFINITION
*-----
...
*-----
*       CLASS lcl_truck DEFINITION
*-----
*
*-----
*       CLASS lcl_bus DEFINITION
```

Continued on next page

```

*-----*
*
*-----*
...
*-----*
*      CLASS lcl_rental DEFINITION
*-----*
*
*-----*
CLASS lcl_rental DEFINITION.

PUBLIC SECTION.

"-----
METHODS:      constructor IMPORTING im_name TYPE string.
METHODS       add_vehicle FOR EVENT vehicle_created OF lcl_vehicle
              IMPORTING sender.
METHODS       display_attributes.
INTERFACES:   if_partners.

PRIVATE SECTION.

"-----
DATA: name TYPE string,
      vehicle_list TYPE TABLE OF REF TO lcl_vehicle.
ENDCLASS.          "lcl_rental DEFINITION

*-----*
*      CLASS lcl_rental IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_rental IMPLEMENTATION.

METHOD if_partners~display_partner.
    display_attributes( ).
ENDMETHOD.          "lif_partners~display_partner

METHOD constructor.
    name = im_name.
    SET HANDLER add_vehicle FOR ALL INSTANCES.
    RAISE EVENT if_partners~partner_created.
ENDMETHOD.          "constructor

METHOD add_vehicle.
    APPEND sender TO vehicle_list.

```

Continued on next page

```

ENDMETHOD.           "add_vehicle

METHOD  display_attributes.
DATA: r_vehicle TYPE REF TO lcl_vehicle.
skip 2.
WRITE: / icon_transport_proposal AS ICON, name.
WRITE: ' Here comes the vehicle list: '. ULINE. ULINE.
LOOP AT vehicle_list INTO r_vehicle.
r_vehicle->display_attributes( ).
ENDLOOP.

ENDMETHOD.           "display_attributes

ENDCLASS.           "lcl_rental IMPLEMENTATION

*-----
*      CLASS lcl_travel_agency DEFINITION
*-----
*
*-----*
CLASS lcl_travel_agency DEFINITION.

PUBLIC SECTION.
"-----
METHODS:   constructor IMPORTING im_name TYPE string.
METHODS    add_partner FOR EVENT partner_created OF if_partners
          IMPORTING sender.
METHODS    display_agency_partners.

PRIVATE SECTION.
"-----
DATA: name TYPE string,
      partner_list TYPE TABLE OF REF TO if_partners.
ENDCLASS.           "lcl_travel_agency DEFINITION

*-----
*      CLASS lcl_travel_agency IMPLEMENTATION
*-----
*
*-----*
CLASS lcl_travel_agency IMPLEMENTATION.

METHOD display_agency_partners.
DATA: r_partner TYPE REF TO if_partners.

```

Continued on next page

```

        WRITE: icon_dependents AS ICON, name.
        WRITE: ' Here are the partners of the travel agency: '.ULINE.ULINE.
        LOOP AT partner_list INTO r_partner.
        r_partner->display_partner( ).
        ENDLOOP.
        ENDMETHOD.           "display_agency_partners

METHOD constructor.
name = im_name.
SET HANDLER add_partner FOR ALL INSTANCES.
ENDMETHOD.           "constructor

METHOD add_partner.
APPEND sender TO partner_list.
ENDMETHOD.           "add_partner

ENDCLASS.           "lcl_travel_agency IMPLEMENTATION

```

SAPBC401_CLSS_A

```

*-----*
*   INCLUDE SAPBC401_CLSS_A
*-----*
* work with interface if_partners
* implement and raise events in lcl_carrier
*-----*
*      CLASS lcl_airplane DEFINITION
*-----*
*-----*
...
*-----*
*      CLASS lcl_cargo_plane DEFINITION
*-----*
...
*-----*
*      CLASS lcl_passenger_plane DEFINITION
*-----*
*
*-----*
...
*-----*
*      CLASS lcl_carrier DEFINITION
*-----*

```

Continued on next page

```

*
*-----*
CLASS lcl_carrier DEFINITION.

PUBLIC SECTION.
"-----
INTERFACES if_partners.
METHODS: constructor IMPORTING im_name TYPE string,
          get_name RETURNING value(ex_name) TYPE string,
          add_airplane FOR EVENT airplane_created OF lcl_airplane
                      IMPORTING sender,
          display_airplanes,
          display_attributes.

PRIVATE SECTION.
"-----
DATA: name           TYPE string,
      airplane_list TYPE TABLE OF REF TO lcl_airplane.
ENDCLASS.           "lcl_carrier DEFINITION

*-----*
*      CLASS lcl_carrier IMPLEMENTATION
*-----*
CLASS lcl_carrier IMPLEMENTATION.

METHOD if_partners-display_partner.
  display_attributes( ).
ENDMETHOD.           "lif_partners~display_partner

METHOD add_airplane.
  APPEND sender TO airplane_list.
ENDMETHOD.           "add_airplane

METHOD display_attributes.
  SKIP 2.
  WRITE: icon_flight AS ICON, name . ULINE. ULINE.
  display_airplanes( ).
ENDMETHOD.           "display_attributes

METHOD display_airplanes.
  DATA: r_plane TYPE REF TO lcl_airplane.
  LOOP AT airplane_list INTO r_plane.
    r_plane->display_attributes( ).
```

Continued on next page

```
ENDLOOP.  
ENDMETHOD.           "display_airplanes  
  
METHOD constructor.  
  name = im_name.  
  SET HANDLER add_airplane FOR ALL INSTANCES.  
  RAISE EVENT if_partners~partner_created.  
ENDMETHOD.           "constructor  
  
METHOD get_name.  
  ex_name = name.  
ENDMETHOD.           "get_name  
  
ENDCLASS.            "lcl_carrier IMPLEMENTATION
```


Exercise 16: (Optional) Refactoring Assistant

Exercise Objectives

After completing this exercise, you will be able to:

- Describe the functions of the *Class Builder*
- Create global classes using the *Class Builder*

Business Example

Practice working with the Refactoring Assistant.

Task 1:

Define a global superclass for houses and have your hotel class inherit from it.

1. If necessary, add the class ZCL_##_HOUSE to your UML diagram. (## is your two-digit group number.))

It will define the attribute NAME and the method DISPLAY_ATTRIBUTES. The class ZCL_##_HOTEL will inherit from it. Draw the relationships in your diagram.

2. Create the global class ZCL_##_HOTEL and leave it **empty**.
3. Define an inheritance relationship to make ZCL_##_HOUSE the superclass and ZCL_##_HOTEL the subclass.

Task 2:

Move the general components of the ZCL_##_HOTEL class to the superclass.

1. Use the *Refactoring Assistant* to move the NAME attribute, the instance constructor, and the DISPLAY_ATTRIBUTES method into the class ZCL_##_HOUSE.
2. Adapt the signature and the implementation of the instance constructor in the superclass.
3. Adapt the implementation of the DISPLAY_ATTRIBUTES method in the superclass.
4. Define the instance constructor for the subclass.

Continued on next page

5. Redefine the DISPLAY_ATTRIBUTES method in the subclass.
6. Observe the execution of the program in the *ABAP Debugger*.

If you have done everything correctly, the display of the list should be the same as in the previous exercise.

Solution 16: (Optional) Refactoring Assistant

Task 1:

Define a global superclass for houses and have your hotel class inherit from it.

1. If necessary, add the class ZCL_##_HOUSE to your UML diagram. (## is your two-digit group number.))

It will define the attribute NAME and the method DISPLAY_ATTRIBUTES. The class ZCL_##_HOTEL will inherit from it. Draw the relationships in your diagram.

- a) Speak to your instructor if you have any questions.
2. Create the global class ZCL_##_HOTEL and leave it **empty**.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.
 - c) Model solution: CL_HOUSE
3. Define an inheritance relationship to make ZCL_##_HOUSE the superclass and ZCL_##_HOTEL the subclass.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.

Task 2:

Move the general components of the ZCL_##_HOTEL class to the superclass.

1. Use the *Refactoring Assistant* to move the NAME attribute, the instance constructor, and the DISPLAY_ATTRIBUTES method into the class ZCL_##_HOUSE.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.
2. Adapt the signature and the implementation of the instance constructor in the superclass.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.

Continued on next page

3. Adapt the implementation of the DISPLAY_ATTRIBUTES method in the superclass.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.
4. Define the instance constructor for the subclass.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.
5. Redefine the DISPLAY_ATTRIBUTES method in the subclass.
 - a) Follow the processes as outlined in the relevant section of this lesson.
 - b) Speak to your instructor if you have any questions.
6. Observe the execution of the program in the *ABAP Debugger*.

If you have done everything correctly, the display of the list should be the same as in the previous exercise.

 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.



Lesson Summary

You should now be able to:

- Describe the functions of the Class Builder
- Create global classes using the Class Builder
- Create interfaces using the Class Builder
- Reference global classes and interfaces in other Repository objects

Related Information

Information about this topic is available in the SAP Library.

Lesson: Special Object-Oriented Programming Techniques

Lesson Overview

In this lesson, you will complete your knowledge of object-oriented programming. The concepts introduced here apply to other object-oriented programming languages in the same or in a similar way. They can be used freely in ABAP Objects, both with local and global classes.



Lesson Objectives

After completing this lesson, you will be able to:

- Define abstract classes
- Define abstract methods
- Define final classes
- Define final methods
- Limit the visibility of the constructor
- Define friendship relationships between classes
- Explain the “singleton pattern”

Business Example

You want to add special object-oriented programming techniques to your ABAP Objects implementations.

Abstract Classes and Abstract Methods

You can **prevent the instantiation** of a class by using the ABSTRACT addition with the CLASS statement. Superclasses are a typical use for abstract classes, as they are not intended to be instantiated themselves, but their subclasses are.

In such an abstract class, you can define abstract methods (amongst other things). This means that you can leave their implementation open. If the subclass of that class is not abstract, the abstract methods **must** be redefined there. This means that it must be implemented for the first time.

→ **Note:** The relevant indicator is in the **Class Builder** on the *Attributes* tab for that class or method.

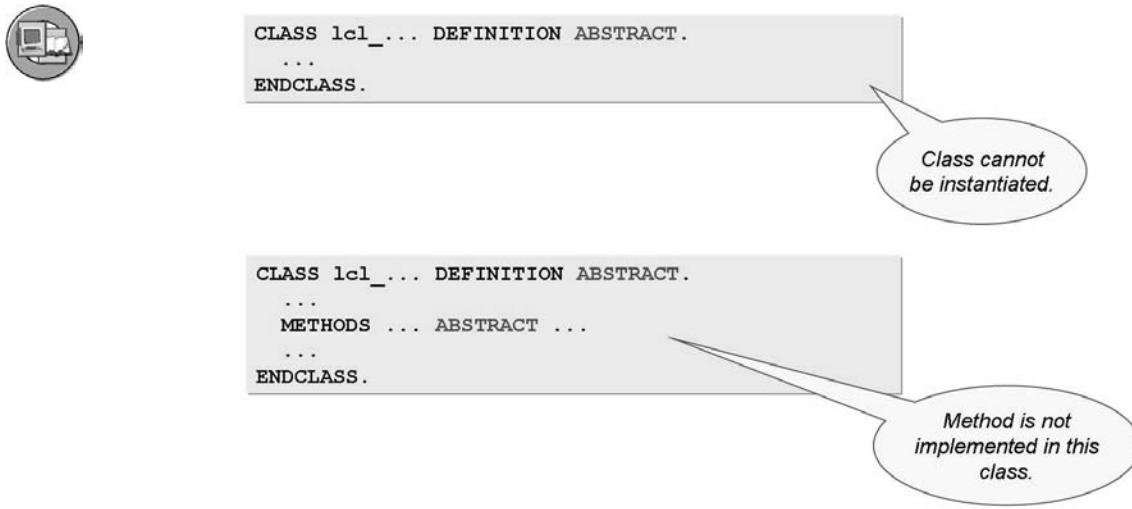


Figure 126: Abstract Classes and Abstract Methods

References to such abstract classes can therefore be used for polymorphic access to subclass instances.

Static methods cannot be abstract because they cannot be redefined.

Final Classes and Methods

You can prevent a class from being inherited by using the FINAL addition with the CLASS statement.

You can prevent a method from being redefined by using the FINAL addition with the METHODS statement.

→ **Note:** The relevant indicator is in the **Class Builder** on the *Attributes* tab for that class or method.

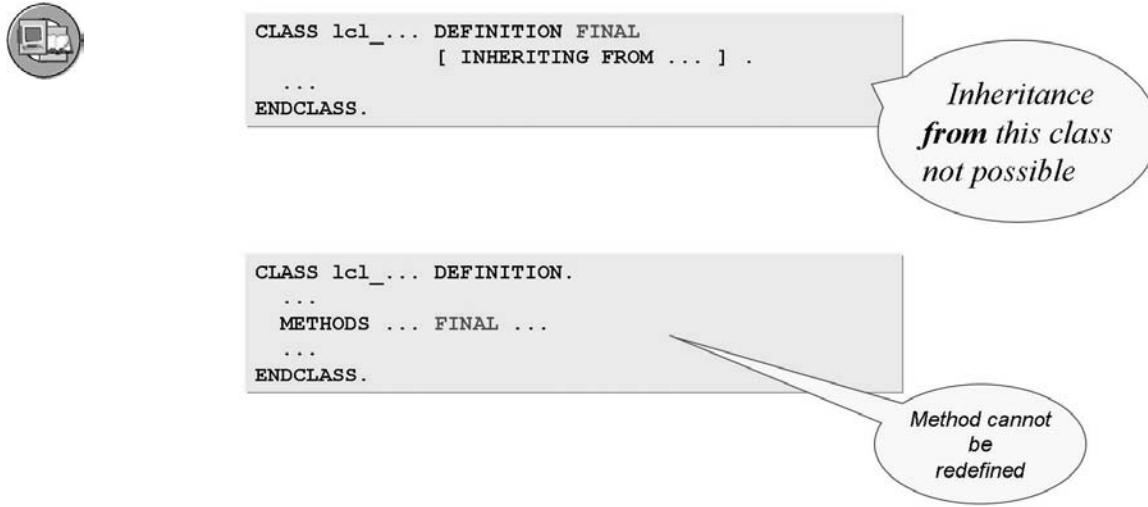


Figure 127: Final Classes and Methods

Thus, all methods of a final class are implicitly final. Therefore, you may not repeat the FINAL addition in the methods themselves.

Classes that are abstract **and** final should only contain static components.

Visibility of the Instance Constructor

You can **limit** the instantiability of a class by using the CREATE addition with the CLASS statement.

→ **Note:** The relevant indicator is in the **Class Builder** on the *Attributes* tab for the relevant class.

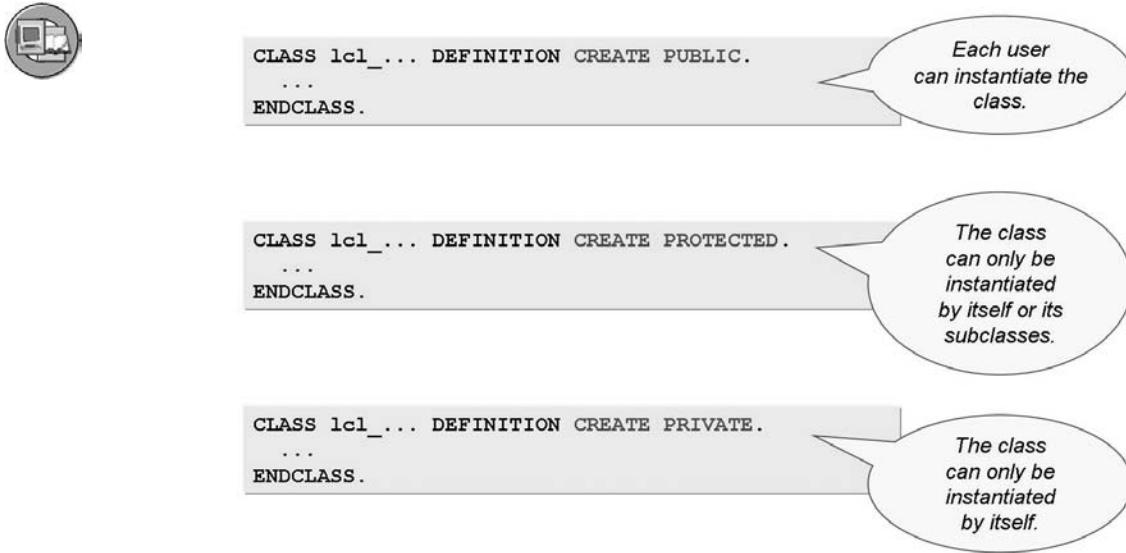


Figure 128: Implicit Setting of the Instance Constructor's Visibility

The variant CLASS ... DEFINITION CREATE PUBLIC ... is the standard case, that is, the normal definition of a class. The other two variants limit the context in which the CREATE OBJECT statement can appear, as specified above. Therefore, you can determine the visibility section for the instance constructor.

→ **Note:** Regardless of that, the instance constructor must always be defined **syntactically** in the public section.

Classes Without Multiple Instantiation

There are many cases in which you need to prevent a class from being instantiated more than once for each program context. You can do this as follows using the **singleton concept**:



```
CLASS lcl_singleton DEFINITION FINAL CREATE PRIVATE.
  PUBLIC SECTION.
    CLASS-METHODS class_constructor.
    CLASS-METHODS get_singleton
      RETURNING value(re_ref_single) TYPE REF TO lcl_singleton.
  PRIVATE SECTION.
    CLASS-DATA ref_single TYPE REF TO lcl_singleton.
ENDCLASS.
```

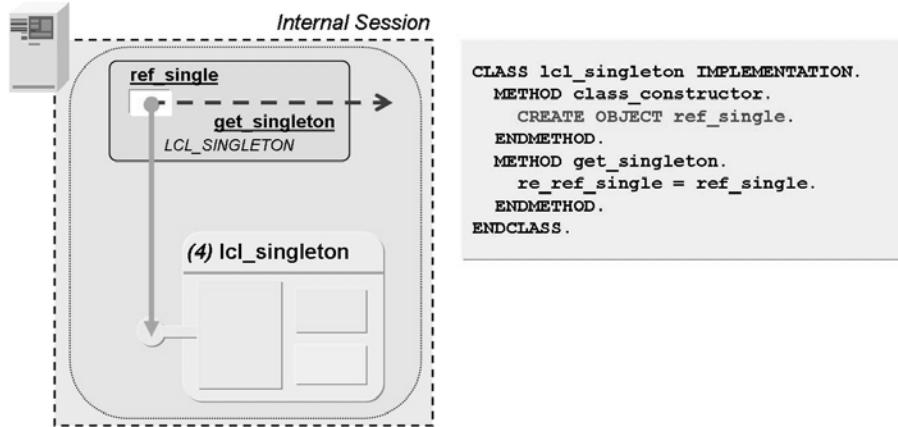


Figure 129: Singleton Classes

The class is defined with the additions FINAL and CREATE PRIVATE and is instantiated using its static constructor.

A public static component could then make the reference to the class available to an external user.

Friendship Relationship

In some cases, classes have to work together so closely that one or both classes need access to the other's protected and private components. Similarly, they need to be able to create instances of these other classes regardless of the visibility of the constructors. To avoid making these options available to all users, you can use the **class friendship** concept. A class can grant friendship to other classes and interfaces (and hence all classes that implement the interface).

This is the purpose of the FRIENDS addition of the CLASS statement or the FRIENDS tab in the *Class Builder*. All classes and interfaces that grant friendship are listed there.

In principle, granting friendship is one-sided: A class granting friendship is not automatically a friend of its friends. If a class granting friendship wants to access the non-public components of a friend, this friend must also explicitly grant friendship to it.



- Efficient direct access to the data of a class that grants friendship
- Distribution of various services across shared data using several classes
- Package creation

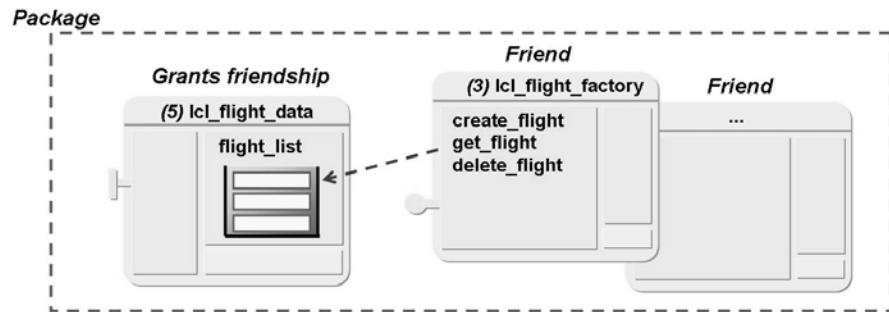


Figure 130: Areas of Use for Friendship Relationships

A typical application of the friendship relationship between classes is when methods that access the same data are distributed over several classes. This shared data is, however, to be protected from access by “foreign” classes. In such friendship relationships, you can make the class containing the data a singleton - that is, make sure it can only be instantiated once in each program instance.

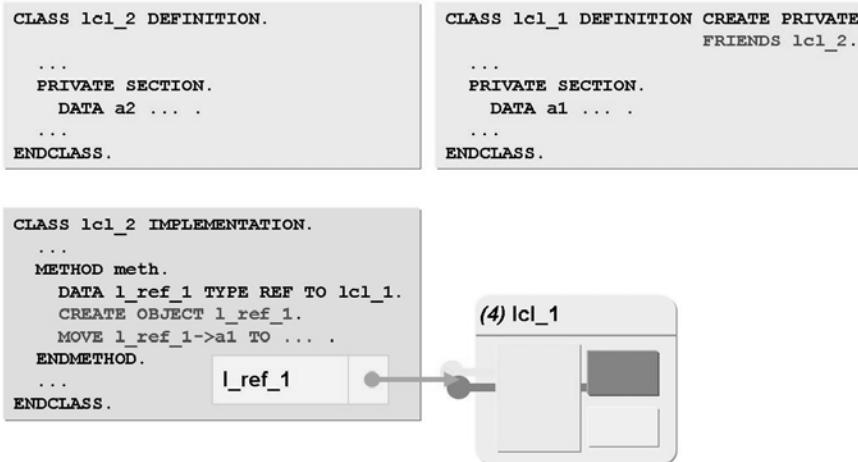


Figure 131: Definition of Friendship Relationships

The friend attribute is **inherited**: Classes that inherit from friends and interfaces containing a friend (as a component interface) also become friends. Therefore, extreme caution is advised when granting friendship. The higher up a friend is in the inheritance tree, the more subclasses can access all components of a class that grants friendship.

Conversely, the **granting** of friendship is **not** inherited. A friend of a superclass is therefore not automatically a friend of its subclasses.

Exercise 17: Singleton Classes (Optional)

Exercise Objectives

After completing this exercise, you will be able to:

- Describe the functions of the *Class Builder*
- Create global classes using the *Class Builder*
- Define singleton classes
- Instantiate singleton classes

Business Example

You need a class that can only be instantiated once for each program context.

Task 1:

Create a global singleton class.

1. Create the global class ZCL_##_SINGLETON. (## is your two-digit group number.) Specify the necessary attributes for the singleton class when you do this.
2. Define the attribute R_SINGLETON for a reference to the singleton instance, making sure that the attribute has a suitable type.
3. Define and implement the method GET_SINGLETON. It will provide the reference to the singleton instance.
4. Ensure that the singleton instance is **only** created with the **first call** of the GET_SINGLETON method.

Task 2:

Create a singleton instance.

1. Create the executable program ZBC401_##_MAIN_SPECIAL.
2. Define a reference variable for the singleton instance.
3. Create the singleton instance.

Solution 17: Singleton Classes (Optional)

Task 1:

Create a global singleton class.

1. Create the global class ZCL_##_SINGLETON. (## is your two-digit group number.) Specify the necessary attributes for the singleton class when you do this.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
 - b) Specify the attributes as they are outlined in the relevant section of this lesson.
2. Define the attribute R_SINGLETON for a reference to the singleton instance, making sure that the attribute has a suitable type.
 - a) Use the relevant sections of the lesson as a guide.
3. Define and implement the method GET_SINGLETON. It will provide the reference to the singleton instance.
 - a) Use the relevant sections of the lesson as a guide.
4. Ensure that the singleton instance is **only** created with the **first call** of the GET_SINGLETON method.
 - a) Use the relevant sections of the lesson as a guide.

Task 2:

Create a singleton instance.

1. Create the executable program ZBC401_##_MAIN_SPECIAL.
 - a) Carry out this step in the usual manner.
 - b) Model solution: SAPBC401_SPCS_MAIN_A
2. Define a reference variable for the singleton instance.
 - a) Carry out this step in the usual manner. See the model solution.
3. Create the singleton instance.
 - a) See the source code excerpt from the model solution.

Continued on next page

Result

Source code excerpt:

SAPBC401_SPCS_MAIN_A

```

REPORT sapbc401_spcs_main_a.

*-----*
*      CLASS lcl_singleton DEFINITION
*-----*
CLASS lcl_singleton DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
    -----
    CLASS-METHODS: class_constructor.
    CLASS-METHODS: get_singleton RETURNING value(re_single)
                  TYPE REF TO lcl_singleton.

  PRIVATE SECTION.
    -----
    CLASS-DATA: r_single TYPE REF TO lcl_singleton.
ENDCLASS.          "lcl_singleton DEFINITION
*-----*
*      CLASS lcl_singleton IMPLEMENTATION
*-----*
CLASS lcl_singleton IMPLEMENTATION.
  METHOD class_constructor.
    CREATE OBJECT r_single.
  ENDMETHOD.          "class_constructor
  METHOD get_singleton.
    re_single = r_single.
  ENDMETHOD.          "get_singleton
ENDCLASS.          "lcl_singleton IMPLEMENTATION

*** Main Program;
*** creating the singleton via class-constructor

DATA: r_single TYPE REF TO cl_singleton.

START-OF-SELECTION.
*#####
r_single = lcl_singleton=>get_singleton( ).
```

Continued on next page

```
*** repeat this statement and watch the execution in the Debugger !
*** you will see the implicit call of the class_constructor !
```

Exercise 18: Friendship Relationships (Optional)

Exercise Objectives

After completing this exercise, you will be able to:

- Describe the functions of the *Class Builder*
- Create global classes using the *Class Builder*
- Define friendship relationships
- Access private attributes of a friendship-granting class

Business Example

Create a global class for travel agencies, which can access the private attributes of a singleton class that is a friend. The class will buffer flight connection data.

Task 1:

Buffer the flight data in a singleton class.

1. Add the CONNECTION_LIST attribute for the flight connection data to your ZCL_##_SINGLETON singleton class, making sure that the right attributes have been set for the flight connection data. (## is your two-digit group number.))



Hint: You can use the global table type TY_CONNECTIONS.

2. Make sure that this internal table is filled with data from the transparent table SPFLI, but **only** when the singleton instance is accessed **for the first time**.

Task 2:

Create a global class for travel agencies and have the singleton class grant it friendship. Enable the new global class to pass flight connection data to a user.

1. Create the global class ZCL_##_AGENCY. (## is your two-digit group number.) The singleton class should grant it friendship.
2. Define the private instance attribute NAME of type STRING.
3. Define and implement an instance constructor that sets the private attribute NAME.

Continued on next page

4. Define the public instance method GET_CONNECTION with the following signature:

ID	Type	Associated Type
IM_CARRID	Importing	S_CARR_ID
IM_CONNID	Importing	S_CONN_ID
EX_CONNECTION	Exporting	SPFLI

5. Implement your method in such a way that the additional data for selected flight connections can be read and exported from the private internal table of the singleton instance using single-record access.

If the required data record does not exist, it should be sufficient in this case to output a message in the ABAP list.



Hint: If required, use the CLASS . . . DEFINITION LOAD statement to **syntactically** enable direct access to the static components of the class.

This addition is not required in later releases.

Task 3:

Create a travel agency instance and have it provide the data of a flight connection.

1. Add a reference variable for a travel agency instance to the executable program ZBC401_##_MAIN_SPECIAL that you created earlier. (## is your two-digit group number.))
2. Create a travel agency instance.
3. Call the GET_CONNECTION method for an existing flight connection from the flight data model (for example, **LH/0400**) and display the data in the ABAP list.

Solution 18: Friendship Relationships (Optional)

Task 1:

Buffer the flight data in a singleton class.

1. Add the CONNECTION_LIST attribute for the flight connection data to your ZCL_##_SINGLETON singleton class, making sure that the right attributes have been set for the flight connection data. (## is your two-digit group number.))



Hint: You can use the global table type TY_CONNECTIONS.

- a) Speak to your instructor if you have any questions.
- b) Model solution: CL_SINGLETON
2. Make sure that this internal table is filled with data from the transparent table SPFLI, but **only** when the singleton instance is accessed **for the first time**.
 - a) Speak to your instructor if you have any questions.

Task 2:

Create a global class for travel agencies and have the singleton class grant it friendship. Enable the new global class to pass flight connection data to a user.

1. Create the global class ZCL_##_AGENCY. (## is your two-digit group number.) The singleton class should grant it friendship.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
 - b) Model solution: CL_AGENCY
2. Define the private instance attribute NAME of type STRING.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
3. Define and implement an instance constructor that sets the private attribute NAME.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.

Continued on next page

4. Define the public instance method GET_CONNECTION with the following signature:

ID	Type	Associated Type
IM_CARRID	Importing	S_CARR_ID
IM_CONNID	Importing	S_CONN_ID
EX_CONNECTION	Exporting	SPFLI

- a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
5. Implement your method in such a way that the additional data for selected flight connections can be read and exported from the private internal table of the singleton instance using single-record access.

If the required data record does not exist, it should be sufficient in this case to output a message in the ABAP list.



Hint: If required, use the CLASS . . . DEFINITION LOAD statement to **syntactically** enable direct access to the static components of the class.

This addition is not required in later releases.

- a) Use the relevant sections of the lesson as a guide.

Task 3:

Create a travel agency instance and have it provide the data of a flight connection.

1. Add a reference variable for a travel agency instance to the executable program ZBC401_##_MAIN_SPECIAL that you created earlier. (## is your two-digit group number.))
 - a) See the source code excerpt from the model solution.
2. Create a travel agency instance.
 - a) See the source code excerpt from the model solution.
3. Call the GET_CONNECTION method for an existing flight connection from the flight data model (for example, LH/0400) and display the data in the ABAP list.
 - a) See the source code excerpt from the model solution.

Continued on next page

Result

Source code excerpt:

SAPBC401_SPCS_MAIN_B

```
REPORT  sapbc401_spcs_main_b.

TYPES: ty_connection_list TYPE HASHED TABLE OF spfli
      WITH UNIQUE KEY carrid connid.

*** just because lcl_agency is defined later !
CLASS lcl_agency DEFINITION DEFERRED.

*-----*
*      CLASS lcl_singleton DEFINITION
*-----*
CLASS lcl_singleton DEFINITION CREATE PRIVATE FRIENDS lcl_agency.
PUBLIC SECTION.
"-----
CLASS-METHODS: class_constructor.
CLASS-METHODS: get_singleton RETURNING value(re_single)
               TYPE REF TO lcl_singleton.

PRIVATE SECTION.
"-----
CLASS-DATA: r_single TYPE REF TO lcl_singleton.
CLASS-DATA: connection_list TYPE ty_connection_list.
ENDCLASS.          "lcl_singleton DEFINITION

*-----*
*      CLASS lcl_singleton IMPLEMENTATION
*-----*
CLASS lcl_singleton IMPLEMENTATION.

METHOD class_constructor.
  CREATE OBJECT r_single.
  SELECT * FROM spfli INTO TABLE connection_list.
ENDMETHOD.          "class_constructor

METHOD get_singleton.
  re_single = r_single.
ENDMETHOD.          "get_singleton

ENDCLASS.          "lcl_singleton IMPLEMENTATION
```

Continued on next page

```

*-----*
*      CLASS lcl_agency DEFINITION
*-----*
CLASS lcl_agency DEFINITION.
  PUBLIC SECTION.
    -----
    METHODS: constructor IMPORTING im_name TYPE string.
    METHODS: get_connection IMPORTING im_carrid TYPE s_carr_id
              im_connid TYPE s_conn_id
              EXPORTING ex_connection TYPE spfli.
  PRIVATE SECTION.
    -----
    DATA: name TYPE string.
ENDCLASS.                      "lcl_agency DEFINITION

*-----*
*      CLASS lcl_agency IMPLEMENTATION
*-----*
CLASS lcl_agency IMPLEMENTATION.
  METHOD constructor.
    name = im_name.
  ENDMETHOD.                  "constructor

  METHOD get_connection.
*** direct access to the private components of the singletonclass.
*** the singleton object is created before via class_constructor !
*** the singleton object itself is not needed in this example
*** because we just access a static attribute !

  READ TABLE lcl_singleton=>connection_list
    INTO ex_connection
    WITH TABLE KEY carrid = im_carrid
          connid = im_connid.

  ENDMETHOD.                  "get_singleton
ENDCLASS.                      "lcl_singleton IMPLEMENTATION

DATA: r_agency TYPE REF TO lcl_agency.
DATA: wa TYPE spfli.

START-OF-SELECTION.
*#####

```

Continued on next page

```
CREATE OBJECT r_agency EXPORTING im_name =           'I will access the private components o
r_agency->get_connection( EXPORTING im_carrid = 'LH'
                           im_connid = '0400'
                           IMPORTING ex_connection = wa ).

WRITE: / wa-carrid, wa-connid, wa-cityfrom, wa-cityto.
```



Lesson Summary

You should now be able to:

- Define abstract classes
- Define abstract methods
- Define final classes
- Define final methods
- Limit the visibility of the constructor
- Define friendship relationships between classes
- Explain the “singleton pattern”

Related Information

For more information about this subject, refer to the SAP Library and the ABAP keyword documentation for the individual statements.



Unit Summary

You should now be able to:

- Describe the functions of the Class Builder
- Create global classes using the Class Builder
- Create interfaces using the Class Builder
- Reference global classes and interfaces in other Repository objects
- Define abstract classes
- Define abstract methods
- Define final classes
- Define final methods
- Limit the visibility of the constructor
- Define friendship relationships between classes
- Explain the “singleton pattern”



Test Your Knowledge

1. Which of the following statements are correct?

Choose the correct answer(s).

- A You can create function modules using the *Class Builder*.
 - B A global class can contain a local class.
 - C A global interface can contain a local interface.
 - D A global class can contain a local interface.
 - E Nested definition of classes is when a local class is within a global class.
 - F Using the *Class Builder*, a local class can be converted into a global class.
 - G A local class can be copied using the *Class Builder*. The copy is then a global class.
 - H You can use the Refactoring Assistant to move the methods to a different class within an inheritance hierarchy.
 - I You can use the Refactoring Assistant to design model diagrams.
-
2. For a user to be able to execute an object-oriented program, you always need to supply a module pool program or a function group program. Otherwise, there is nowhere for the CREATE OBJECT statement to create the instance.
- Determine whether this statement is true or false.*
- True
 - False

3. Which of the following statements are correct?

Choose the correct answer(s).

- A A non-abstract class can contain abstract methods.
- B An abstract class contains no implementations.
- C An abstract method contains no implementations.
- D Final classes cannot be superclasses within a class hierarchy.
- E A final method must be redefined.
- F Final classes can contain non-final methods.
- G A friend of a class is also a friend of its subclasses.
- H The subclasses of a class's friend are also the class's friend.
- I The visibility of an instance constructor can be limited.
- J A private instance constructor (instantiation only by the class itself) can be defined in the private section.



Answers

1. Which of the following statements are correct?

Answer: B, D, G, H, I

Additional information is available in the SAP Library.

2. For a user to be able to execute an object-oriented program, you always need to supply a module pool program or a function group program. Otherwise, there is nowhere for the CREATE OBJECT statement to create the instance.

Answer: False

Additional information is available in the SAP Library.

3. Which of the following statements are correct?

Answer: C, D, H, I, J

Unit 4

Class-Based Exception Concept

Unit Overview

This unit deals with the new class-based exception concept.



Unit Objectives

After completing this unit, you will be able to:

- Handle class-based exceptions in ABAP programs
- Create exception classes
- Raise class-based exceptions in ABAP programs
- Propagate class-based exceptions in ABAP programs
- Map class-based exceptions in ABAP programs to one another

Unit Contents

Lesson: Class-Based Exceptions.....	292
Procedure: Defining Global Exception Classes	301
Exercise 19: Handle Class-Based Exceptions	311
Exercise 20: Create and Use Global Exception Classes.....	317

Lesson: Class-Based Exceptions

Lesson Overview

This lesson deals with the new class-based exception concept that was introduced with *SAP Web Application Server 6.10*. The main features and the activities you need to perform are presented in detail and then applied in practical examples.



Lesson Objectives

After completing this lesson, you will be able to:

- Handle class-based exceptions in ABAP programs
- Create exception classes
- Raise class-based exceptions in ABAP programs
- Propagate class-based exceptions in ABAP programs
- Map class-based exceptions in ABAP programs to one another

Business Example

You should use the new exceptions concept in your ABAP programs.

Exceptions: An Overview

We use the term **exception** to refer to a situation that arises while a program is being executed, where there is no point in continuing to run the program in the normal way. The *SAP Web Application Server 6.10* introduced a new ABAP exception concept in parallel with the existing concept. Exceptions and exception handling are now based on classes.

If, prior to *SAP Web Application Server 6.10*, exceptions were raised by the system (due to memory overflow or a failed parameter mapping for example), the system variable *sy-subrc* was set to a value other than zero. A developer could enclose the critical point with the syntax `CATCH ... ENDCATCH.` and read the value from *sy-subrc* after this block. If there was no `CATCH ... ENDCATCH.`, a runtime error occurred.

When unwanted states arise in function modules and methods, the developer could declare this kind of procedural exception and raise it with `RAISE <exception>`. Callers had to program a suitable query to the system variable *sy-subrc* in their source code after the procedure call, otherwise a runtime error occurred.

Both cases still exist in systems based on *SAP Web Application Server 6.10* and above. When system exceptions occur, a classic type exception is always generated **as well as** a class-based exception too. When developers respond to this, they can choose which way they want to handle an exception. With function modules and methods, the developer has to opt for one of the concepts for such a procedure. Accordingly, the user of a function module is obliged to choose the new **or** the old handling technique.

A new feature of *SAP Web Application Server 6.10* and above is the possibility of raising exceptions at any points in the program. This only works with the class-based technique.

Overview of Both Exception Concepts



Place where exception is raised	before 6.10	as of 6.10
System exception (for example, zero division)	sy-subrc	sy-subrc AND class-based
function modules, methods	sy-subrc	sy-subrc OR class-based
raised from any processing block by explicit command	not possible	class-based

Class-Based Exception Handling

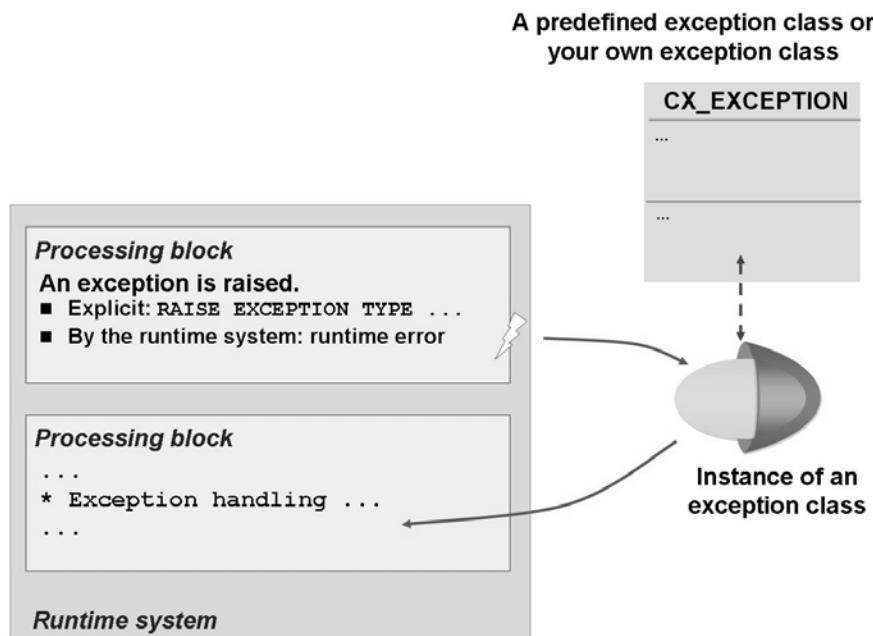


Figure 132: Overview of the Class-Based Exception Concept

Class-based exceptions are raised either by the `RAISE EXCEPTION` statement or by the runtime environment. Division by zero, for example, would be an example of the latter.

In an exception situation, an exception is represented by an exception object - that is, an instance of an exception class. The attributes of each exception object contain information about the error situation.

When handling an exception of this kind you can raise new exceptions and thus create an exception chain. Thus, in special cases, you can catch a runtime exception and then have it raise an exception of its own.

You can define exception classes yourself, although there is already a range of predefined exception classes in the system, particularly for exceptions in the runtime environment. You usually create exception classes globally in the *Class Builder*, but you can also define local exception classes within a program or global class.

The use of class-based exceptions is not limited to object-oriented contexts. Class-based exceptions can be raised and handled in all processing blocks. All previously catchable runtime errors in particular can now be handled as class-based exceptions.

If a class-based exception is raised, the system interrupts the normal program flow and tries to branch to a suitable handler. If it cannot find a handler, a runtime error occurs.

All exception classes are derived from the classes CX_NO_CHECK, CX_DYNAMIC_CHECK, or CX_STATIC_CHECK, and thus from the shared superclass CX_ROOT.

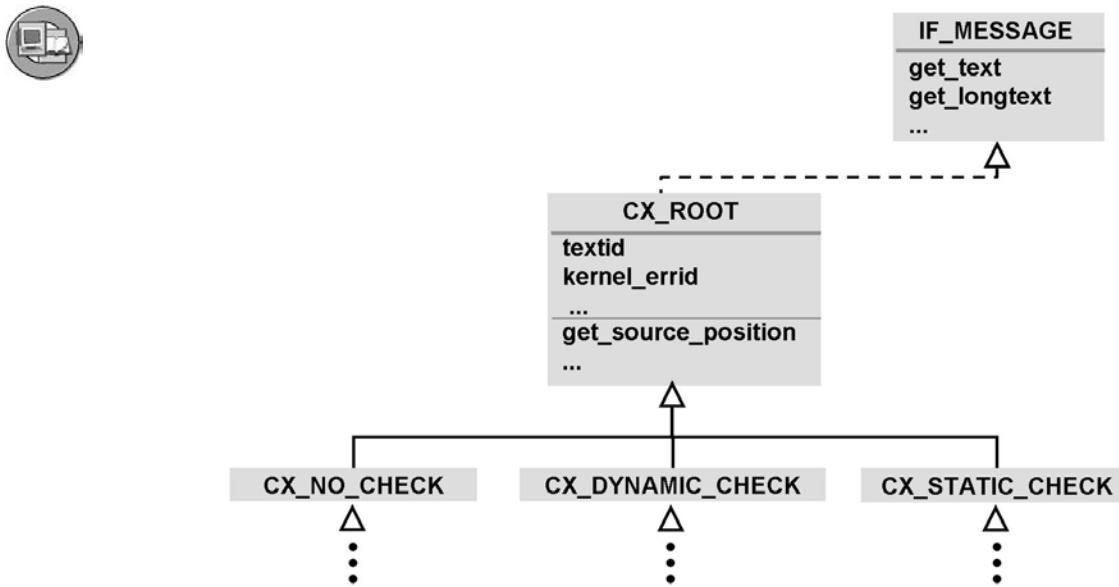


Figure 133: Exception Classes: The Inheritance Hierarchy

The assignment of exception classes to one of these three branches in the inheritance hierarchy defines how the associated exceptions are propagated. (This will be discussed in more detail later.)

In the SAP standard system, the names of all exception classes start with CX_. Customer-defined exception classes have a namespace + the prefix CX_ (for example, ZCX_ or /NAMESPACE/CX_).

The class CX_ROOT provides predefined methods, which all exception classes inherit. The GET_SOURCE_POSITION method returns the name of the framework program, the include name (if relevant), and line number in the source code where the exception occurred. The GET_TEXT method returns an exception text in the form of a string. The GET_LONGTEXT method is also available to all exception classes, however, a long text is not always generated for every exception.

Newly defined exception classes inherit from one of these superclasses, so that other components can be added. These are usually individual exception texts.

You can assign several texts to each class. The IDs assigned to these are created by the *Class Builder* as identically named static constants. You can then specify which text is to be used when an exception is raised by transferring one of these constants to the import parameter TEXTID of the instance constructor.

All exception classes inherit the KERNEL_ERRID attribute from CX_ROOT. This attribute contains the name of the relevant runtime error if the exception was raised by the runtime environment, such as BCD_ZERODIVIDE if the program catches a CX_SY_ZERODIVIDE exception (division by zero). If the exception is not handled, this runtime error occurs.

An exception can only be handled if the statement that **could** raise it is nested in a TRY-ENDTRY block. The exception is then handled using the CATCH statement in the TRY-ENDTRY block.

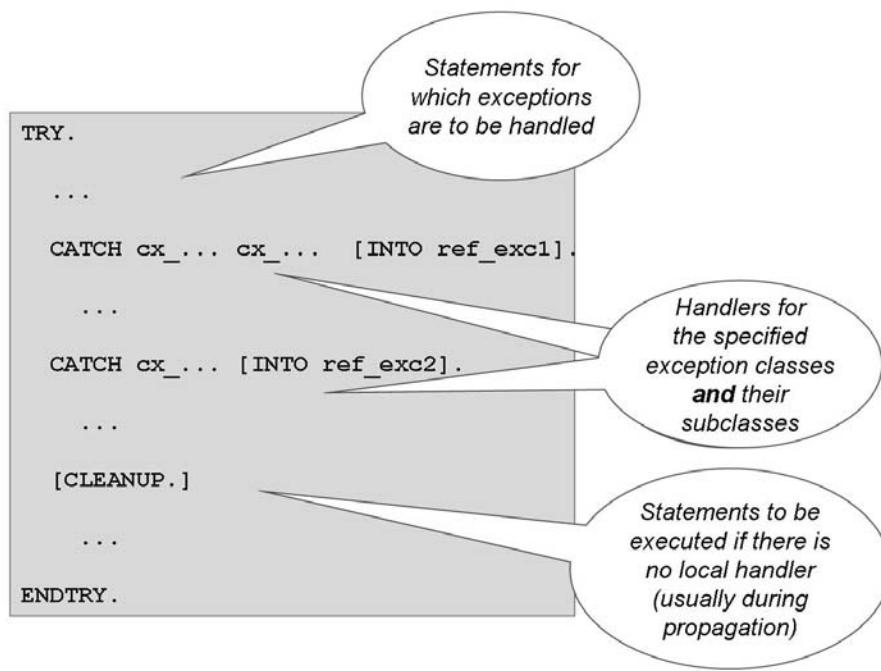


Figure 134: TRY-ENDTRY Sequence

The TRY block contains the sequence with the exceptions that have to be handled. If an exception occurs in the TRY block, the system first searches for a CATCH statement in the same TRY-ENDTRY block and then searches outwards for a CATCH statement blocks that handles the exception in any enclosing TRY-ENDTRY that may exist. If it finds one, it navigates to this handler. If it cannot find a handler, but the TRY-ENDTRY block is in a procedure, the system then tries to propagate the exception to the calling program. (This will be discussed in more detail later.)

A CATCH block contains the exception handler that is executed if a specified exception has occurred in the corresponding TRY block. You can specify any number of exception classes for the CATCH statement. In this way, you define an exception handler for all these exception classes and their subclasses.

Like all control structures in ABAP, TRY-ENDTRY blocks can be nested as deep as you wish. Thus the TRY block, CATCH blocks, and the CLEANUP block in particular can contain complete TRY-ENDTRY blocks themselves.

After an exception occurs, the system searches through the listed exception handlers in the order specified. It then executes the first exception handler whose CATCH statement contains the relevant exception class or one of its superclasses.

A CLEANUP block is executed if the TRY-ENDTRY block is exited because the system cannot find a handler for an exception but the exception is handled in a surrounding TRY-ENDTRY block or propagated to a calling program.

The following sections will focus on global exception classes that are handled locally. Remember, you can also use local exception classes handled locally or global exception classes handled in methods of other global classes.

Example: Handling a Predefined Exception

There are two special features to note in the following example: Firstly, we will simply instantiate a standard exception class, rather than define a new one. Secondly, the exception will be raised by the runtime system as the result of an error, rather than by a procedure.



```
PARAMETERS: pa_int1  TYPE i,
            pa_int2  TYPE i.
```

```
DATA: ge_result  TYPE i,
      ge_text    TYPE string,
      gx_exc     TYPE REF TO cx_root.
```

```
...
```

```
TRY.
```

```
  ge_result = pa_int1 * pa_int2.
  WRITE ge_result.
  CATCH cx_sy_arithmetic_overflow INTO gx_exc.
    ge_text = gx_exc->get_text( ).
```

```
  MESSAGE ge_text TYPE 'I'.
```

```
ENDTRY.
```

When an overflow occurs, the runtime will trigger the exception CX_SY_ARITHMETIC_OVERFLOW.

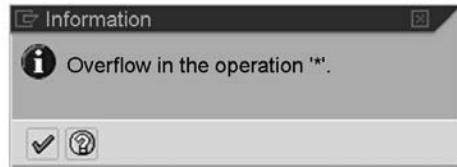


Figure 135: Example Syntax for Handling Predefined Exceptions

In the above calculation, if the value range for data type I is exceeded, the runtime system raises the exception CX_SY_ARITHMETIC_OVERFLOW. This exception is handled by the implemented CATCH block. The reference to the relevant instance is stored in the data object *gx_exc*. The handler can then access the exception text for the instance using the functional method GET_TEXT. The text is stored in the STRING-type data object *ge_text* and then output as an information message.

→ **Note:** The MESSAGE statement was enhanced for *SAP Web Application Server 6.10*, so that any string can be sent as a CLIKE-type data object.

MESSAGE string TYPE message_type.

As well as the *string* message to be output, you must also specify the message type *message_type*.

If the value range for data type I is not exceeded, no exception is raised and the TRY block is processed completely. The processing block is then executed again after the keyword ENDTRY.

The class CX_SY_ARITHMETIC_OVERFLOW is a subclass of the classes CX_SY_ARITHMETIC_ERROR, CX_DYNAMIC_CHECK, and CX_ROOT. Thus the exception raised above can also be handled if you enter one of these classes after the CATCH statement.



Caution: This is a demonstration example that has been deliberately kept simple. Normally, you would define the *ge_result* variable with numerical type P to avoid a runtime error. You could then define the size of the result value before using MOVE to convert to an integer variable if required.

As with the (obsolete) use of CATCH SYSTEM-EXCEPTIONS, this is also about catching runtime errors that **cannot** be **entirely** excluded. A typical example of this can be found in the use of downcast assignments.

The ABAP keyword documentation for each statement lists the classes whose exceptions may occur when that statement is executed.

You can choose between the classic and the class-based exception concept when declaring exceptions for function modules and methods of global classes. Accordingly, as a user you have to catch exceptions with a *sy-subrc* query or with a TRY-ENDTRY statement. The pattern in the ABAP Editor automatically inserts the correct one, providing you have defined this in the ABAP Workbench under *Utilities* → *Settings* → *ABAP Editor* → *Pattrn*.



Pattern functionality in ABAP Editor inserts TRY... ENDTRY:

```

9   *TRY.
10  CALL METHOD cl_salv_table=>factory
11  * EXPORTING
12  *   list_display      = IF_SALV_C_BOOL_SAP=>FALSE
13  * IMPORTING
14  *   r_salv_table      =
15  * CHANGING
16  *   t_table           =
17
18  * CATCH cx_salv_msg .
19  *ENDTRY.

```

Figure 136: Catching Exceptions with the Pattern

The following will show you how to create your own global exception classes.



Defining Global Exception Classes

1. In the *Class Builder*, create a global class as described in the *SAP Library*. When you enter its name, use your customer namespace and then the prefix **CX_**. Choose the option *Exception Class* as the *Class Type*.

As of *SAP NetWeaver 7.0* you can select the *with Message Class* checkbox when you create an exception class. The texts that you then want to appear when exceptions occur can be taken from the message class known to the MESSAGE statement. (You will find these messages and the message classes in the table T100.)

If you do not select this field, or you are working in a system with a release level prior to *SAP NetWeaver 7.0*, you create the exception texts by double-clicking on them in the Class Builder and storing them in the OTR (Online Text Repository).

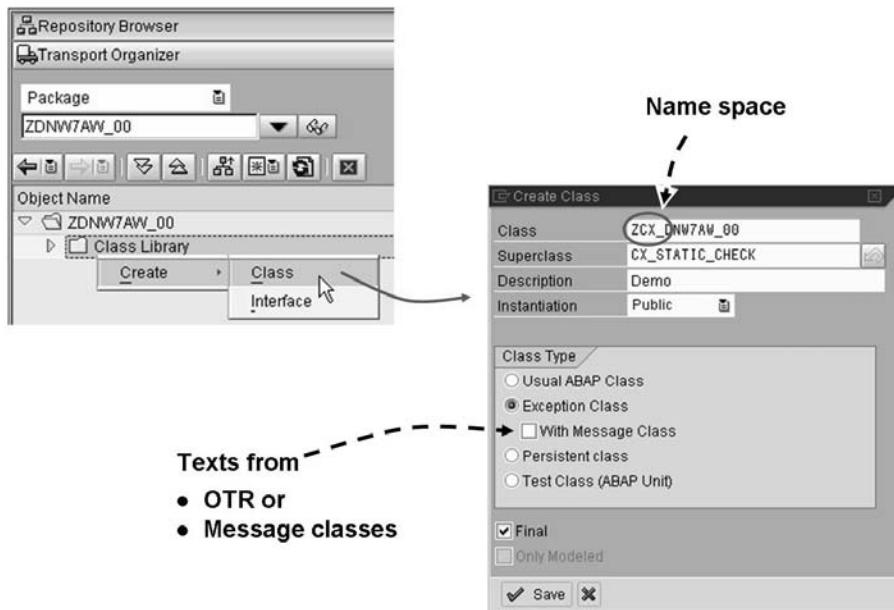


Figure 137: Creating Global Exception Classes

2. Give your exception class additional attributes if necessary (for example, for extensions to exception texts).

With public attributes, the *Class Builder* generates corresponding import parameters for the instance constructor itself, so that these attributes can be set when the exception is raised. The import parameters are generated with the same names as the attributes.

Continued on next page

3. Save as many exception texts as you need. When you do so, you can insert your attributes as parameters in the text in the form &<attribute_name>&.

→ **Note:** For the first text you create yourself, always use a predefined static constant as an ID. It always has the same name as the exception class itself. If no text is specified explicitly when the exception is raised, the text with this ID is used instead.

For all other texts, you define other IDs. The *Class Builder* then generates identically named static constants automatically. When the exception is raised, you pass one of these constants to the import parameter TEXTID to specify the appropriate text for the exception instance.

Additional attributes serve to transport extra information to the handler

As of SAP NetWeaver 7.0, texts from message classes are possible, too

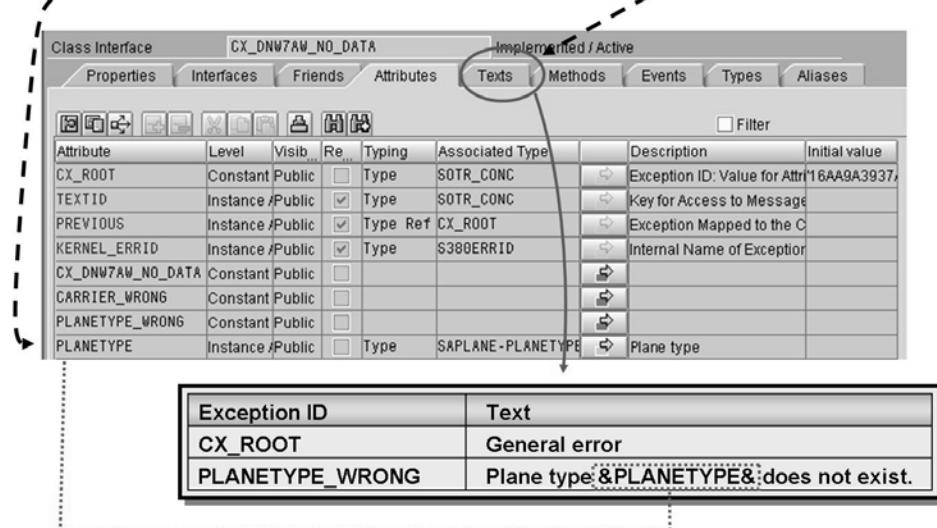


Figure 138: Defining Variable Exception Texts

→ **Note:** Prior to SAP NetWeaver 7.0, the exception texts for global exception classes are stored with their translations in the *Open Text Repository (OTR)*. Note that several texts can be assigned to a single class. You assign a text to an exception using the TEXTID attribute, which contains the globally unique ID of the text object in the *OTR* in an instance at runtime. The GET_TEXT method then reads this text, replaces any text parameters with the contents of the relevant attributes as necessary, and returns the text as a character string.

Continued on next page

4. Activate your exception class.

Propagating and Handling an Exception

Class-based exceptions that occur in procedures do not necessarily need to be handled where they occur; they can be propagated to the calling program of the procedure. The caller can then handle the exception itself or propagate it to its own caller, and so on. The highest levels to which an exception can be propagated are processing blocks without local data areas, that is, event blocks or dialog modules. The exceptions propagated by the called procedures **must** be dealt with there, as must any exceptions raised within this processing block itself. Otherwise a runtime error occurs.

To propagate an exception from a procedure, you generally use the RAISING addition when you are defining the signature for this procedure. In methods of local classes and subroutines, the RAISING addition is specified directly when you define the procedure:

```
METHODS meth_name ... RAISING cx_... cx_... or FORM  
subr_name ... RAISING cx_... cx_....
```

After RAISING come the exception classes whose instances are to be propagated. In methods of global classes, you enter the exception classes whose instances are to be propagated in the exception table for the method in the *Class Builder*. You also need to set the *Exception Class* indicator for each exception table. The process is similar for function modules. To set the indicator in the *Function Builder*, choose the *Exceptions* tab. This clearly shows that a global method or function module can only raise class-based or conventional exceptions.

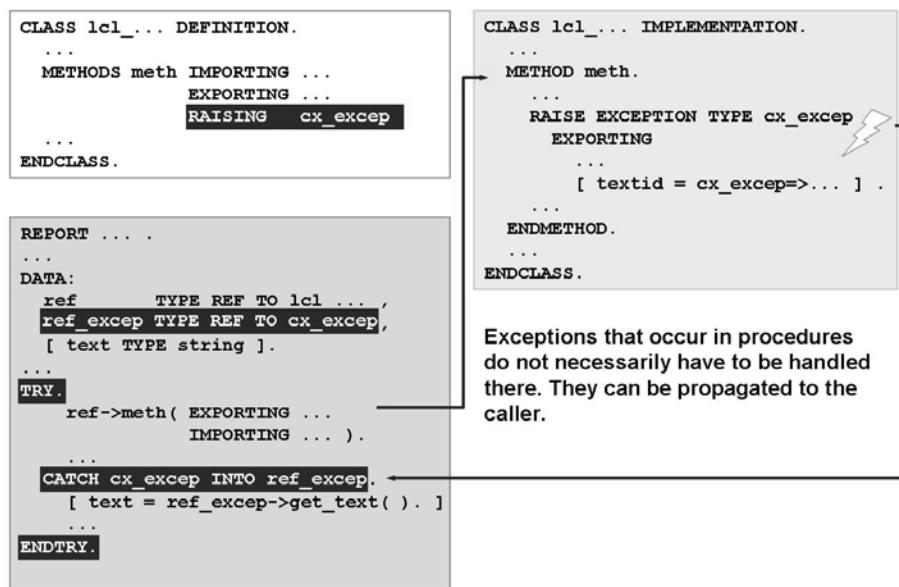


Figure 139: Propagating Exceptions

The TEXTID parameter is optional: The default value is always the static constant, which has the same name as the exception class. However, one of the static constants defined as additional text IDs in the exception class can also be used to supply the value. When the exception is raised, the corresponding text for that exception instance is set. After the exception has been caught, you can then read the text using the functional method GET_TEXT.

If the caller does not catch the exception, it can be propagated upwards to the next-level caller. Before control is passed to the next-level caller, the (optional) CLEANUP block is executed. You then use this block to run the appropriate statements.

Example: Raising, Propagating, and Handling an Exception

```

REPORT sapdnw7aw_exceptions_d1.

DATA: gx_no_data TYPE REF TO cx_dnw7aw_no_data,
      gs_plane     TYPE saplane,
      ge_text      TYPE string.
*-----*
CLASS lcl_planes DEFINITION.
PUBLIC SECTION.
CLASS-METHODS:
  read_plane_details

```

```

IMPORTING ie_planetype TYPE saplane-planetype
      RETURNING value(re_plane) TYPE saplane
      RAISING cx_dnw7aw_no_data.
ENDCLASS. "lcl_planes

*-----*
CLASS lcl_planes IMPLEMENTATION.
METHOD read_plane_details.
  SELECT SINGLE * FROM saplane INTO re_plane
    WHERE planetype = ie_planetype.
  IF sy-dbcnt = 0.
    RAISE EXCEPTION TYPE cx_dnw7aw_no_data
    EXPORTING
      textid      = cx_dnw7aw_no_data->planetype_wrong
      planetype   = ie_planetype.
  ENDIF.
ENDMETHOD.                      "read_plane_details
ENDCLASS. "lcl_planes

START-OF-SELECTION.
TRY.
  gs_plane = lcl_planes->read_plane_details( 'A320' ).
  WRITE: / gs_plane-planetype, gs_plane-seatsmax, gs_plane-producer.
  CATCH cx_dnw7aw_no_data INTO gx_no_data.
    ge_text = gx_no_data->get_text( ).
    WRITE: / ge_text.
ENDTRY.

```

Class-Based Exceptions in Debugging Mode

If an exception is raised, the name of the exception class is displayed in the *Exception Raised* field in the classic debugger, or in the status message in the new Debugger.

If this exception is caught using a CATCH block, this is displayed in the form of a success message. The pointer for the current statement then moves to this CATCH block.

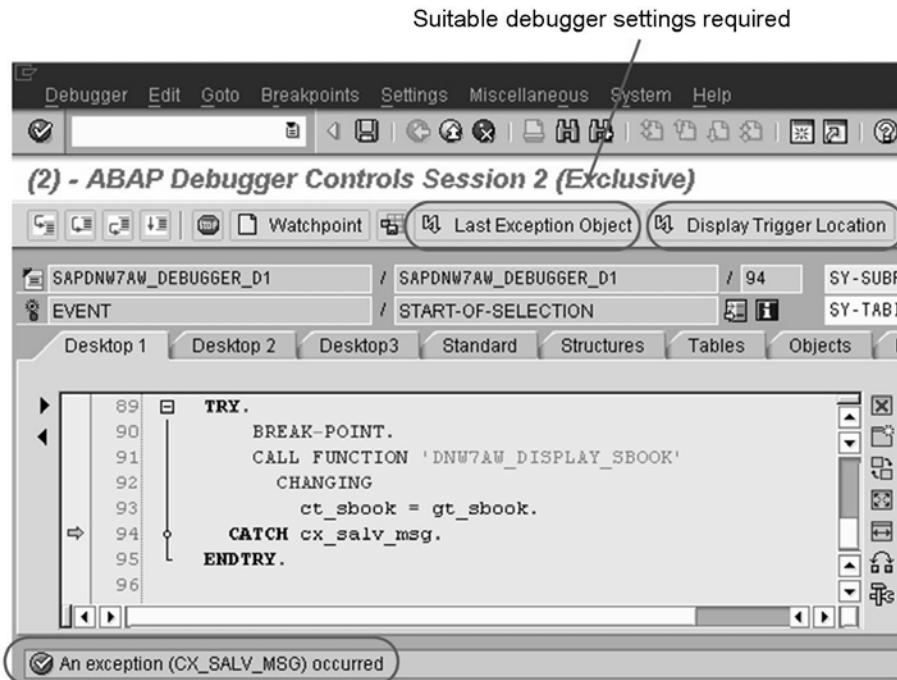


Figure 140: Class-Based Exceptions in Debugging Mode

Two pushbuttons also appear. They allow you to display the values of the attributes of the exception instance and navigate to the point in the source code where the exception occurred. If you have not used the optional INTO <reference> addition with the CATCH statement and want to see an exception instance in the Debugger too, you must select *Always Create Exception Obj.* in the Debugger settings beforehand.

Exception Classes: The Inheritance Hierarchy

The following explains consequences that arise from the choice of an exception class's superclass. It also shows how the exceptions in the runtime system were integrated into the inheritance hierarchy of the exception classes.

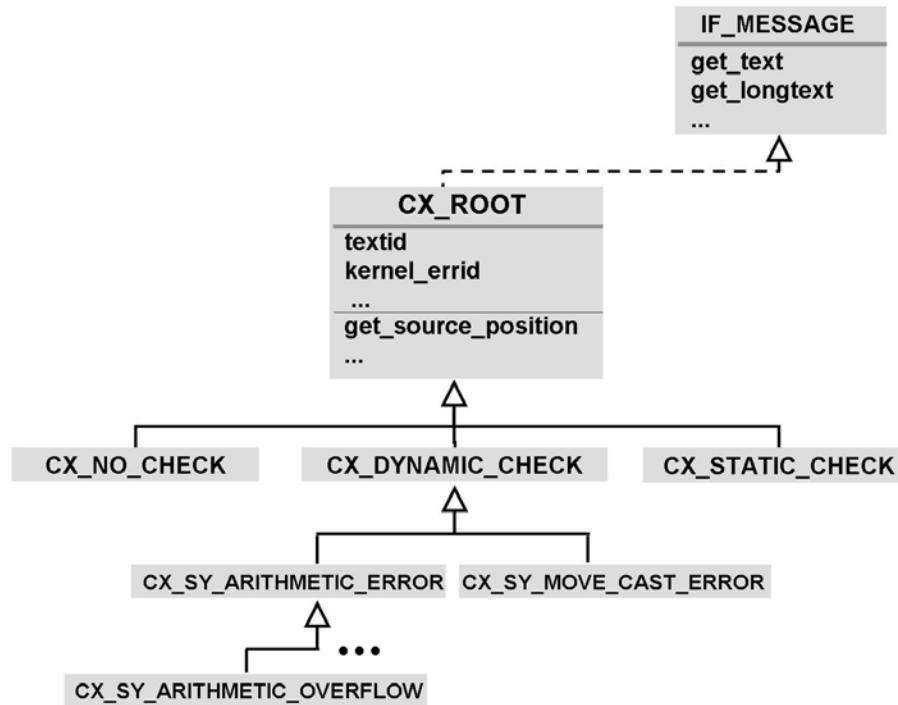


Figure 141: Integration of Standard Exceptions in the Runtime System

For subclasses of CX_STATIC_CHECK, a relevant exception in procedures **must** either be handled **or** propagated explicitly using the RAISING addition. If this is not the case, the **syntax check** displays a warning. If you define your own exception classes, CX_STATIC_CHECK is defined as the superclass by default.

The same applies to subclasses of CX_DYNAMIC_CHECK as to subclasses of CX_STATIC_CHECK: Exceptions based on this **must** either be handled or propagated explicitly with the RAISING addition. The difference is that this is **not** statically checked. **No** syntax warning is reported if the exception is neither handled nor propagated. If the exception is then raised, a **runtime error** occurs. Typical examples of this situation are the predefined exceptions CX_SY_... for errors that occur in the runtime environment. These are usually subclasses of CX_DYNAMIC_CHECK.

For subclasses of CX_NO_CHECK, the rule is that the corresponding exceptions **cannot be propagated explicitly using the RAISING addition**. These exceptions can be handled. Otherwise they are **propagated automatically**. **Neither** a syntax warning **nor** a runtime error is caused directly **where it is raised**. Instead all exceptions that are not handled somewhere in the call hierarchy are propagated **up to the highest call level**. If it is not caught there either, a runtime error occurs **at that point**. Some predefined exceptions with the prefix CX_SY_... for error situations in the runtime environment are subclasses of CX_NO_CHECK.

Mapping Exceptions to Each Other

For the sake of clarity, it should be emphasized that an exception can be propagated through any number of call hierarchies before it is finally handled.

The following section shows how you can **interlink** the raising of exceptions. One exception can raise **a second**, and so on. Each instance should remain valid, regardless of whether or not the corresponding CATCH block has already been left or not. To do this, you must make sure that the previous exception instance can always be accessed by means of at least one reference. The public instance attribute *previous*, inherited by all exception classes from CX_ROOT, provides a convenient way of doing this.

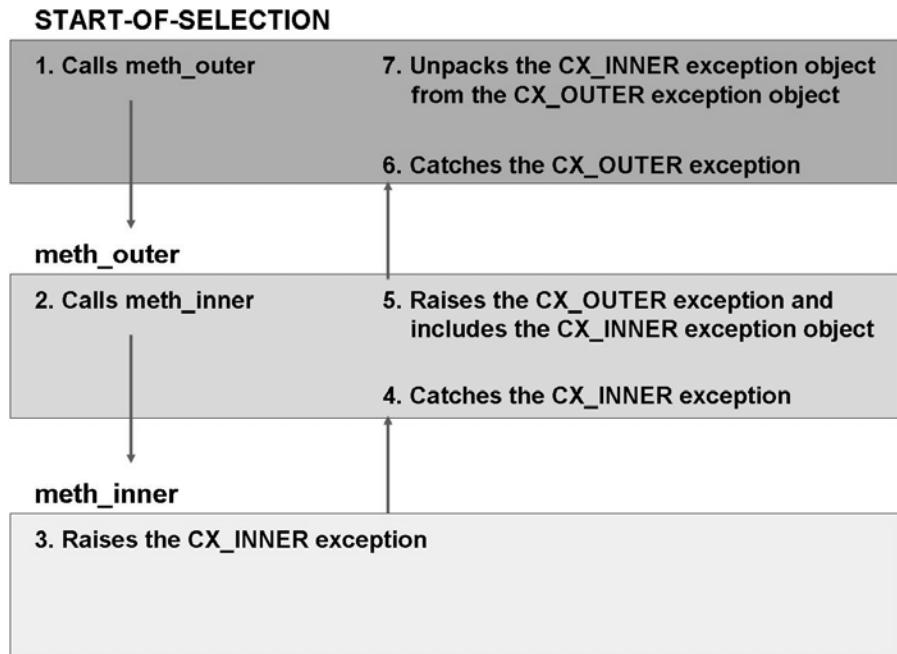


Figure 142: Mapping Exceptions to Each Other – the Principle

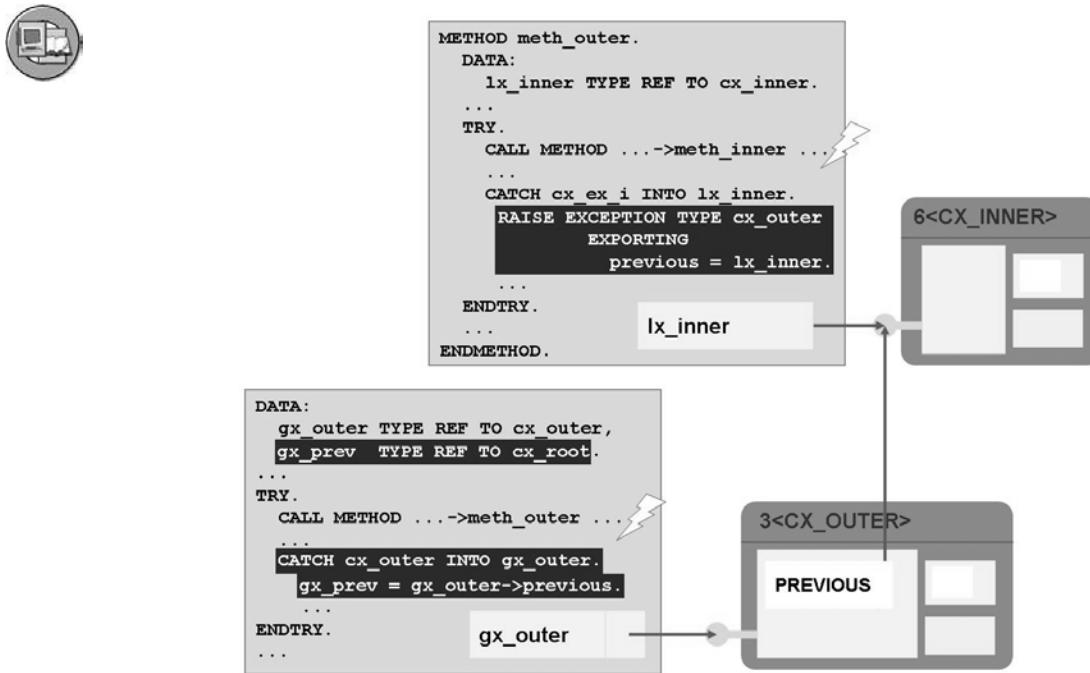


Figure 143: Mapping Exceptions to Each Other

When raising the next exception, only one reference to the previous exception instance must be passed to the new instance. You do this with the optional *previous* parameter for the instance constructor, that is, at `RAISE EXCEPTION`. This **maps** the new exception to the previous one. After the mapped exception has been caught, the *previous* instance attribute contains a reference to the previous exception instance, and so on. In this way, you can access each instance in the chain of exceptions. You can thus follow the history of the raising of the exceptions **in the program**.

Exercise 19: Handle Class-Based Exceptions

Exercise Objectives

After completing this exercise, you will be able to:

- Handle class-based exceptions

Business Example

You use ABAP source code in your company that can raise class-based exceptions. You must therefore be able to react to this can catch these exceptions.

Task 1:

Copy the template program

- Copy the template program SAPDNW7AW_EXCEPTIONS_T1 to ZDNW7AW_EXCEPTIONS_##_1. (## stands for your two-digit group number.) The program is a simple calculator.

Activate your copy.

Task 2:

Cause an exception (dump) to occur

- Make the necessary entries on the selection screen to cause the following errors and take a look at the exceptions that are raised in the dump analysis:

Zero division

Overflow

Character entries instead of numbers. (Admittedly, the developer of the program could have avoided this by placing integer fields on the selection field and not character fields...)



Hint: To analyze dumps afterwards too, start transaction ST22.

Continued on next page

Task 3:

Catch exceptions

1. The calculation of the operation takes place in the program at the AT SELECTION-SCREEN event.

Background: If you catch errors here and send e-messages, users have to change their entries on the selection screen.

Catch exceptions with the class-based exception concept as follows:

In the case of zero division, the following error message should be output: "Even ABAP cannot divide by zero."

In the case of overflow or entry of an invalid operand, the exception object generated by the system should be caught and the result of its method GET_TEXT output as an error message.

Solution 19: Handle Class-Based Exceptions

Task 1:

Copy the template program

1. Copy the template program SAPDNW7AW_EXCEPTIONS_T1 to ZDNW7AW_EXCEPTIONS_##_1. (## stands for your two-digit group number.) The program is a simple calculator.

Activate your copy.

- a) Proceed in the usual manner.

Task 2:

Cause an exception (dump) to occur

1. Make the necessary entries on the selection screen to cause the following errors and take a look at the exceptions that are raised in the dump analysis:

Zero division

Overflow

Character entries instead of numbers. (Admittedly, the developer of the program could have avoided this by placing integer fields on the selection field and not character fields...)



Hint: To analyze dumps afterwards too, start transaction ST22.

- a) Start the program with **F8**

Task 3:

Catch exceptions

1. The calculation of the operation takes place in the program at the AT SELECTION-SCREEN event.

Background: If you catch errors here and send e-messages, users have to change their entries on the selection screen.

Catch exceptions with the class-based exception concept as follows:

In the case of zero division, the following error message should be output: "Even ABAP cannot divide by zero."

Continued on next page

In the case of overflow or entry of an invalid operand, the exception object generated by the system should be caught and the result of its method GET_TEXT output as an error message.

- a) See model solution

Result

Model solution

```
REPORT  sapdnw7aw_exceptions_s1.

DATA:
  ge_result  TYPE p DECIMALS 2,
  ge_message TYPE string,
  gx_error    TYPE REF TO cx_root.

PARAMETERS:
  pa_int1    TYPE c          LENGTH 10 DEFAULT '9999999999',
  pa_op      TYPE dnw7aw_operator OBLIGATORY DEFAULT '*' VALUE CHECK,
  pa_int2    TYPE c          LENGTH 10 DEFAULT '9999999999'.

AT SELECTION-SCREEN.
TRY.
CASE pa_op.
  WHEN '+'.
    ge_result = pa_int1 + pa_int2.
  WHEN '-'.
    ge_result = pa_int1 - pa_int2.
  WHEN '*'.
    ge_result = pa_int1 * pa_int2.
  WHEN '/'.
    TRY.
      ge_result = pa_int1 / pa_int2.
    CATCH cx_sy_zerodivide.
      MESSAGE 'Even ABAP cannot divide by zero'(zer) TYPE 'E'.
    ENDTRY.
ENDCASE.

CATCH  cx_sy_arithmetic_overflow
       cx_sy_conversion_no_number
```

Continued on next page

```
INTO gx_error.  
ge_message = gx_error->get_text( ).  
MESSAGE ge_message TYPE 'E'.  
  
ENDTRY.  
  
START-OF-SELECTION.  
WRITE: 'Result:'(res), ge_result.
```


Exercise 20: Create and Use Global Exception Classes

Exercise Objectives

After completing this exercise, you will be able to:

- Define global exception classes
- Declare, raise, and handle class-based exceptions

Business Example

You want to try out the class-based exception concept using a small example.

Task 1:

Create a global exception class

1. Create a global exception class called ZCX_DNW7AW_## . (## stands for your two-digit group number.) Choose the superclass so that the syntax check determines whether the relevant exception is handled or explicitly propagated. The error texts should be derived from a message class.
2. When the exception is raised, the name of an airline should be transferred to the exception object. You therefore need to create a public instance attribute with the name *CARRIER* and type it with reference to SCARR-CARRID.
3. Each class automatically has a text ID with the same name as the class itself. Determine that the message 005 from the message class DNW7AW is used for your text ID *ZCX_DNW7AW_##*.
4. Create another text ID and name it *CARRIER_WRONG*. Determine that message 006 from message class DNW7AW is used for this text ID. This message has a placeholder (&1). The attribute *CARRIER* is to be used in its place.
5. Activate your exception class.

Task 2:

Copy the template program

1. Copy the template program SAPDNW7AW_EXCEPTIONS_T2 to ZDNW7AW_EXCEPTIONS_##_2. (## stands for your two-digit group number.) Activate your copy and familiarize yourself with the various functions.

Continued on next page

Task 3:

Declare and raise an exception

1. Extend the signature for the READ_CARRIER_DETAILS method in such a way that an exception of your class ZCX_DNW7AW_## can be raised.
2. Extend the implementation of the method READ_CARRIER_DETAILS in such a way that a type ZCX_DNW7AW_## exception is raised if no relevant airline is found with the database select. Use *CARRIER_WRONG* for the text ID. The abbreviation for the airline should also be transferred.
3. Catch the possible exception at START-OF-SELECTION. If an error occurs, call the GET_TEXT method and output the resulting text on the list.

Solution 20: Create and Use Global Exception Classes

Task 1:

Create a global exception class

1. Create a global exception class called **ZCX_DNW7AW_##**. (## stands for your two-digit group number.) Choose the superclass so that the syntax check determines whether the relevant exception is handled or explicitly propagated. The error texts should be derived from a message class.
 - a) Display the object list for your package in transaction SE80. Then click on your package with the right mouse-button and choose *Create → Class Library → Class*.
 - b) On the following screen, enter the name **ZCX_DNW7AW_##**. Enter a suitable description. Choose *Exception Class* and *with Message Class*. Save the class.
2. When the exception is raised, the name of an airline should be transferred to the exception object. You therefore need to create a public instance attribute with the name **CARRIER** and type it with reference to SCARR-CARRID.
 - a) In the Object Navigator, go to the *Attributes* tab page in the detail view for your class. Position the cursor on the first free field in the *Attribute* column and enter: **CARRIER INSTANCE PUBLIC TYPE SCARR-CARRID** in the relevant columns. Press Enter.
3. Each class automatically has a text ID with the same name as the class itself. Determine that the message 005 from the message class DNW7AW is used for your text ID **ZCX_DNW7AW_##**.
 - a) In the Object Navigator, go to the *Texts* tab page in the detail view for your class. Place the cursor on the exception ID **ZCX_DNW7AW_##** and click on *Message Text*. On the following screen, enter the message class DNW7AW and the message 005. Save your entries.

Continued on next page

4. Create another text ID and name it *CARRIER_WRONG*. Determine that message 006 from message class DNW7AW is used for this text ID. This message has a placeholder (&1). The attribute *CARRIER* is to be used in its place.
 - a) In the Object Navigator, go to the *Texts* tab page in the detail view for your class. Place the cursor on the first empty field in the *Exception ID* column and enter **CARRIER_WRONG**. Click on *Message Text*. On the following screen, enter the message class DNW7AW and the message 005. For *Attribute 1*, select the entry *CARRIER*. Save your entries.
5. Activate your exception class.
 - a) Click on or choose **Ctrl + F3**.

Task 2:

Copy the template program

1. Copy the template program SAPDNW7AW_EXCEPTIONS_T2 to ZDNW7AW_EXCEPTIONS_##_2. (## stands for your two-digit group number.) Activate your copy and familiarize yourself with the various functions.
 - a) Proceed in the usual manner.

Task 3:

Declare and raise an exception

1. Extend the signature for the READ_CARRIER_DETAILS method in such a way that an exception of your class ZCX_DNW7AW_## can be raised.
 - a) See model solution
2. Extend the implementation of the method READ_CARRIER_DETAILS in such a way that a type ZCX_DNW7AW_## exception is raised if no relevant airline is found with the database select. Use *CARRIER_WRONG* for the text ID. The abbreviation for the airline should also be transferred.
 - a) See model solution
3. Catch the possible exception at START-OF-SELECTION. If an error occurs, call the GET_TEXT method and output the resulting text on the list.
 - a) See model solution

Result

Model solution

Continued on next page

```

REPORT  sapdnw7aw_exceptions_s2.

DATA: gx_no_data TYPE REF TO cx_dnw7aw_no_data,
      gs_carrier TYPE scarr,
      ge_text      TYPE string.
*-----
CLASS lcl_carrier DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      read_carrier_details
        IMPORTING ie_carrid TYPE scarr-carrid
          RETURNING value(re_carrier) TYPE scarr
          RAISING cx_dnw7aw_no_data.

ENDCLASS. "lcl_carrier

*-----
CLASS lcl_carrier IMPLEMENTATION.
  METHOD read_carrier_details.
    SELECT SINGLE * FROM scarr INTO re_carrier
      WHERE carrid = ie_carrid.
    IF sy-dbcnt = 0.
      RAISE EXCEPTION TYPE cx_dnw7aw_no_data
        EXPORTING
          textid = cx_dnw7aw_no_data->carrier_wrong
          carrier = ie_carrid.
    ENDIF.
    ENDMETHOD.                      "read_carrier_details
  ENDCLASS. "lcl_carrier

*-----
*START-OF-SELECTION.

TRY.
  gs_carrier = lcl_carrier->read_carrier_details( 'XY' ).
  WRITE:
    / gs_carrier-carrid,
    gs_carrier-carrname,
    gs_carrier-currcode.
  CATCH cx_dnw7aw_no_data INTO gx_no_data.
    ge_text = gx_no_data->get_text( ).
    WRITE: / ge_text.
ENDTRY.

```



Lesson Summary

You should now be able to:

- Handle class-based exceptions in ABAP programs
- Create exception classes
- Raise class-based exceptions in ABAP programs
- Propagate class-based exceptions in ABAP programs
- Map class-based exceptions in ABAP programs to one another



Unit Summary

You should now be able to:

- Handle class-based exceptions in ABAP programs
- Create exception classes
- Raise class-based exceptions in ABAP programs
- Propagate class-based exceptions in ABAP programs
- Map class-based exceptions in ABAP programs to one another



Test Your Knowledge

1. The new exception concept replaces the old one. All the old source code sections must therefore be rewritten. As of *SAP Web Application Server 6.20*, function modules from the SAP standard system automatically raise object-oriented exceptions.

Determine whether this statement is true or false.

- True
- False

2. Unlike the old exceptions, those embodying the new concept can also be raised from subroutines and propagated.

Determine whether this statement is true or false.

- True
- False

3. The new exception classes can only be defined globally. This ensures central maintenance and reuse.

Determine whether this statement is true or false.

- True
- False

4. When you define an exception class, the superclass you choose specifies whether or not its exceptions must be caught explicitly with TRY-CATCH-ENDTRY, and if so, how the system reacts if the exception is not caught.

Determine whether this statement is true or false.

- True
- False



Answers

1. The new exception concept replaces the old one. All the old source code sections must therefore be rewritten. As of *SAP Web Application Server 6.20*, function modules from the SAP standard system automatically raise object-oriented exceptions.

Answer: False

Function modules can use either the old or the new exception concept.

2. Unlike the old exceptions, those embodying the new concept can also be raised from subroutines and propagated.

Answer: True

Refer to the relevant section of the lesson.

3. The new exception classes can only be defined globally. This ensures central maintenance and reuse.

Answer: False

Refer to the relevant section of the lesson.

4. When you define an exception class, the superclass you choose specifies whether or not its exceptions must be caught explicitly with TRY-CATCH-ENDTRY, and if so, how the system reacts if the exception is not caught.

Answer: True

Refer to the relevant section of the lesson.

Unit 5

Shared Objects

Unit Overview

This unit tells you how to realize shared objects, a special memory area that can be used by several users at the same time. It was made available with *SAP NetWeaver 2004*.



Unit Objectives

After completing this unit, you will be able to:

- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program

Unit Contents

Lesson: Shared Objects	328
Exercise 21: Shared Objects	351

Lesson: Shared Objects

Lesson Overview

As of *SAP NetWeaver 2004*, you can store data in the form of shared objects in the shared memory. In this lesson, we will examine this concept and its implementation.

The advantage of this technique is that data can be saved as objects. Programs can access this data quickly, independently of the user session. This technique was developed with the primary objective of saving catalog information and shopping carts. An attribute of catalogs is that they are read often, but changed rarely. In the case of the shopping cart, the data is written by one application and then read by another program at a different time (in a different user session). Here as well, this technique delivers considerably better performance than saving the data persistently on the database.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program

Business Example

Mr. Jones is a software developer at a major corporation that develops proprietary business applications in ABAP. He is asked to develop a flexible new application for flight bookings. One requirement of the application is that many employees must be able to access the same customer data simultaneously, without having to read this data from the database each time. Mr. Jones knows that shared objects can be used to make data in the main memory accessible across session boundaries. He therefore uses shared objects to develop his flight booking application.

Introducing Shared Objects

As of *SAP NetWeaver 2004*, you can store data in the form of shared objects across programs and user sessions in the shared memory. This enables you to create applications in which users write data to this area. Other users can then read this data later.

It is possible to imagine many different potential uses for shared objects:



- Storing a **catalog**

An “author” writes the catalog to the shared objects area – many users can then access this catalog at the same time.

- Saving a **shopping cart**

The buyer fills the shopping cart and the seller reads it later.

Shared Memory

Shared memory is a memory area on an application server that can be accessed by all the ABAP programs running on that server.

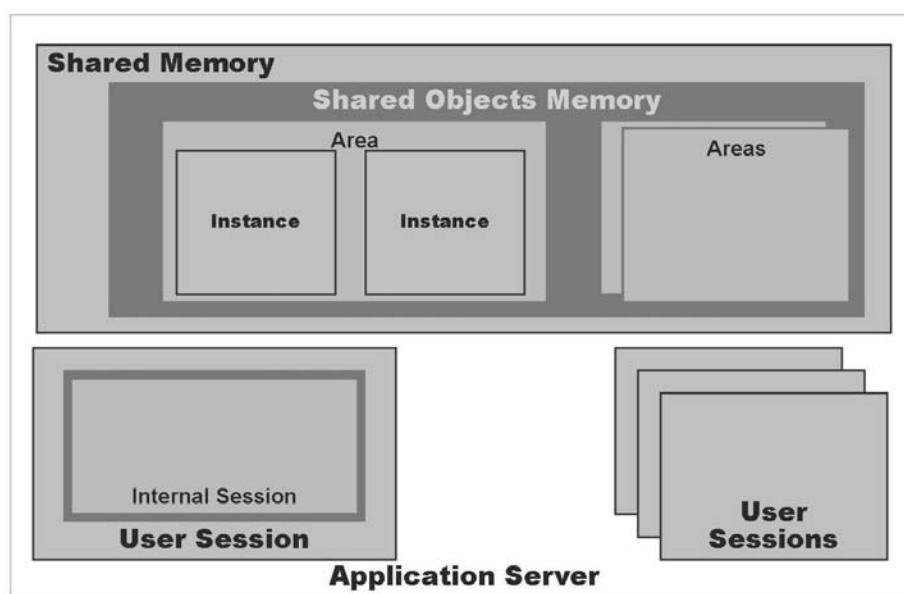


Figure 144: Memory Model of an Application Server

Before shared objects were introduced, ABAP statements had to use the EXPORT and IMPORT statements with the SHARED BUFFER and SHARED MEMORY additions to access this memory area. Instances of classes "lived" exclusively in the internal session

of an ABAP program. With the introduction of shared objects in *SAP NetWeaver 2004*, shared memory has been enhanced with **Shared Objects Memory**, where the shared objects can be stored. These shared objects are saved in areas of shared memory.

→ **Note:** *SAP NetWeaver 2004* only supports the storage of class instances.

As of *SAP NetWeaver 7.0*, you can store data references that point to data objects of the same area instance version in the shared memory. You can use the following types for the anonymous data object: all visible data types of global interfaces and classes; structures and database tables in the ABAP Dictionary; data types from type groups. However, it is not (yet) possible to create data references in the shared memory whose data objects are typed to data elements or table types.

It is not (yet) possible to save any data objects as shared objects. However, data objects other than reference variables can be stored as attributes of classes.

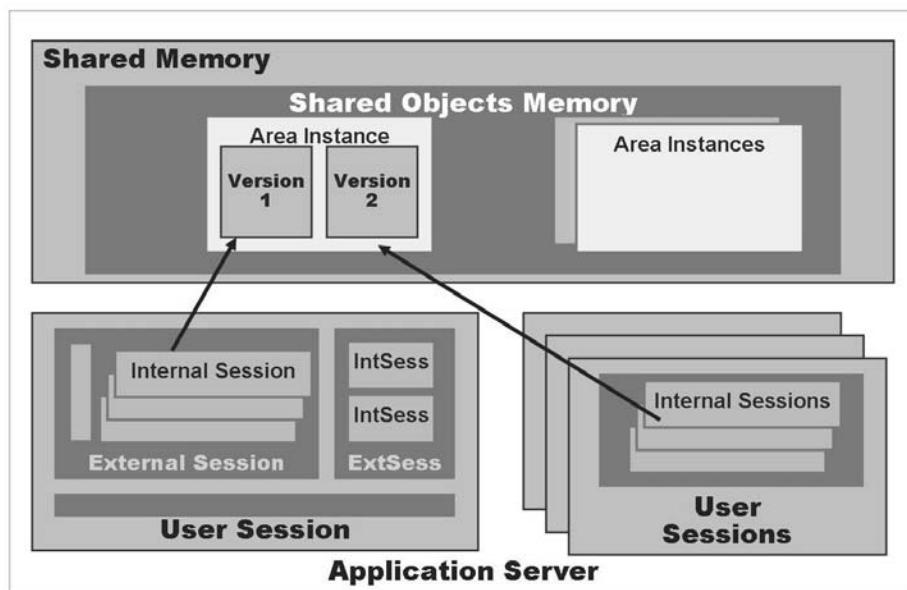


Figure 145: Accessing Shared Objects

Attributes of Shared Objects



- Cross-program buffering of data that is often read, but rarely written.
(Recommendation: Once per day to once per hour.)
- Simultaneous read accesses are supported.
- Access is regulated by a lock mechanism.
- Data is stored as object attributes.
- Memory bottlenecks result in runtime errors and have to be caught.

Write accesses need to be seldom because writing data to the shared objects area is performance-intensive. It is runtime that you want to optimize here, though, which would be lost if write access were more frequent.

→ **Note:** Shared objects are also used internally by SAP. For example, this technique is used to navigate in the *ABAP Workbench*. In addition to saving memory (around 3 MB per user logon), navigation during the first access is faster by up to a factor of 100.

A prerequisite for saving an object in shared memory is that the class of that object is defined with the SHARED MEMORY ENABLED addition of the CLASS statement, or that the *Shared Memory Enabled* attribute is selected in the *Class Builder*.

Areas and Area Instances

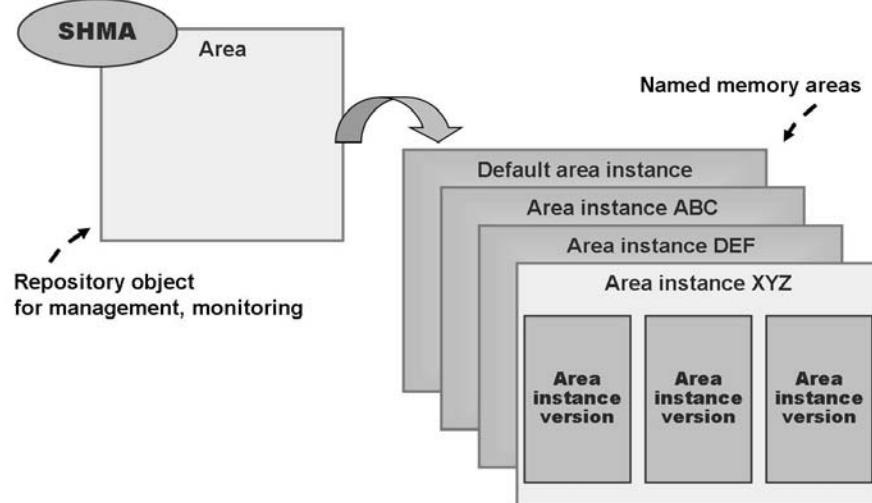


Figure 146: Areas and Area Instances

An area is a template for area instances in the shared memory. Several area instances with different names can be created from one area. In addition, an area instance can have several versions (“area instance versions”), which differ in their version IDs. In the simplest case, without version management, an area instance consists of a single area instance version.

Area Classes and Area Handles

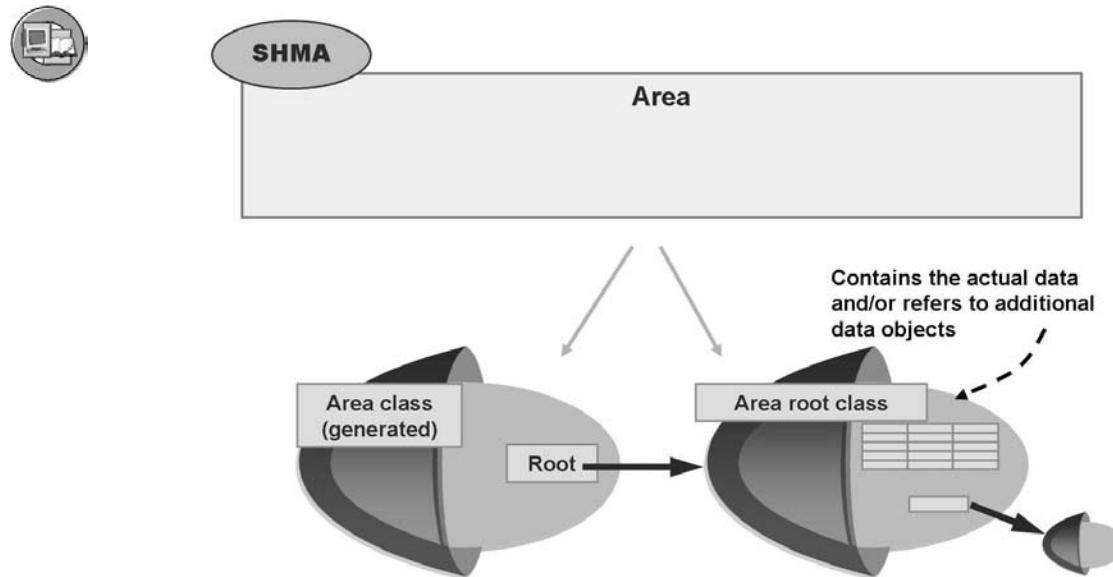


Figure 147: Creating an Area

You define an area in transaction SHMA. This creates a global, final area class **of the same name** as a subclass of CL_SHM_AREA. In an ABAP program, the area is accessed exclusively using methods of the generated area class.



Hint: Because the area and area class have the same name, it makes sense to name the area according to the naming conventions for classes –ZCL_*, for example, in the customer namespace.

You can use static methods (“Attach” methods) of an area class to connect an ABAP program (or its internal session, where the ABAP program is processed) to an area instance in shared memory. When you connect an ABAP program, an instance of the area class is created as an area handle.

The above diagram also shows another class, which is called the area root class. You can create any number of objects in an area instance, depending on your specific application. You access these objects uniformly through the instance of the area root class.

→ **Note:** We will limit ourselves to two classes in a specific example.

Developing an Example Application

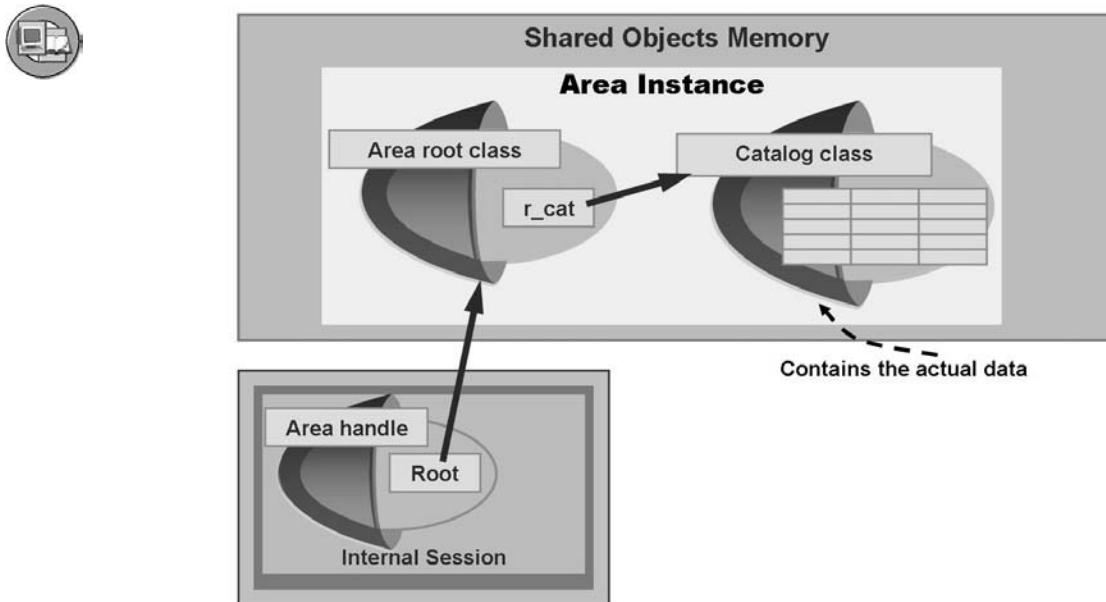


Figure 148: Example Application

In this lesson, we will develop a simple catalog application as an example. We want to use shared objects to create a catalog of flight dates, where any user can select a flight.

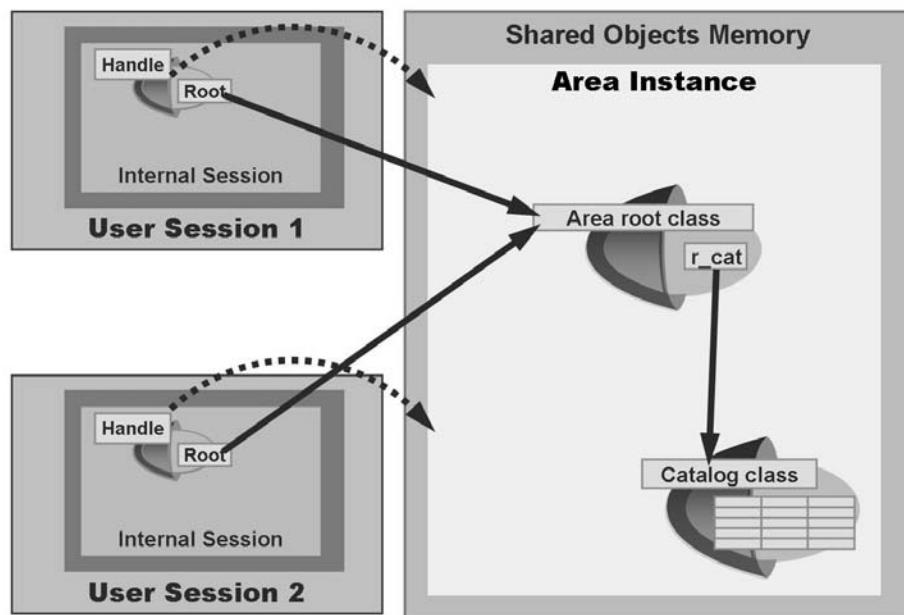


Figure 149: Accessing Areas

The diagram again illustrates the fact that any programs can access objects in the shared objects memory. In this case, two applications, which run in different user sessions, access objects in the same area. We therefore need several things for the example application:

- Create an area
- Develop a program to create an area instance
- Develop a program to read data from the area

Creating an Area

Shared objects are stored in areas of shared memory. You use transaction SHMA to create and manage areas and their attributes.

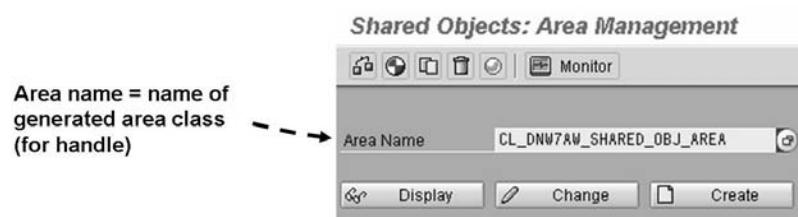


Figure 150: Area Management (Transaction SHMA)

Call the transaction code SHMA and enter the name of the area. The usual namespace rules apply, which means the area name has to start with Y or Z in the customer system. Namespaces containing slashes are also supported.



Hint: Note that a global, final area class with the same name as the area is also created as a subclass of CL_SHM_AREA. It therefore makes sense to choose the area name wisely, as shown in our example.

In an ABAP program, the area will be accessed later using only the methods of the generated area class.

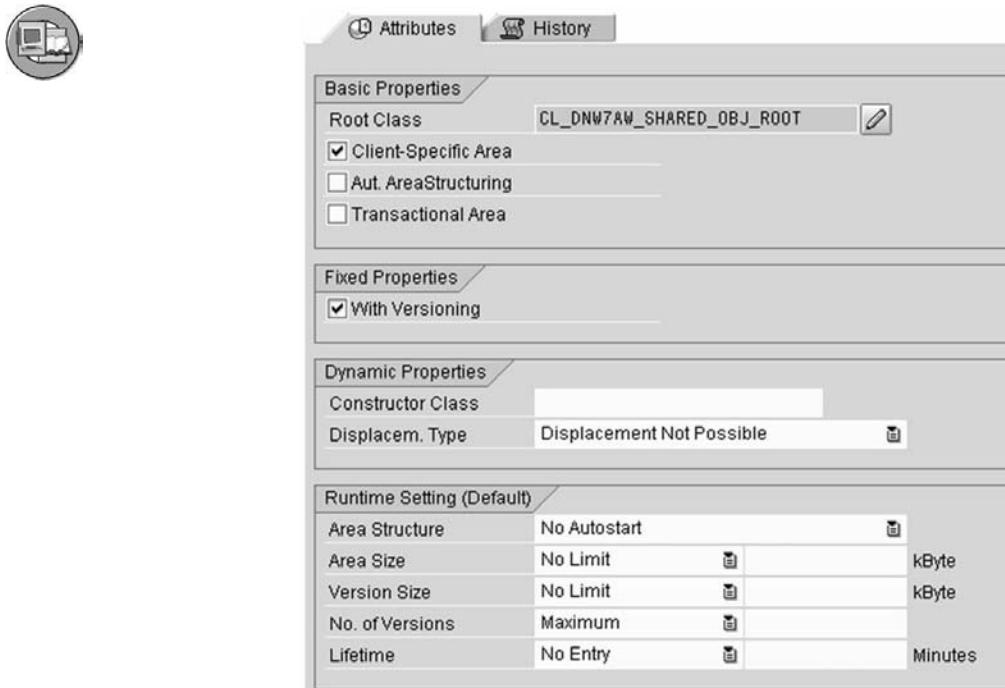


Figure 151: Maintaining Areas

After you press the Create, Change, or Display buttons, the maintenance screen for areas appears.

Each area is linked with a global **area root class**, whose attributes can contain proprietary data and references to other shared memory-enabled classes. The area root class has to be assigned to an area when you maintain it. If an area instance version is not empty, it has to contain at least one instance of the area root class as its root object, which is used to reference other objects. When you generate the area class, a *ROOT* attribute is generated and typed with the static type of the area root class.

Other Important Terms when Creating an Area

Client-dependent area

Areas, and thus the objects within an area, do not have a client ID by default. You can specify an area as client-dependent, however. In client-dependent areas, the methods of the area class for accessing an area instance refer to the active client by default. You can use the optional importing parameter CLIENT to access another client explicitly.

Transactional area

An area instance version of a transactional area is not active immediately after you remove a change lock with the DETACH_COMMIT method; it is only active after the next database commit.

This is particularly helpful in implementing shopping carts in the shared objects memory.

Setting Up an Area Instance

In the last section, we defined an area. We also created the classes that will be instantiated in this area. In this section, we will examine the statements that we can use to create an area **instance**. We will continue to use the example introduced above.

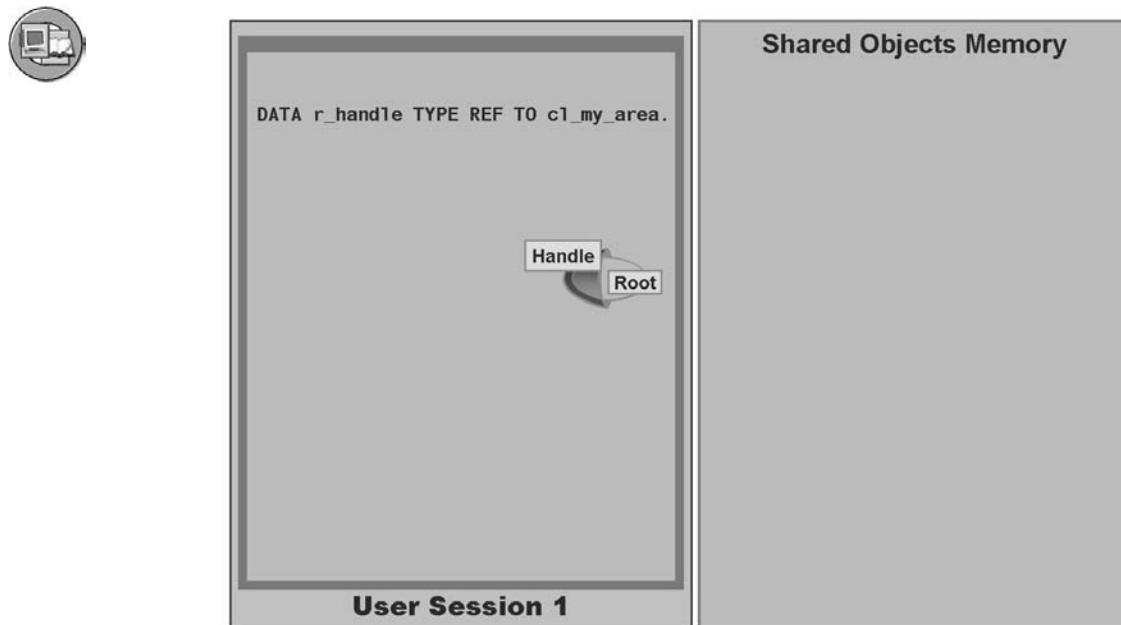


Figure 152: Before Creating an Area Instance

When you create an area, a global final class with the same name is also created. To set up an area or access an existing area, you need a reference variable that is typed with the generic area class. This reference serves as a handle for accessing the area.

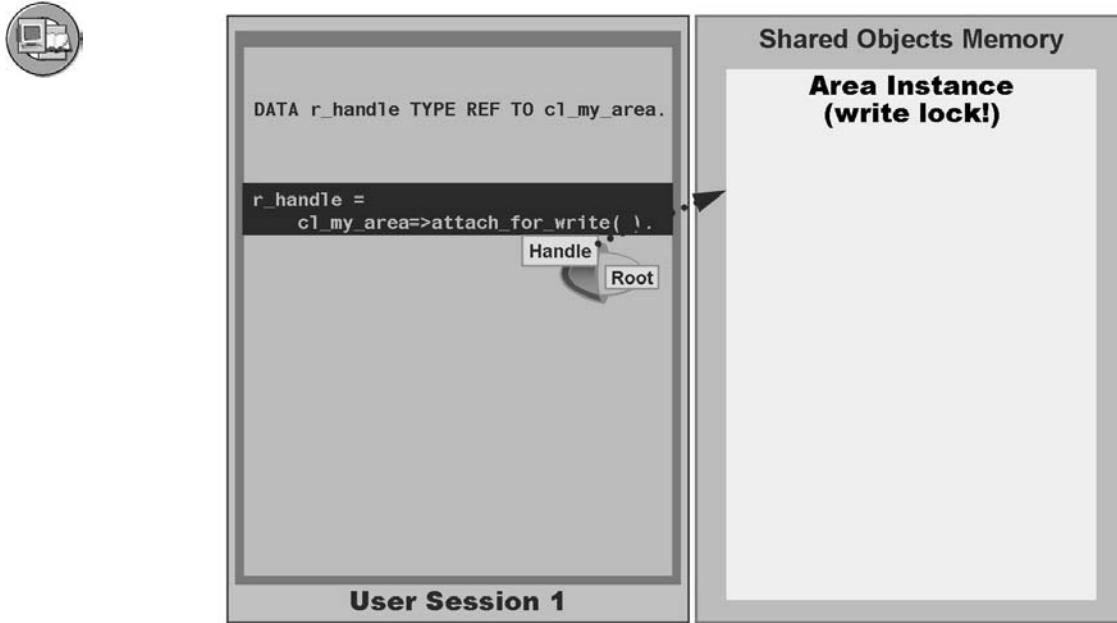


Figure 153: Creating an Area Instance

When you instantiate the area class, an instance of the area is created in the shared memory. The program has a handle for this instance of the area. All future operations are performed using this handle.

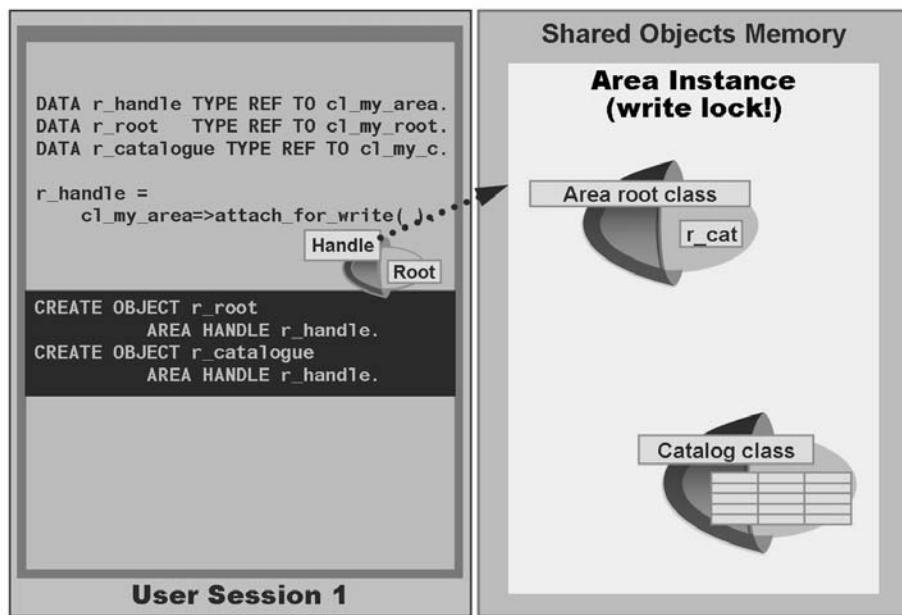


Figure 154: Generating Objects in the Shared Memory

Once the area instance has been created, the objects can be created in the shared objects memory. To do so, use the AREA HANDLE addition for the CREATE OBJECT statement. This tells the system the area instance where the objects will be created.

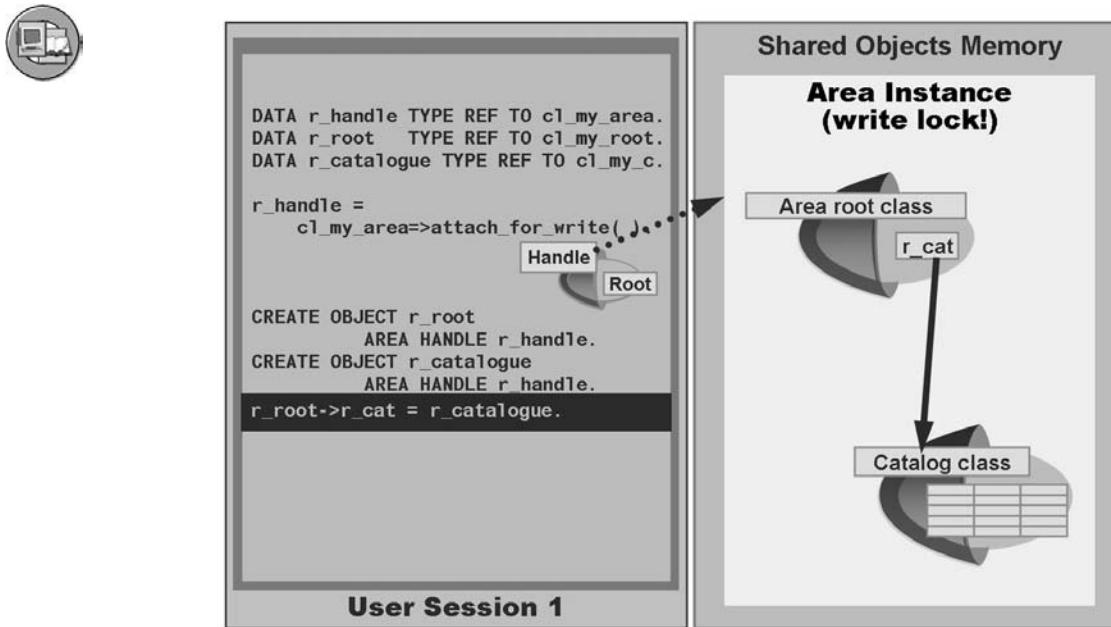


Figure 155: Generating Objects in the Shared Memory II

In the above diagram, both objects are instantiated from within the program. Alternatively, you can instantiate the root object from within the program. You can also create the other objects in this area instance from the constructor of the root object.

You do not have to assign the references in this case.

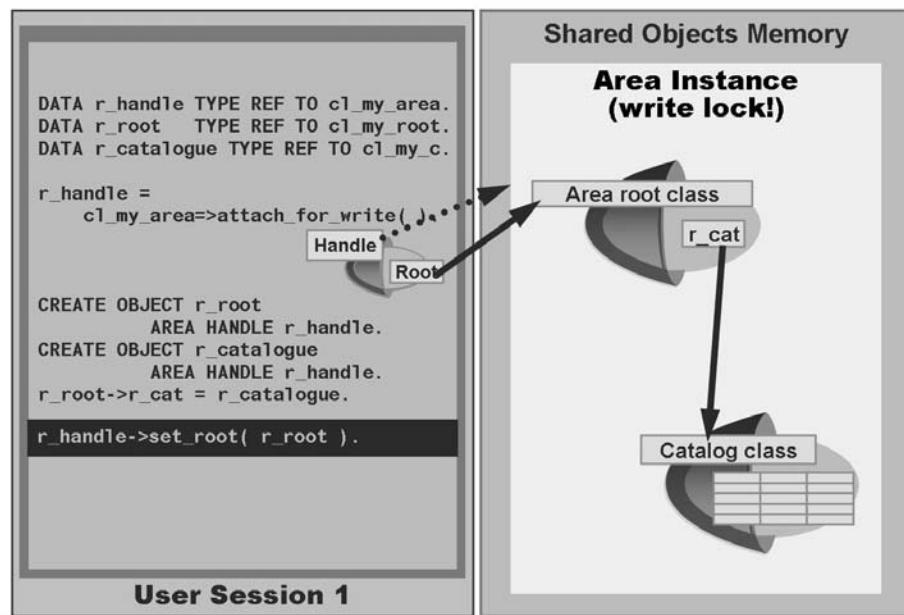


Figure 156: Setting the Root Object

To address the objects created in the area instance, you have to assign the root object to the ROOT attribute of the area handle. To do so, you use the SET_ROOT method of the area handle.

As a consequence, any program from any application can access this area. To do so, the application merely has to fetch a reference to the area instance, and can then immediately access the objects contained in that area instance.

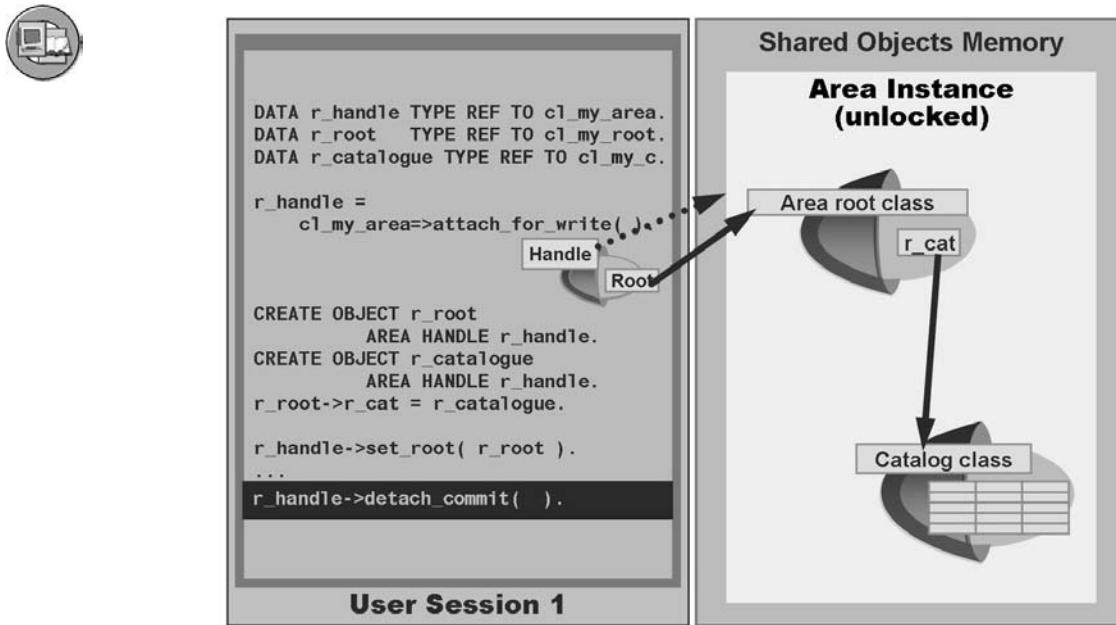


Figure 157: Releasing the Write Lock

Read access to an area instance is only possible when the write lock has been canceled. To do so, use the DETACH_COMMIT method, which inherits the area class from class CL_SHM_AREA.

Accessing an Area Instance

Once an area instance has been set up, any other users and applications can access it. The read programs have to implement the following steps:

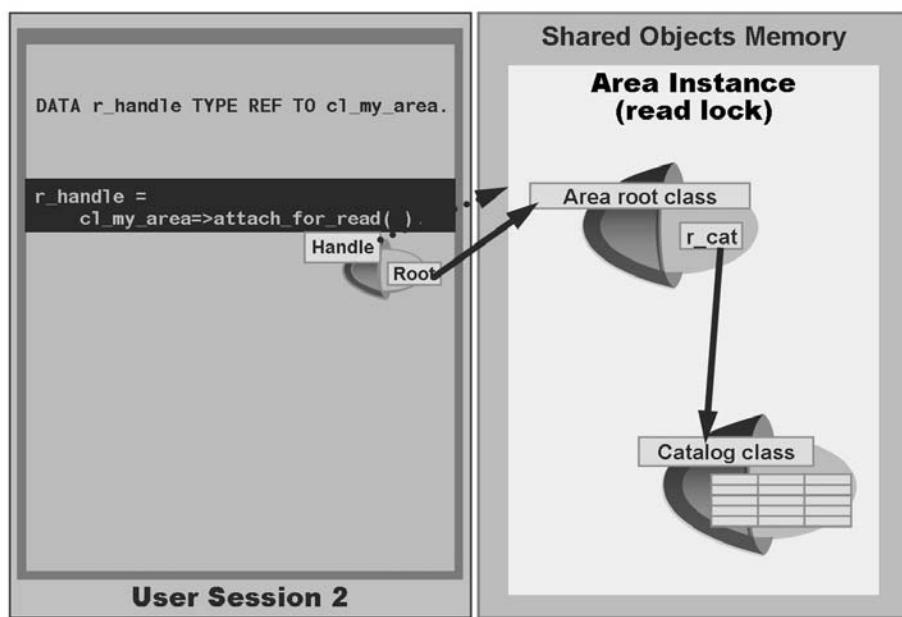


Figure 158: Accessing an Existing Object in the Shared Memory

The read program first needs a reference variable that is typed with the area class. This reference variable serves as a handle for the area instance that is being accessed.

The program also has to get the handle of the area instance. This is performed using method ATTACH_FOR_READ, which is provided by the class CL_SHM_AREA. This sets a read lock that prevents the area instance from being erased during the access.

The objects in this area instance can now be accessed. They are always accessed using the area handle.

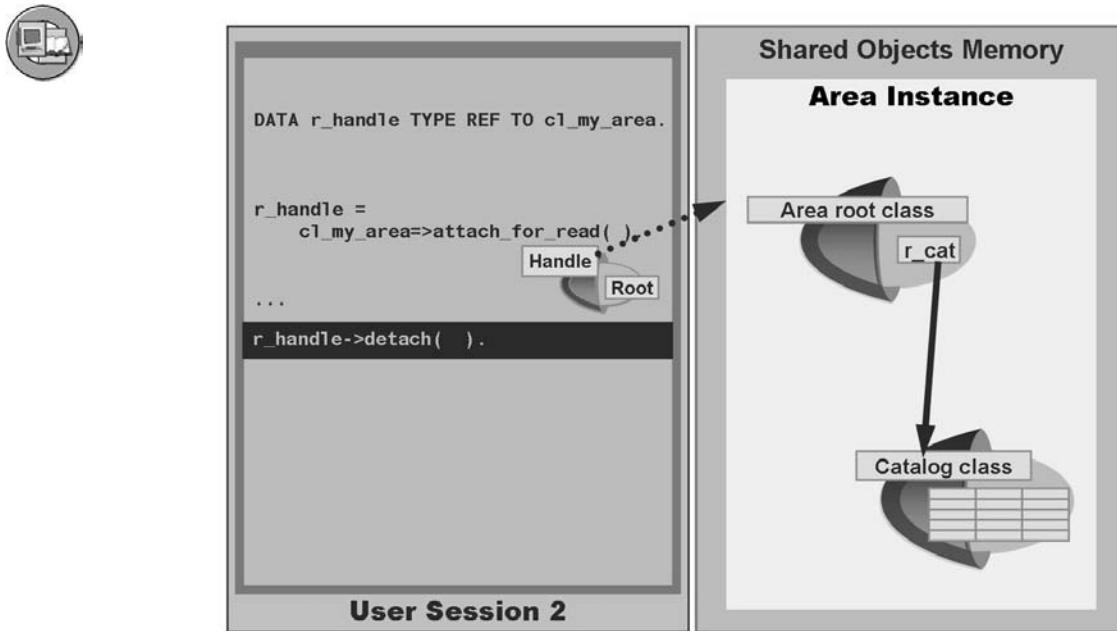


Figure 159: Releasing the Read Lock

Once the read activity has been completed, the application releases the read lock. You can use the DETACH method for the area handle for this purpose. The read lock is also released automatically when the internal session is closed.

States of Area Instance Versions

When you create an area, you can specify that several versions of an area instance are allowed. What exactly do these area instance versions mean? To answer this question, we will examine an example in the next section.

To monitor area instances for shared objects, you use the transaction SHMM.

Active Version

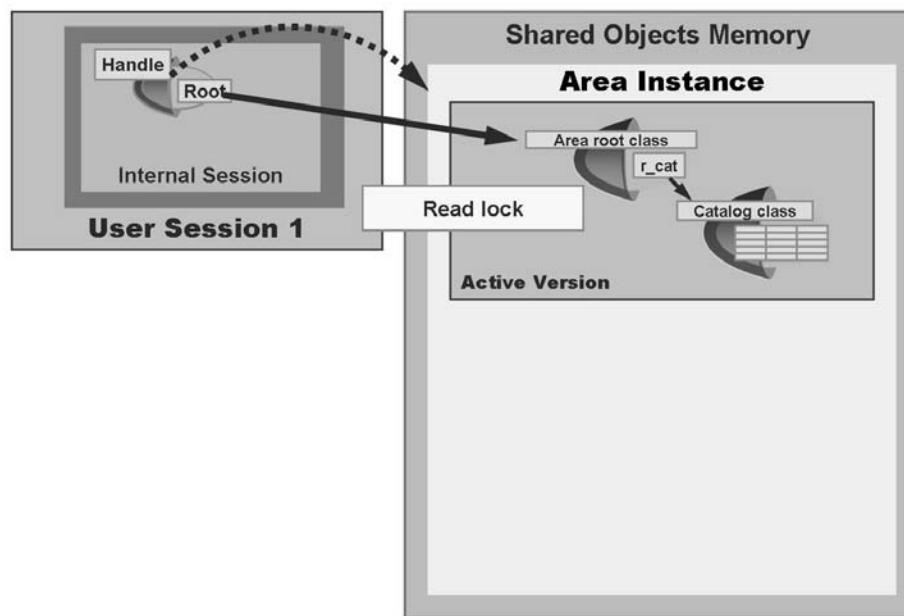


Figure 160: Setting a Read Lock on the Active Version

This is the normal state: An area instance has been set up. Once the setup is complete (using the method DETACH_COMMIT) and the system sends a database commit, the area instance version is active.

Version Being Set Up

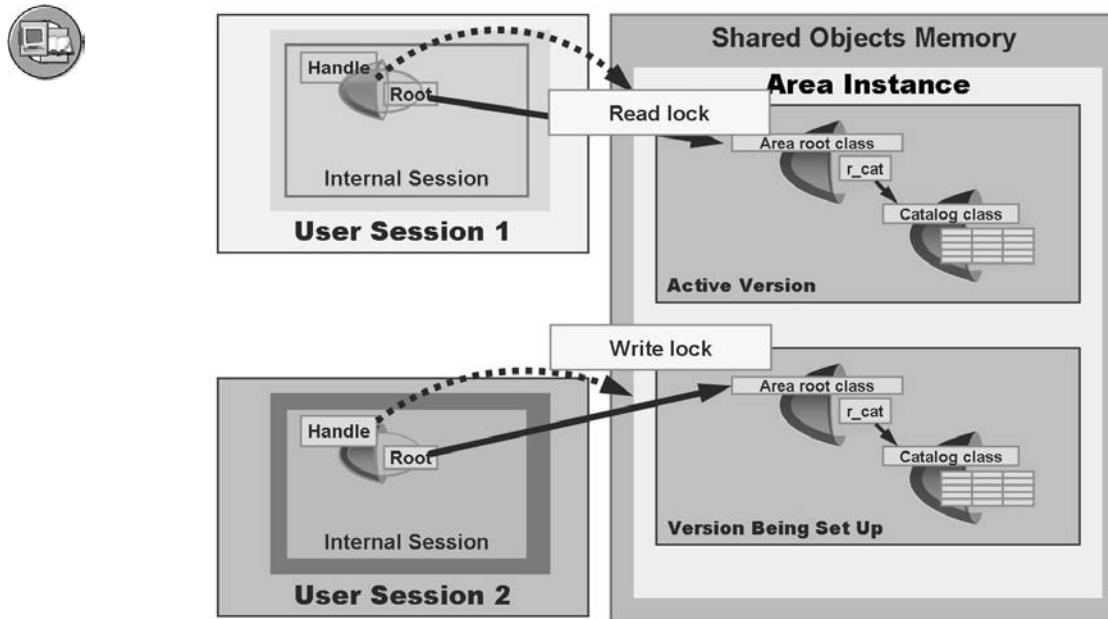


Figure 161: Version Being Set Up

If the *Number of Versions* attribute of the area is set accordingly, additional versions of the area instance can exist in addition to the active version.

When a new catalog is being set up, several temporary versions of the same area instance exist concurrently. As soon as an application sets a change lock for an area instance, a “version being set up” is created and exists in parallel with the active version.

Obsolete Version

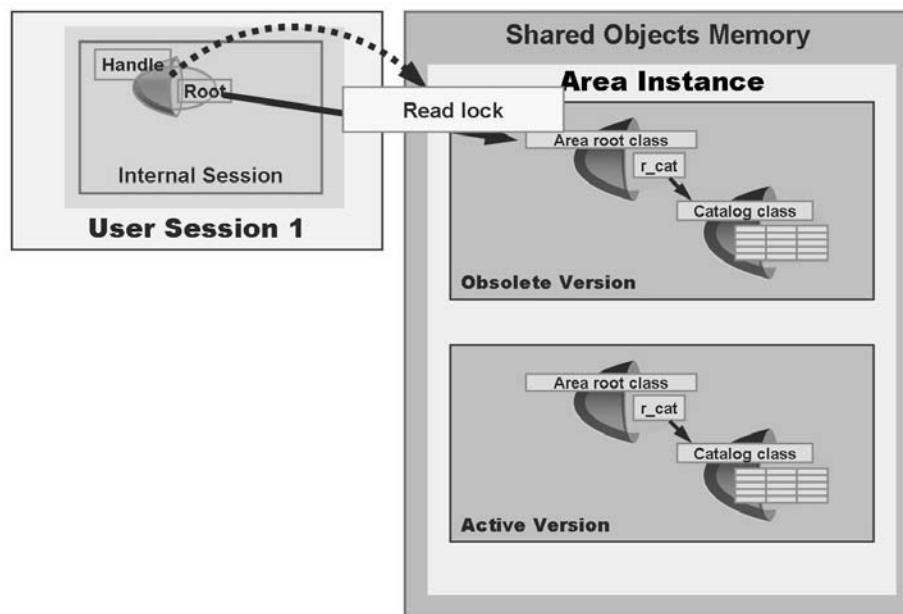


Figure 162: Writing Completed - Previous Version Is Obsolete

If the setup of the new version is completed during a read access of the currently active version, the version being set up become active. The previously active version acquires the attribute *Obsolete*.

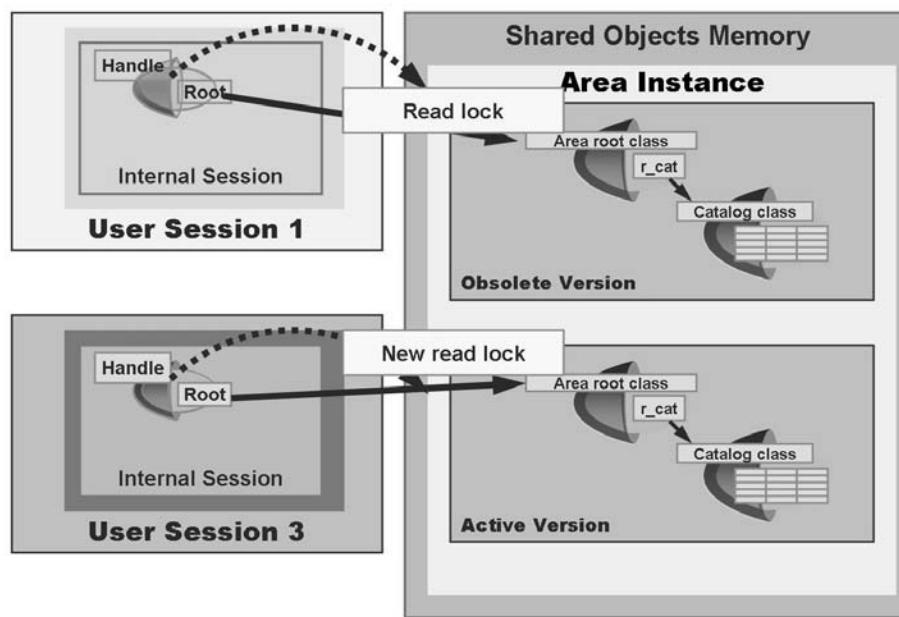


Figure 163: New Read Locks for the New Active Version

The read locks for the obsolete version remain until the read operations have been completed. New read locks for the area instance are always set for the active version, however. Two different readers may therefore get two different read results in this case.



Expired Version

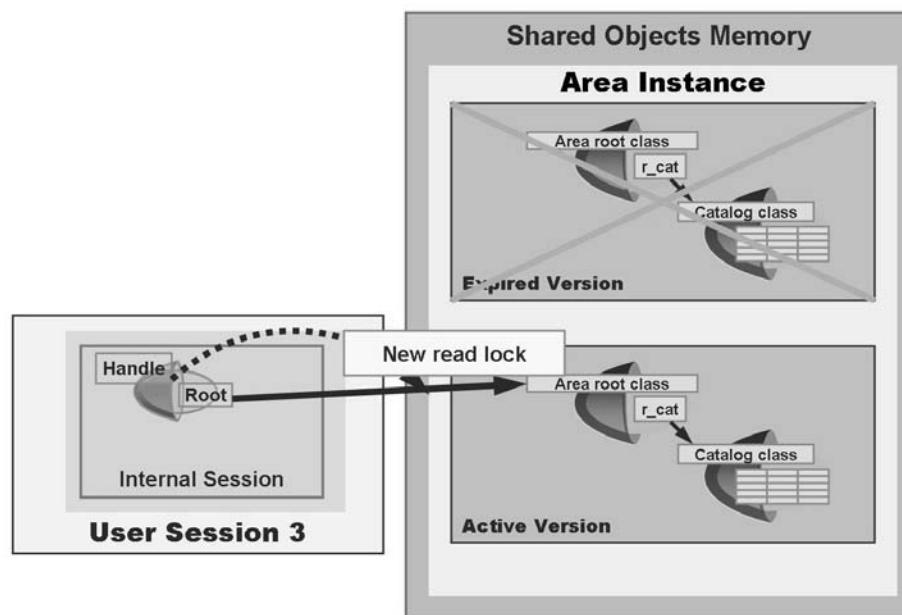


Figure 164: No Read Lock for Obsolete Version: Version Expired

When the last existing read lock for an obsolete version is removed, the version expires. It is then deleted by the garbage collector. You cannot set locks for expired versions, nor are they used to determine the number of versions (the latter is important if the number of area instance versions has been restricted).

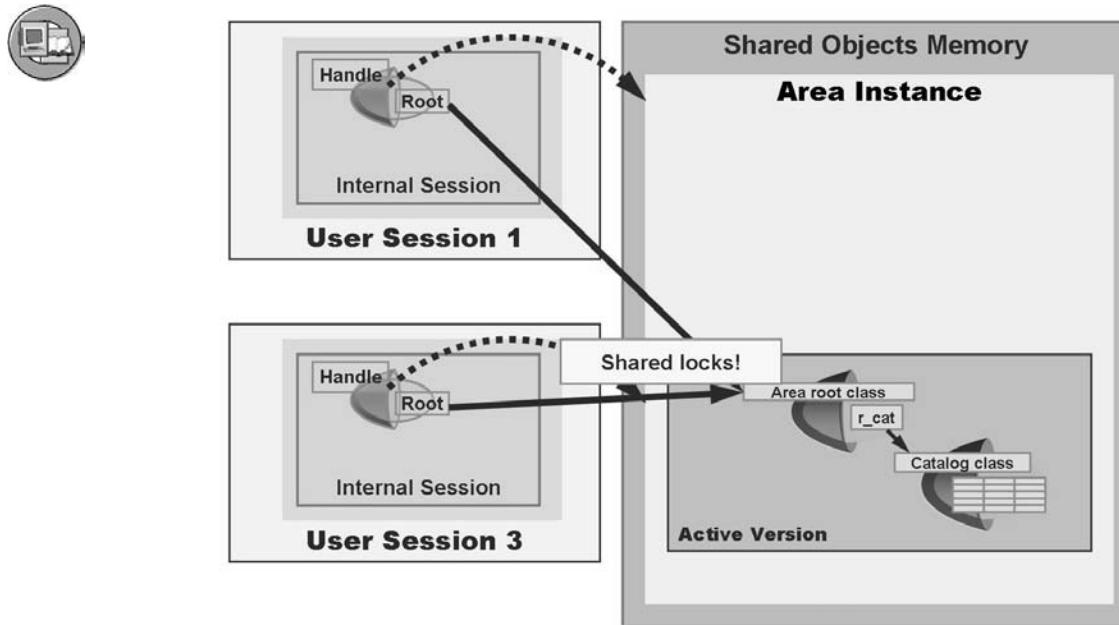


Figure 165: New Read Locks for Active Version

New read locks are always set for the active version. There can only be one active version for each area instance. Depending on the maximum number of versions, several outdated versions for which read locks are still set may exist in parallel.

Exercise 21: Shared Objects

Exercise Objectives

After completing this exercise, you will be able to:

- Create a shared objects area
- Write data to a shared objects area
- Read data from a shared objects area

Business Example

You want to run an Internet shop based on an SAP system. In this exercise, you create the foundations for this, namely saving the flights on offer. To give your Internet customers the fastest possible access to offers, you store the flight information in the shared objects memory.

You will need a “write program” to manage and describe this information, and will need to develop a test application to read the data.

Task 1:

Create root class

1. Create a global class ZCL_DNW7AW_ROOT_##. (## stands for your two-digit group number.) It should be shared memory-enabled.

The class should have a public internal table named *MT_FLIGHTS* of the type *TY_FLIGHTS* as its only (instance) component.

Activate the class.

Task 2:

Create an area in the shared objects memory

1. Create an area named ZCL_DNW7AW_AREA_## using transaction SHMA. This area should have the following attributes:

Client-specific
With version management
Displacement not possible
Root class: ZCL_DNW7AW_ROOT_##

Continued on next page

Task 3:

Create a write program for setting up an area instance in the shared objects memory

1. Create a program with the type *Executable Program*.
Name of program: **ZDNW7AW_WRITE_CATALOG_##**.
2. You need two reference variables: one for the area handle and one for the area root class.
3. Generate an area instance and an instance for the area root class. Link the area root class instance to the area instance handle.
4. Read all the data from the database table SFLIGHT to the (public) attribute *MT_FLIGHTS* of the area root class instance.
5. Do not forget to release the write lock after the data is written successfully.
6. Start your program in debugging mode and follow the step-by-step build-up of the area and the set locks in a separate window in the transaction SHMM.

Task 4:

Implement read program

1. Create a program with the type *Executable Program*.
Name of program: **ZDNW7AW_READ_CATALOG_##**.
2. You need a reference variable for the area handle.

Create a read lock on your area. The handle now gives you access to the root class object and consequently to its internal table with the flight information.

Show the contents of the internal table with the WRITE statement.

A more elegant solution would be output with ALV. To do this, you need to copy the contents of the internal table in the shared object memory to an internal program table that has been typed in exactly the same way. (The ALV requires write access to the internal table.)

Solution 21: Shared Objects

Task 1:

Create root class

1. Create a global class ZCL_DNW7AW_ROOT_##. (## stands for your two-digit group number.) It should be shared memory-enabled.

The class should have a public internal table named *MT_FLIGHTS* of the type *TY_FLIGHTS* as its only (instance) component.

Activate the class.

- a) Create the class in the *Object Navigator* (transaction SE80) in the usual way. On the *Properties* tab page, choose *Shared Memory-Enabled*.
- b) Create the internal table on the *Attributes* tab page.
- c) Press **Ctrl + F3**.

Task 2:

Create an area in the shared objects memory

1. Create an area named ZCL_DNW7AW_AREA_## using transaction SHMA. This area should have the following attributes:

Client-specific
With version management
Displacement not possible
Root class: ZCL_DNW7AW_ROOT_##

- a) Enter **ZCL_DNW7AW_AREA_##** on the initial screen for the transaction SHMA and choose *Create*. On the screen that follows, enter the necessary details. Save your entries.

Task 3:

Create a write program for setting up an area instance in the shared objects memory

1. Create a program with the type *Executable Program*.

Continued on next page

Name of program: **ZDNW7AW_WRITE_CATALOG_##**.

- a) Proceed in the usual manner.
2. You need two reference variables: one for the area handle and one for the area root class.
 - a) See model solution
3. Generate an area instance and an instance for the area root class. Link the area root class instance to the area instance handle.
 - a) See model solution
4. Read all the data from the database table SFLIGHT to the (public) attribute *MT_FLIGHTS* of the area root class instance.
 - a) See model solution
5. Do not forget to release the write lock after the data is written successfully.
 - a) See model solution
6. Start your program in debugging mode and follow the step-by-step build-up of the area and the set locks in a separate window in the transaction SHMM.
 - a) Start transaction SHMM. After you have executed the method ATTACH_FOR_WRITE and refreshed the display, you should be able to see the area ZCL_DNW7AW_AREA## and a write lock should be displayed in the transaction SHMM. After the SELECT command, the internal table of the area instance should have been filled. After you have called the method DETACH_COMMIT, there should be only one active version and no locks. Now you can also check the content of the internal table: Double-click on your area. In the screen that follows, select the (default) area instance version and then choose *Read Active Version*.

Task 4:

Implement read program

1. Create a program with the type *Executable Program*.

Name of program: **ZDNW7AW_READ_CATALOG_##**.

- a) Proceed in the usual manner.
2. You need a reference variable for the area handle.

Create a read lock on your area. The handle now gives you access to the root class object and consequently to its internal table with the flight information.

Continued on next page

Show the contents of the internal table with the WRITE statement.

A more elegant solution would be output with ALV. To do this, you need to copy the contents of the internal table in the shared object memory to an internal program table that has been typed in exactly the same way. (The ALV requires write access to the internal table.)

- a) See model solution

Result

Program to Write the Data

```

REPORT  sapdnw7aw_shared_obj_writer_s1.
DATA: go_handle  TYPE REF TO cl_dnw7aw_shared_obj_area_s1,
      go_root    TYPE REF TO cl_dnw7aw_shared_obj_root_s1.

TRY.
  go_handle = cl_dnw7aw_shared_obj_area_s1->attach_for_write( ).

  CREATE OBJECT go_root AREA HANDLE go_handle.

  go_handle->set_root( go_root ).

  SELECT * FROM sflight INTO TABLE go_root->mt_flights.

  CALL METHOD go_handle->detach_commit.

  CATCH cx_shm_exclusive_lock_active
        cx_shm_version_limit_exceeded
        cx_shm_change_lock_active
        cx_shm_parameter_error
        cx_shm_pending_lock_removed

        cx_shm_wrong_handle
        cx_shm_already_detached
        cx_shm_secondary_commit
        cx_shm_event_execution_failed
        cx_shm_completion_error.

ENDTRY.

```

Continued on next page

Program to Read the Data

```

REPORT  sapdnw7aw_shared_obj_reader_s1.

DATA: go_handle      TYPE REF TO cl_dnw7aw_shared_obj_area_s1,
      gt_sfflight   TYPE ty_flights,
      gs_sfflight   TYPE sflight,
      go_alv        TYPE REF TO cl_salv_table.

TRY.
  go_handle = cl_dnw7aw_shared_obj_area_s1=>attach_for_read( ).

LOOP AT go_handle->root->mt_flights INTO gs_sfflight.
  WRITE:
    / gs_sfflight-carrid,
    gs_sfflight-connid,
    gs_sfflight-fldate,
    gs_sfflight-seatsocc,
    gs_sfflight-seatsmax.
  ENDLOOP.

* Alternative: ALV display
* ALV requires write access to the table, hence a local copy
* is needed.
*   gt_sfflight = go_handle->root->mt_flights.
*
*   CALL METHOD cl_salv_table=>factory
*     IMPORTING
*       r_salv_table = go_alv
*     CHANGING
*       t_table      = gt_sfflight.
*
*   go_alv->display( ).

  CALL METHOD go_handle->detach.
CATCH cx_shm_inconsistent
  cx_shm_no_active_version
  cx_shm_read_lock_active
  cx_shm_exclusive_lock_active
  cx_shm_parameter_error
  cx_shm_change_lock_active
  cx_salv_msg
  cx_shm_wrong_handle

```

Continued on next page

```
cx_shm_already_detached.  
ENDTRY.
```



Lesson Summary

You should now be able to:

- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program



Unit Summary

You should now be able to:

- Explain how classes are created for shared objects
- Explain how you can use shared objects to implement applications
- Access shared objects from within an ABAP program

Unit 6

Dynamic Programming

Unit Overview

The syntax supporting dynamic programming with ABAP has been expanded continuously from release to release – starting with ABAP/4. In this unit, we will systematically introduce the entire scope.

Since *SAP R/3 4.6C*, the SAP standard classes in ABAP allow you to **query** type attributes at runtime. This concept (**RTTI** = Runtime Type Identification) has been enhanced in *SAP NetWeaver 2004* with an option to **create** dynamic objects at runtime (**RTTC** = Runtime Type Creation). Combined, these two concepts form the **RTTS** (Run Time Type Services), which is described in the second part of this unit.



Unit Objectives

After completing this unit, you will be able to:

- Access class components and object components dynamically
- Define field symbols
- Define data references
- Dereference data references
- Generate data objects dynamically
- Query type attributes at runtime
- Create types dynamically

Unit Contents

Lesson: Dynamic Programming with Field Symbols and References	362
Exercise 22: Type Casting for Structures	375
Lesson: Runtime Type Services	381
Exercise 23: RTTI – Querying the Attributes of a Data Type	391

Lesson: Dynamic Programming with Field Symbols and References

Lesson Overview

The syntax supporting dynamic programming with ABAP has been expanded continuously from release to release – starting with ABAP/4. Our intention here is **not** to trace this development; instead, we will systematically introduce the entire scope.

Nonetheless, knowing about the history may help you understand why similar concepts are sometimes implemented using widely differing syntax structures.

Since *SAP Web AS 6.10* at the latest, ABAP offers **at least** the same options that you may know from other programming languages. In contrast to these other languages, ABAP can also rely on support from the *ABAP Workbench*, even in the area of dynamic programming.



Lesson Objectives

After completing this lesson, you will be able to:

- Access class components and object components dynamically
- Define field symbols
- Define data references
- Dereference data references
- Generate data objects dynamically

Business Example

Mr. Jones is a software developer at a major corporation that develops proprietary business applications in ABAP. He is asked to develop a flexible application. Mr. Jones is looking for a programming technique in ABAP that is comparable to pointers in other programming languages. Field symbols seem to be a similar feature, so Mr. Jones uses them to develop his flexible application.

Field symbols

ABAP has featured field symbols as **dereferenced** pointers for some time now. Field symbols allow you “symbolic” access to an existing data object. All accesses that you make to the field symbol are made to the data object assigned to it. Therefore, you can access only the **content** of the data object to which the field symbol points. This technique is referred to as the “value semantics of field symbols”.



```

FIELD-SYMBOLS <fs> { { TYPE|LIKE } ... | TYPE ANY }.
ASSIGN dataobject TO <fs>.
UNASSIGN <fs>.
... <fs> IS ASSIGNED ...
  
```

Generic or complete type definition

```

DATA int TYPE i VALUE 15.
FIELD-SYMBOLS <fs_int> TYPE i.

ASSIGN int TO <fs_int>.
WRITE: / int, <fs_int>.

<fs_int> = 17.
WRITE: / int, <fs_int>.

UNASSIGN <fs_int>.
IF <fs_int> IS ASSIGNED.
  WRITE: / int, <fs_int>.
ELSE.
  WRITE: / 'fieldsymbol not assigned' (fna).
ENDIF.
  
```

<FS_INT> → 15 INT

<FS_INT> → 15 INT

<FS_INT> → 17 INT

<FS_INT> → 17 INT

Figure 166: Field symbols

You declare field symbols using the FIELD-SYMBOLS statement.



Caution: Note that the angle brackets (<>) in the field symbol name are part of the syntax.

You use the ASSIGN statement to assign a data object to the field symbol. If the field symbol is not typed (TYPE ANY), it adopts the type of the data object.

By specifying a type for the field symbol, you can ensure that only compatible objects are assigned to it.

Example:

```

DATA: date TYPE d VALUE '20040101', time TYPE t.
FIELD-SYMBOLS: <fs_date> TYPE d, <fs_time> TYPE t.
ASSIGN: date TO <fs_date>, time TO <fs_time>.

* possible?
<fs_time> = <fs_date>.
  
```

The last statement represents a syntax error. Conversely, using the following construction would trigger a runtime error:

```
FIELD-SYMBOLS: <fs_date> TYPE ANY, <fs_time> TYPE ANY.
```

You can use `IS ASSIGNED` to check whether a data object is assigned to the field symbol. The `UNASSIGN` statement sets the field symbol to “point to nothing”. Accordingly, the logical expression `IS ASSIGNED` is false after this statement.



```
ASSIGN dataobject TO <fs> CASTING [ TYPE type_name | ... ] .
```

```
TYPES: BEGIN OF st_date,
      year(4) TYPE n,
      month(2) TYPE n,
      day(2)   TYPE n,
    END OF st_date.

* option 1: implicit
FIELD-SYMBOLS <fs> TYPE st_date
ASSIGN sy-datum TO <fs> CASTING.
WRITE: / <fs>-year, <fs>-month, <fs>-day.

* option 2: explicit
FIELD-SYMBOLS: <fs> TYPE ANY,
               <fs_year> TYPE st_date>year ...
ASSIGN sy-datum TO <fs> CASTING TYPE st_date.
ASSIGN COMPONENT 1 OF STRUCTURE TO <fs_year>.
ASSIGN COMPONENT 2 ...
WRITE: / <fs_year>, ... .
```

The contents of the assigned data object are interpreted as if the type were specified implicitly or explicitly

The contents of the assigned data object are interpreted as if the type were specified implicitly or explicitly



Figure 167: Type Casting for Field Symbols

If you use the `CASTING` addition when you assign a data object to a field symbol that has a different type, you can remove the restrictions of having to use the data object's original type. The data object is then interpreted **as though** it had the data type of the field symbol.

If you use the `CASTING TYPE` addition when you assign a data object to a field symbol that has a different type, you can access the data object using the field symbol as if the object had that **explicitly specified** type.

In the above example, note that the system field `sy-datum` is an elementary character-type component of length 8.

You can also use type casting dynamically when you assign a data object to a field symbol.

Example of Dynamic Type Casting:

```
PARAMETERS tabname TYPE dd02l-tabname.
DATA: line(65535) TYPE c.

FIELD-SYMBOLS <fs_wa> TYPE ANY.
```

```
ASSIGN line TO <fs_wa> CASTING TYPE (tabname).
```

You can now use <fs_wa> to access line as if this elementary data object had the same type as the line type of the transparent table passed using tabname.

→ **Note:** This simple example will only work in Unicode systems if tabname defines a character-like structure.

Accessing Attributes with Field Symbols

Field symbols can be used to access both static attributes and instance attributes, by assigning their contents to the field symbols. You then access the attribute in the syntax instead of the data object.

Attribute Access with ASSIGN Statement

```
ASSIGN class=>static_attribute TO <fs>.  
ASSIGN o_ref->instance_attribute TO <fs>.
```

If the field symbol <fs> is typed generically, any missing type characteristics will be copied from the attributes during assignment.

This access can also be performed dynamically:

Dynamic Attribute Access with ASSIGN Statement

```
DATA:  
      classname      TYPE seoclsname VALUE 'CLASS',  
      attributename  TYPE seocpdname VALUE 'ATTRIBUTE'.  
  
      ASSIGN (classname)=>(attributename) TO <fs>.  
      ASSIGN o_ref->(attributename) TO <fs>.
```

Data References

Data references and object references were introduced alongside field symbols as part of the enhancements in *SAP R/3 4.6A*. Since this point, ABAP features full “reference semantics”.

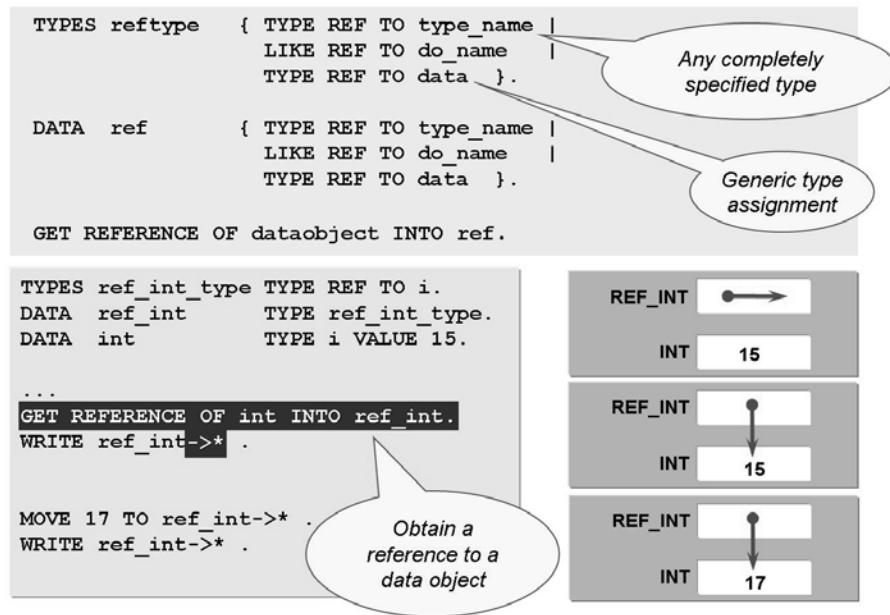


Figure 168: Data References

Data reference variables contain data references, or pointers to data objects. You can use the `TYPE REF TO` addition for the `TYPES` statement, to define a reference type for a data object. In this statement, you can either specify an explicit data type or use `TYPE REF TO data` to choose the generic variant. In this case, your data reference can point to any type of data object.

→ **Note:** The syntax item `data` represents a predefined identifier, and is comparable to `space` or `constructor`.

The corresponding `DATA` statement defines the data reference variable itself. Reference variables are data objects that can contain the address of any data object (`TYPE REF TO data`) or a data object of the specified type.

Data references involve using reference semantics, that is, when a data reference variable is accessed, the data reference itself is addressed, which means any changes affect the **addresses**.

Data reference variables are handled in ABAP like other data objects with an elementary data type. This means a reference variable can be defined not only as a single field, but also as the smallest indivisible unit of complex data objects such as structures or internal tables.

After it has been defined, a data reference variable is initial, and thus contains a blank pointer. In order for a data reference to contain a reference that points to a data object, the GET REFERENCE OF statement has to retrieve a reference to a data object that has already been defined.

It can also be assigned an existing data reference from another data reference variable, or a data object can be created dynamically with the reference. (More about this later.)

Statically typed data references can be dereferenced directly using the dereferencing operator `->*`. This means they directly access the content of the data object that points to the reference. To ensure compatibility, you have to dereference **generically** typed data references (TYPE REF TO data) and assign them to a field symbol, which you can then use to access the content. (This method will be demonstrated again later.)

Therefore, the expression `data_reference->*` is clearly comparable to the expression `<field_symbol>`.

When statically **structured** typed data references are involved, you can address the components of the referenced data object directly, using the component selector `->`, and use them in any operand item.

Example:

```
DATA: wa  TYPE sflight,
      ref TYPE REF TO sflight.

      GET REFERENCE OF wa INTO REF.

      ref->flddate = ref->flddate + 5.

      WRITE: / ref->seatsmax.
```

The component selector therefore corresponds here to the hyphen – for regular component access to structured data objects.



Validity of References - Logical Expression

... ref IS [NOT] BOUND ...

The expression `ref IS [NOT] BOUND` is used to query whether the reference variable `ref` contains a **valid** reference. `ref` must be a data or object reference variable.

When a data reference is involved, this logical expression is true if it can be dereferenced. When an object reference is involved, it is true if it points to an object. The logical expression is always false if `ref` contains a zero reference.

In comparison, the logical expression `... ref IS [NOT] INITIAL ...` merely lets you determine whether or not `ref` contains a zero reference.

Dynamic Instantiation and Cast Assignments for Data Objects

You can use data references and the CREATE DATA statement to generate data objects **at runtime** analogous to class instances:



Instantiation of Data Objects at Runtime

```
DATA ref TYPE REF TO typename.
```

```
CREATE DATA ref.
```

The `ref` data reference then points to the created data object. As an alternative to the static variant, you can also determine the data type at runtime:



Generic Typing of Data Objects at Runtime

```
DATA ref TYPE REF TO data.
```

```
CREATE DATA ref TYPE typename.
```

The `ref` reference has to be typed generically with `TYPE REF TO data`. In the following dynamic variant, the type name `typename` can also be specified using a variable `var_type`:



Dynamic Instantiation of Data Objects at Runtime

```
DATA ref TYPE REF TO data.
```

```
DATA var_type TYPE ... .
```

```
var_type = ... .
```

```
CREATE DATA ref TYPE (var_type).
```

In order to access the content of a data object that a data reference points to, the object has to be dereferenced, as described above. To ensure compatibility, however, a field symbol is required for generically typed data references (`TYPE REF TO data`):



```

ASSIGN ref->* TO <fs> [ CASTING ... ] .

DATA ref_itab TYPE REF TO data.

FIELD-SYMBOLS <fs_itab> TYPE ANY TABLE.

CREATE DATA ref_itab TYPE ... TABLES
      OF line_type
      WITH DEFAULT KEY.

ASSIGN ref_itab->* TO <fs_itab>.

SELECT * FROM tab_name INTO TABLE <fs_itab>.

```

The field symbol can be used to address the content of the internal table.

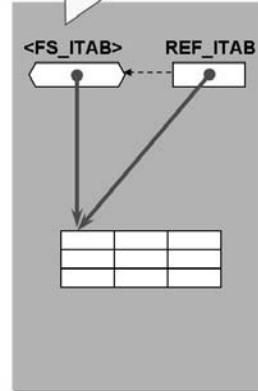


Figure 169: Dereferencing Generically Typed Data References

The ASSIGN ref_itab->* TO <fs_itab> statement assigns the data object specified in reference variable ref_itab to the field symbol <fs_itab>. If the assignment is successful, sy-subrc is set to zero.

If the field symbol is typed completely generically, it assumes the data type of the data object. If the field symbol is typed partially or fully, the system checks the compatibility of the data types. You can also cast the assigned data object.

If the data reference is initial or invalid, dereferencing is not possible. In this case, the field symbol is not changed and sy-subrc is set to four.



Generating Data Objects at Runtime – Example Application

```

REPORT ... .

DATA:
  ref_itab TYPE REF TO data,
  ref_wa   TYPE REF TO data.

FIELD-SYMBOLS:
  <fs_itab> TYPE ANY TABLE,
  <fs_wa>    TYPE ANY,
  <fs_comp>  TYPE ANY.

```

```

PARAMETERS pa_tab TYPE dd02l-tablename DEFAULT 'SPFLI'.

START-OF-SELECTION.
  CREATE DATA ref_itab TYPE STANDARD TABLE OF (pa_tab)
    WITH NON-UNIQUE DEFAULT KEY.
  ASSIGN ref_itab->* TO <fs_itab>.

  SELECT * FROM (pa_tab)
    INTO TABLE <fs_itab>
    UP TO 100 ROWS.

  CREATE DATA ref_wa LIKE LINE OF <fs_itab>. "or: TYPE (pa_tab).
  ASSIGN ref_wa->* TO <fs_wa>.

  LOOP AT <fs_itab> INTO <fs_wa>.
    DO.
      ASSIGN COMPONENT sy-index OF STRUCTURE <fs_wa> TO <fs_comp>.
      IF sy-subrc NE 0.
        NEW-LINE.
        EXIT.
      ENDIF.
      WRITE <fs_comp>.
    ENDDO.
  ENDLOOP.

```

The example can display the content of any transparent table. You can make the FROM clause of the SELECT statement dynamic.

For the INTO clause, you will need a data object that has a line type compatible with that of the table being displayed. Since the name – and therefore the line type – of the table is not known until runtime, you will not create the data object until then. Unlike conventional data objects, you can specify the type of a data object created at runtime dynamically. The TYPE addition of the CREATE DATA statement contains the name of the table, to ensure that the system always creates the “appropriate” structure.

The statement ASSIGN ref_wa->* TO <fs_wa> assigns the data object to the field symbol. The data type of the table is inherited by the field symbol so type casting is no longer necessary. You can now write each data record from the SELECT statement directly to the compatibly-typed data object using the <fs_wa> field symbol.

If you knew the component names, you could display the fields directly using WRITE <fs_wa>-.... In most cases, however, you will not normally know the names of the components, or how many of them there are. As a result, you have to use

the ASSIGN-COMPONENT variant for output: Each component in the structure <fs_wa> is assigned consecutively to the field symbol <fs_comp> and then output. When the loop runs out of components, the program requests the next data record.



```
MOVE ref1 ?TO ref2.
```

```
DATA:  
    int  TYPE i VALUE 15,  
    date TYPE d VALUE '20040101'.  
  
DATA:  
    ref_int  TYPE REF TO i,  
    ref_date  TYPE REF TO d,  
    ref_gen  TYPE REF TO data.  
...  
GET REFERENCE OF int INTO ref_int.  
* narrowing cast:  
ref_gen = ref_int.  
* widening cast:  
ref_int ?= ref_gen.  
  
* but:  
GET REFERENCE OF date INTO ref_date.  
* narrowing cast:  
ref_gen = ref_date.  
* widening cast:  
ref_int ?= ref_gen.
```

Fully specified type

Generic type

No type conflict

Type conflict.
Runtime error will occur if not caught.

Figure 170: Cast Assignment of Data References

Analogous to object references, you can also use the CAST operator to copy data references to reference variables whose static type is identical to the dynamic type of the source reference. This is the case in the first example above, but not in the second.

→ **Note:** If the cast assignment operator ?= were not used, these assignments would not even pass a **syntax check**, because the source and target types are incompatible.

If a data reference variable is typed statically, it passes on its type attributes when it is assigned to a **non-typed** data reference. This attribute can be very useful, especially where the additional information of global data types is involved.

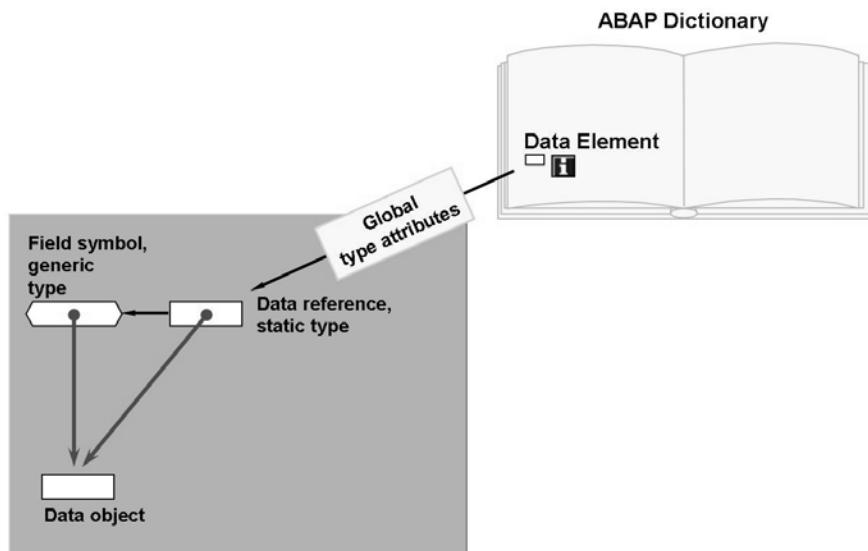


Figure 171: Transfer of Global Type Information for References - Theory

During dereferencing, the type of the reference variable, and not the type of the data object to which the reference “points”, is decisive. This is illustrated in the following example:



```

DATA:
  cityfrom    TYPE spfli-cityfrom,
  ref_cityto  TYPE REF TO spfli-cityto.
  .
  .
  .
FIELD-SYMBOLS <fs_gen> TYPE ANY.
...
GET REFERENCE OF cityfrom INTO ref_cityto.
ASSIGN ref_cityto->* TO <fs_gen>.
WRITE: <fs_gen>.

```

Annotations for the code:

- 'Departure city' is associated with 'cityfrom'.
- 'Arrival city' is associated with 'ref_cityto'.
- 'Different global types.' is associated with the assignment statement.
- 'Static type of reference variable is used' is associated with the assignment statement.

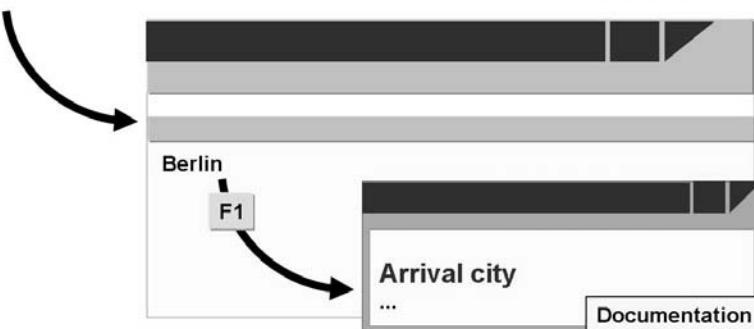


Figure 172: Transfer of Global Type Information for References – Example of Syntax

In the above example, the reference variable REF_CITYTO is typed statically with the data element. At runtime, it receives a reference to the data object CITYFROM, which has a **different** type. The reference is then dereferenced with the generically typed field symbol <FS_GEN>, and thus assumes the type attributes of the reference. As a result, the field documentation for the data element would be displayed in a list output, for example.



Hint: To achieve a consistent result, the REF_CITYTO reference would have to be typed generically or typed in agreement with the data object CITYFROM.



Caution: This example has been designed specifically to show that the type of a statically typed reference variable really is decisive.

Exercise 22: Type Casting for Structures

Exercise Objectives

After completing this exercise, you will be able to:

- Use field symbols for type casting
- Develop a simple program to display table contents

Business Example

You are asked to develop a simple program to display table contents. The user should be able to enter the name of a table in the selection screen. The program should read the data from this table and output it in a conventional list.

Task:

Develop a simple program that outputs the content of a given table in a list. Make it possible for the user to enter the table name in a selection screen.

Generate a structured data object with the type of the entered name. You want to read the data into this structure from the specified database table. Use field symbols.

Note that you will need another field symbol to output the data in the list according to the ABAP type.

1. Create a program called **ZBC402##_CREATE_DATA**.

Alternatively, you can copy the template program
SAPBC402_DYNT_CREATE_DATA.

2. At runtime, create a structured data object whose line type is compatible with the table entered by the user (suggested name: **GR_STRUC**).
3. Assign a suitable typed field symbol to the generated data object, to enable you to access its contents.
4. Program the data selection. Read the data from the specified transparent table into a the structure in a **SELECT** loop.

Continued on next page

Within this SELECT loop, output the data in a conventional list. To do so, you will need a second field symbol, to which you consecutively assign the components in the structure.



Hint: For simplicity's sake, use the UP TO . . . ROWS addition, to keep your program's runtime requirements within reasonable limits. You can always find a more elegant solution to restrict the data quantity at a later stage.

Solution 22: Type Casting for Structures

Task:

Develop a simple program that outputs the content of a given table in a list. Make it possible for the user to enter the table name in a selection screen.

Generate a structured data object with the type of the entered name. You want to read the data into this structure from the specified database table. Use field symbols.

Note that you will need another field symbol to output the data in the list according to the ABAP type.

1. Create a program called **ZBC402_##_CREATE_DATA**.

Alternatively, you can copy the template program
SAPBC402_DYNT_CREATE_DATA.

- a) Create the program in your package. Set the attributes such that the program can be executed directly.

Model solution: **SAPBC402_DYNS_CREATE_DATA**

- a) See source text excerpt.
2. At runtime, create a structured data object whose line type is compatible with the table entered by the user (suggested name: **GR_STRUC**).
 - a) See source text excerpt.
3. Assign a suitable typed field symbol to the generated data object, to enable you to access its contents.
 - a) See source text excerpt.
4. Program the data selection. Read the data from the specified transparent table into a the structure in a SELECT loop.

Within this SELECT loop, output the data in a conventional list. To do so, you will need a second field symbol, to which you consecutively assign the components in the structure.



Hint: For simplicity's sake, use the UP TO ... ROWS addition, to keep your program's runtime requirements within reasonable limits. You can always find a more elegant solution to restrict the data quantity at a later stage.

- a) See source text excerpt.

Continued on next page

Result

Source code excerpt:

```

*&-----
*& Report  SAPBC402_DYNS_CREATE_DATA
*&
*&-----
*& This program works similar to the Data Browser.
*& Features:
*&     Selection-Screen for a table name
*&     Selects Data from the table provided
*&     Output as classical list
*&-----

REPORT  sapbc402_dybs_create_data.

*-----
DATA:
    ok_code    LIKE sy-ucomm,
    ref_struct TYPE REF TO data.

*-----
FIELD-SYMBOLS:
    <fs_struct>  TYPE ANY,
    <fs_comp>   TYPE ANY.

*-----
PARAMETERS:
    pa_tab  TYPE dd02l-tabname DEFAULT 'SPFLI'.

*-----
START-OF-SELECTION.
*-----


CREATE DATA ref_struct TYPE (pa_tab).
ASSIGN ref_struct->* TO <fs_struct>.

SELECT * FROM (pa_tab)
    INTO <fs_struct>
    UP TO 100 ROWS

```

Continued on next page

```
DO.  
  ASSIGN COMPONENT sy-index OF STRUCTURE <fs_struct> to <fs_comp>.  
  IF sy-subrc <> 0.  
    NEW-LINE.  
    EXIT.  
  ENDIF.  
  WRITE: <fs_comp>.  
  ENDDO.  
  
ENDSELECT.  
  
IF sy-subrc <> 0.  
  MESSAGE e041(bc402).  
ENDIF.
```



Lesson Summary

You should now be able to:

- Access class components and object components dynamically
- Define field symbols
- Define data references
- Dereference data references
- Generate data objects dynamically

Lesson: Runtime Type Services

Lesson Overview

Since *SAP R/3 4.6C*, the SAP standard classes in ABAP allow you to **query** type attributes at runtime. This concept (**RTTI** = Runtime Type Identification) has been enhanced in *SAP NetWeaver 2004* with an option to **create** dynamic objects at runtime (**RTTC** = Runtime Type Creation). Combined, these two concepts form the **RTTS** (Runtime Type Services).



Lesson Objectives

After completing this lesson, you will be able to:

- Query type attributes at runtime
- Create types dynamically

Business Example

Mr. Jones is a software developer at a major corporation that develops proprietary business applications in ABAP. He is asked to develop a new, flexible application. Mr. Jones knows that ABAP features several programming options. One new feature allows the creation of data types at runtime. Mr. Jones will use these new options and develop a flexible application.

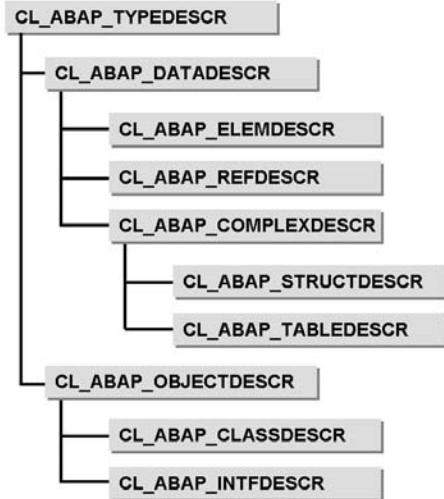
Runtime Type Identification (RTTI)

The implementation of ABAP Objects introduced a class-based concept called Runtime Type Identification (RTTI), which you can use to determine type attributes at runtime. This concept includes all ABAP types and therefore covers all functions of the now **obsolete** statements `DESCRIBE FIELD` and `DESCRIBE TABLE`.

There is a description class for each type with special attributes for special type attributes:



Hierarchy of description classes:



Determination of type properties of a (data) object at runtime:

- Transfer of data object or object reference to the static method `DESCRIBE_BY_DATA` or `DESCRIBE_BY_OBJECT_REF` of the class `CL_ABAP_TYPEDESCR`
- Acceptance of reference to the corresponding description object
- Evaluation of type attributes from the public attributes and/or methods of this description object

Figure 173:

The class hierarchy of the description classes corresponds to the hierarchy of the types in ABAP. In addition, the description classes for complex types, references, classes, and interfaces have special methods that are used to specify references to subtypes. You can use these methods to navigate through a compound type to all its subtypes.

To obtain a reference to a description object of a type, you must use the static methods of the class `CL_ABAP_TYPEDESCR` or the navigation methods of the special description class. The description objects are then created from one of the subclasses. At runtime, only one description object exists for each type. The attributes of the description object contain information on the attributes of the type.

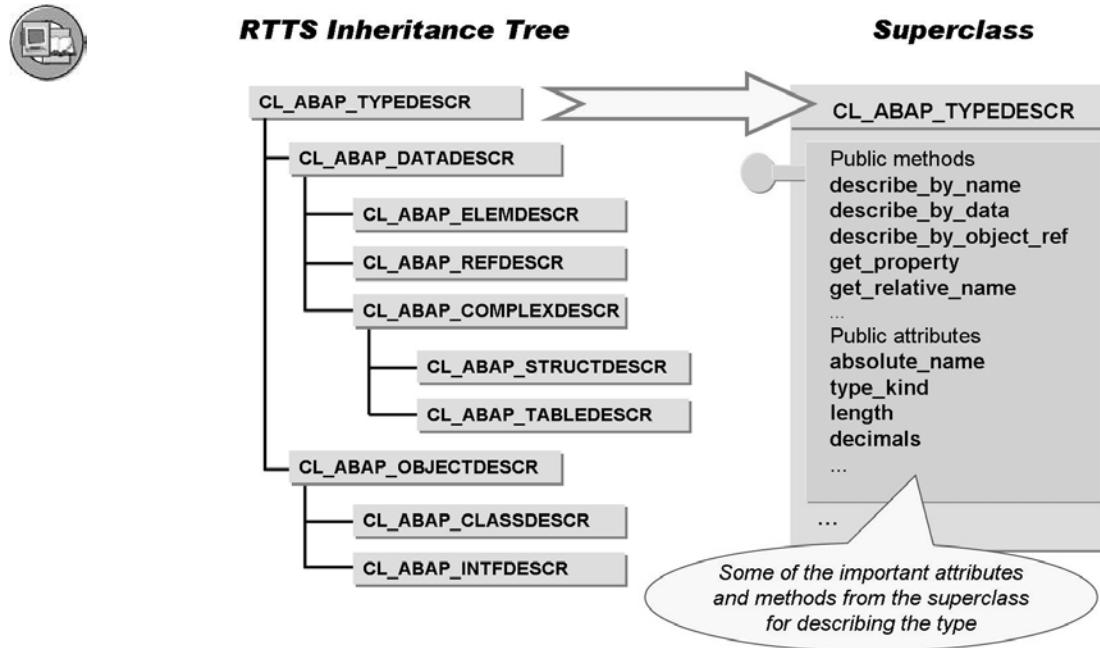
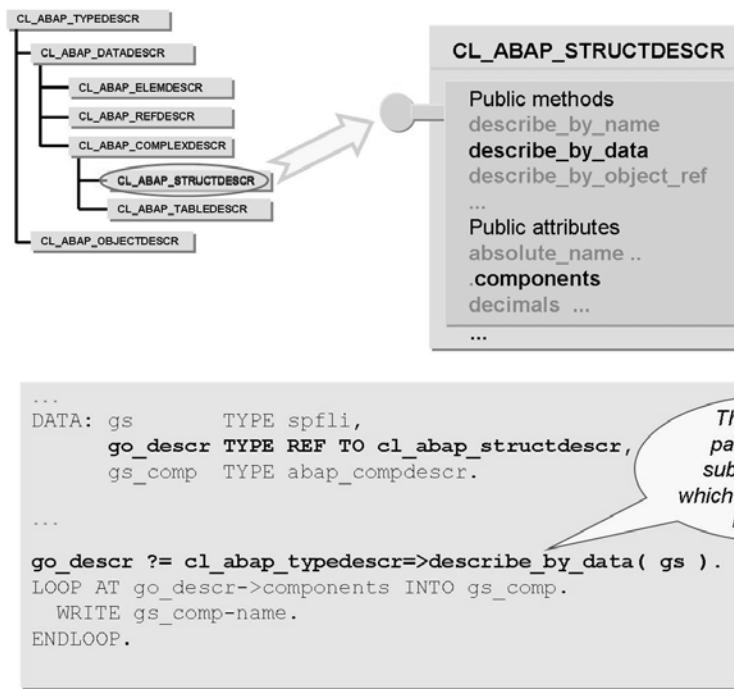


Figure 174:

Dynamic Data Type Analysis

In the following application example, the attributes of a structure will be identified using RTTI. To do so, we will use subclass *CL_ABAP_STRUCTDESCR*.

**Figure 175:**

Since we need the attributes of a structure, we first define a reference to the appropriate description class. Instances of this class possess a *COMPONENTS* attribute, which you use to describe the individual components of the relevant structure. This attribute is an internal table. You therefore also need to define a work area with a compatible line type.

The functional method call returns the reference to the description instance of the transferred structure. (The *gs* structure could also have been generated dynamically; it is static, as usual, in this example.)

Only the abstract class *CL_ABAP_TYPEDESCR* contains the method *DESCRIBE_BY_DATA*. Its returning parameter is typed as a reference to this superclass. Since the current parameter *REF_DESCR* is typed to the subclass *CL_ABAP_STRUCTDESCR*, however, a downcast is required for the assignment.

You can then access the attributes of the description instance in any manner. In this example, the component names are displayed as headers. (For the sake of clarity, we have omitted the formatting options.)

Dynamic Object Type Analysis

RTTI also permits you to analyze the attributes of objects. Class CL_ABAP_OBJECTDESCR and its two subclasses, CL_ABAP_CLASSDESCR and CL_ABAP_INTFDESCR, are available for this purpose.

In the example below, you want to analyze the attributes of an object. The reference variable SENDER is transferred to the method DESCRIBE_BY_OBJ_REF.

 **Note:** To obtain information about a class definition, call the method DESCRIBE_BY_NAME and transfer the class name.

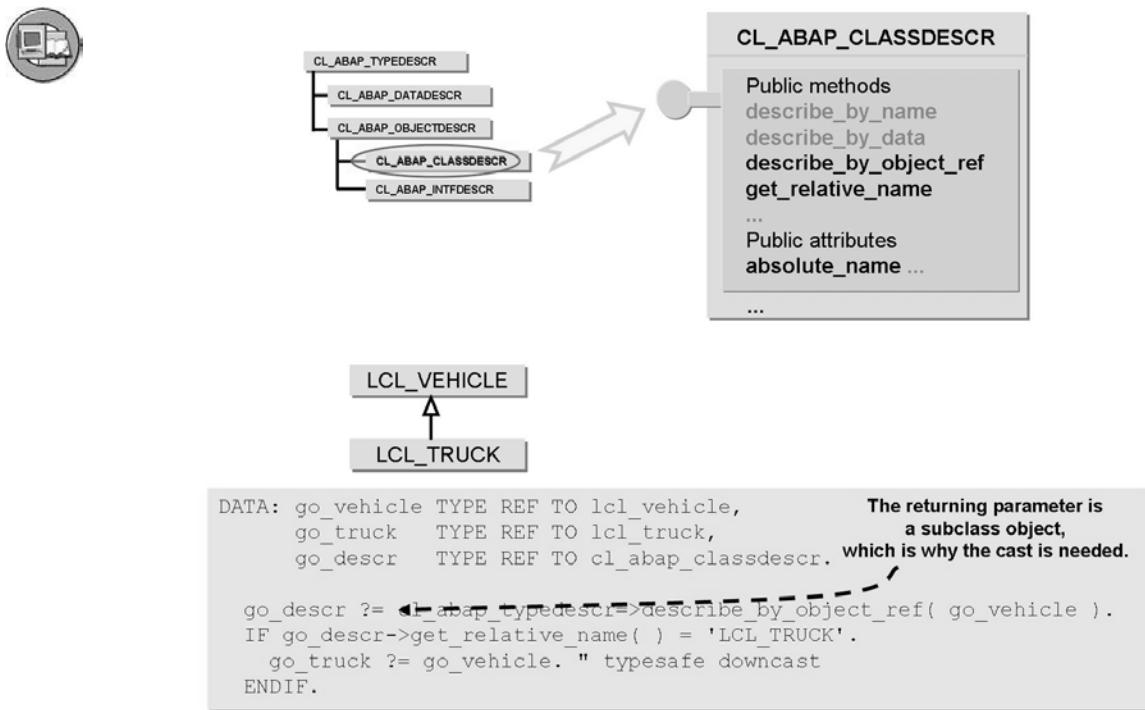


Figure 176:

After the method call, reference variable *go_descr* contains a reference to a description object, which in turn contains information about the object from *go_vehicle*. Among other information, the description object indicates the (object) type of *go_vehicle*. If the object *go_vehicle* is typed to the class LCL_TRUCK, you can safely perform a cast using `?=`.



Hint: Only the abstract class CL_ABAP_TYPEDESCR has the method DESCRIBE_BY_OBJECT_REF. Its returning parameter is typed as a reference to this superclass. Since the current parameter *go_descr* is typed to the subclass CL_ABAP_CLASSDESCR, however, a downcast is required for the assignment.

Runtime Type Creation (RTTC)

In *SAP NetWeaver 2004*, the RTTI type classes were enhanced with RTTC, to make it possible to create types at runtime. Whereas the CREATE DATA statement creates a type implicitly, RTTC also allows you to create new types explicitly at program runtime.

In the process, the attributes of the types are implemented through attributes of type objects. This means that each type has a type object, whose attributes describe the properties of the type.

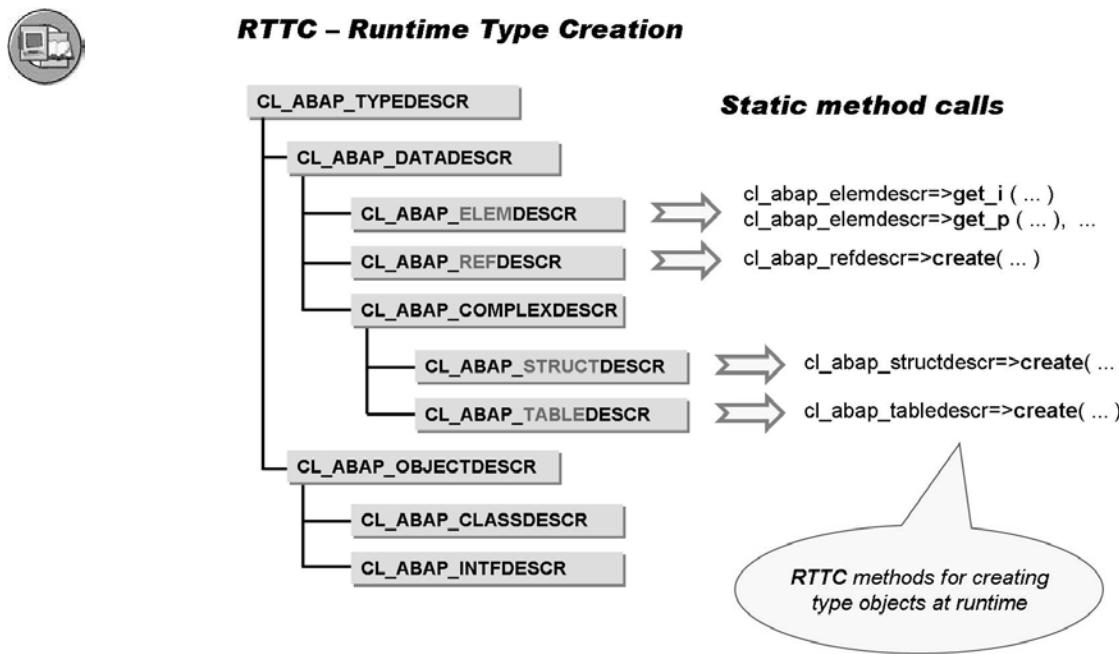
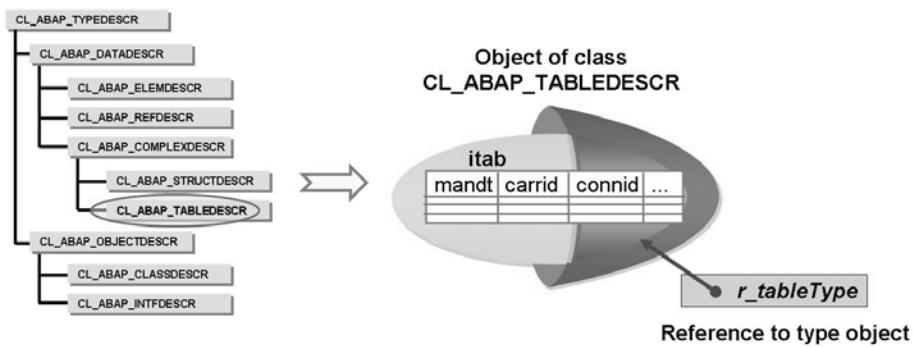


Figure 177:

The class hierarchy of the type classes corresponds to the hierarchy of the types in the ABAP type system. Type objects can be created by type class methods. You can use the static methods of the class CL_ABAP_TYPEDESCR or call the methods of the specific type classes (GET_I, GET_C, ..., CREATE) to get references to type objects.

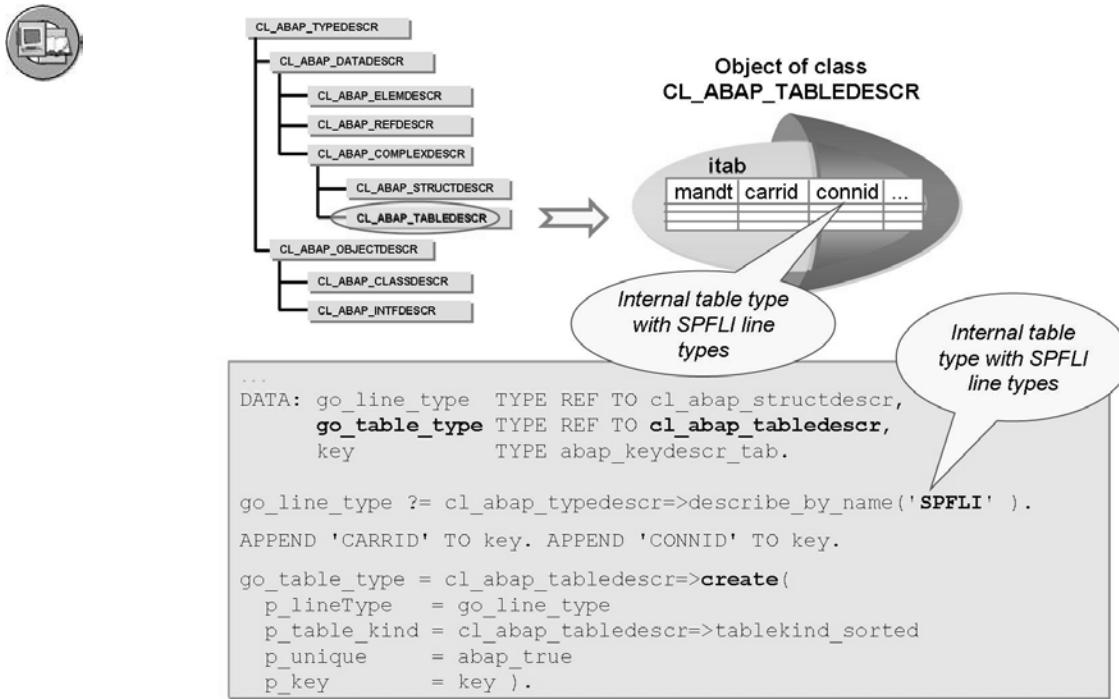


A type is fully defined by its type object.

- Each type has a runtime type object (RTTS instance).
- The runtime type object fully describes the data type.
- A type object is local to the program, transient, and anonymous (nameless).
- A type object cannot be deleted or changed.

Figure 178:

RTTC makes it possible to create both elementary and complex data types (such as structures and internal tables) at runtime. This can be demonstrated using an internal table.

**Figure 179:**

Using the **CREATE** method, the class **CL_ABAP_TABLEDESCR** enables you to create a table type at runtime. An important formal parameter of this static method is **P_LINE_TYPE**, which describes the line type through a reference to the class **CL_ABAP_TYPEDESCR**. In the example provided here, **SPFLI** is used as the line type. The method **DESCRIBE_BY_NAME** instantiates a suitable object for it.

The question now is, what can the program do with the table type you just created? For instance, you could use it as an example of creating an internal table at runtime:



Generating an Internal Table with a Dynamically Created Type

```

...DATA r_itab TYPE REF TO data,
      r_tableType TYPE REF TO cl_abap_tabledescr.
...
* Creation of internal tabletype with static method
* CREATE of RTTS-class cl_abap_tabledescr
...
CREATE DATA r_itab TYPE HANDLE r_tabletype.
```

Using the HANDLE addition, the CREATE DATA statement in the example creates an internal table, whose type is described by an RTTS type object. A reference variable from the static type of the CL_ABAP_TABLEDESCR class that refers to a type object is specified for the HANDLE. This type object may have been created by applying the RTTS methods to existing data objects or through dynamic definition of a new data type.

Exercise 23: RTTI – Querying the Attributes of a Data Type

Exercise Objectives

After completing this exercise, you will be able to:

- Use RTTI to query the attributes of a data type at runtime

Business Example

You would like to use the options provided by Runtime Type Identification for flexible programming.

Task 1:

Create a program and call a function module

1. Create a new executable program (report) and call it ZDNW7AW_RTTI_##. (## stands for your two-digit group number.)
2. Call the function module DNW7AW_RETURN_RANDOM_DATA_REF. This function module returns references to anonymous data objects of random types.

In principle, the following types are possible for anonymous data objects with this function module:

Elementary data types (for example, numeric fields or character strings), if the import parameter IV_SIMPLE_TYPES = 'X'.

Flat structure types, if the import parameter IV_STRUCTURED_TYPES = 'X'.

Internal table types, if the import parameter IV_TABLE_TYPES = 'X'.

In this task, maintain the named parameters in such a way that a reference to an elementary data object is definitely returned.

3. Create a suitable typed reference variable named *gr_data* and transfer it to the function module.

Continued on next page

Task 2:

Examine the data reference in the case of an elementary data object

1. Call the static method DESCRIBE_BY_DATA_REF of the class CL_ABAP_TYPEDESCR for your data reference *gr_data*. Save the result in an object reference named *go_type*, which you create with reference to the class CL_ABAP_TYPEDESCR.
2. Use a suitable method of the object reference *go_type* to find the relative name of the type and output it with the WRITE command.
3. Compare the attribute *type_kind* of the object reference *go_type* with suitable constants of the class CL_ABAP_TYPEDESCR, to determine whether the anonymous data object is a character string, a date field or a packed number. (You can ignore the other elementary data type options.) Output the result with the WRITE command.
4. To access the special attributes of the elementary data type, you need a variable that you type to the class CL_ABAP_ELEMDESCR. Create this variable and name it *go_elem_type*. Program a downcast from the superclass reference *go_type* to the subclass reference *go_elem_type*. Then evaluate the *output_length* attribute and output the result with the WRITE command.
5. Activate and execute your program.

Task 3:

Optional: Examine the data reference in the case of a structured data object.

1. Now change the function module call DNW7AW_RETURN_RANDOM_DATA_REF in such a way that the anonymous data object created by the random number generator can be both elementary and structured. Set the IV_SIMPLE_TYPES and IV_STRUCTURED_TYPES parameters to 'X'. Consequently, in the evaluation you have to query whether a reference to an elementary data object or a reference to a structure was returned. To do this, compare the *kind* attribute of the *go_type* variable within a CASE statement with relevant constants of the CL_ABAP_TYPEDESCR class.

If the data reference *gr_data* points to an elementary data object, this fact should be output. In addition, the code for the previous statement should be executed.

Continued on next page

If, on the other hand, the data reference points to a structure, this should be output with the WRITE command as well as the number of fields in the structure. Look for a suitable method in the class CL_ABAP_STRUCTDESCR that will return the field names (and thus indirectly also their number). Consider using a downcast again to call this method.

2. Activate and execute your program.

Task 4:

Optional: Examine the data reference if the data object is an internal table type.

1. Now change the function module call DNW7AW_RETURN_RAN-DOM_DATA_REF in such a way that the anonymous data object created by the random generator can be elementary, structured, or an internal table. To do this, set the IV_SIMPLE_TYPES, IV_STRUCTURED_TYPES and IV_TABLE_TYPES parameters to 'X'. As a result of this, you have to extend the CASE evaluation. To do this, compare the *kind* attribute of the *go_type* variable within a CASE statement with relevant constants of the CL_ABAP_TYPEDESCR class.

If the object actually is an internal table, this fact should be output with the WRITE command, together with the table type. Search for a suitable attribute in the class CL_ABAP_TABLEDESCR. Consider using a downcast again to access this attribute.

2. Activate and execute your program.

Solution 23: RTTI – Querying the Attributes of a Data Type

Task 1:

Create a program and call a function module

1. Create a new executable program (report) and call it ZDNW7AW_RTTI_##. (## stands for your two-digit group number.)
 - a) Proceed in the usual manner.
2. Call the function module DNW7AW_RETURN_RANDOM_DATA_REF. This function module returns references to anonymous data objects of random types.
In principle, the following types are possible for anonymous data objects with this function module:
 - Elementary data types (for example, numeric fields or character strings), if the import parameter IV_SIMPLE_TYPES = 'X'.
 - Flat structure types, if the import parameter IV_STRUCTURED_TYPES = 'X'.
 - Internal table types, if the import parameter IV_TABLE_TYPES = 'X'.In this task, maintain the named parameters in such a way that a reference to an elementary data object is definitely returned.
 - a) See model solution
3. Create a suitable typed reference variable named *gr_data* and transfer it to the function module.
 - a) See model solution

Task 2:

Examine the data reference in the case of an elementary data object

1. Call the static method DESCRIBE_BY_DATA_REF of the class CL_ABAP_TYPEDESCR for your data reference *gr_data*. Save the result in an object reference named *go_type*, which you create with reference to the class CL_ABAP_TYPEDESCR.
 - a) See model solution

Continued on next page

2. Use a suitable method of the object reference *go_type* to find the relative name of the type and output it with the WRITE command.
 - a) See model solution
3. Compare the attribute *type_kind* of the object reference *go_type* with suitable constants of the class CL_ABAP_TYPEDESCR, to determine whether the anonymous data object is a character string, a date field or a packed number. (You can ignore the other elementary data type options.) Output the result with the WRITE command.
 - a) See model solution
4. To access the special attributes of the elementary data type, you need a variable that you type to the class CL_ABAP_ELEMDESCR. Create this variable and name it *go_elem_type*. Program a downcast from the superclass reference *go_type* to the subclass reference *go_elem_type*. Then evaluate the *output_length* attribute and output the result with the WRITE command.
5. Activate and execute your program.
 - a) Select **Ctrl + F3** and **F8**.

Task 3:

Optional: Examine the data reference in the case of a structured data object.

1. Now change the function module call DNW7AW_RETURN_RANDOM_DATA_REF in such a way that the anonymous data object created by the random number generator can be both elementary and structured. Set the IV_SIMPLE_TYPES and IV_STRUCTURED_TYPES parameters to 'X'. Consequently, in the evaluation you have to query whether a reference to an elementary data object or a reference to a structure was returned. To do this, compare the *kind* attribute of the *go_type* variable within a CASE statement with relevant constants of the CL_ABAP_TYPEDESCR class.

If the data reference *gr_data* points to an elementary data object, this fact should be output. In addition, the code for the previous statement should be executed.

If, on the other hand, the data reference points to a structure, this should be output with the WRITE command as well as the number of fields in the structure. Look for a suitable method in the class CL_ABAP_STRUCTDESCR that will return the field names (and thus indirectly also their number). Consider using a downcast again to call this method.

- a) See model solution

Continued on next page

2. Activate and execute your program.
- a) Select **Ctrl + F3** and **F8**.

Task 4:

Optional: Examine the data reference if the data object is an internal table type.

1. Now change the function module call DNW7AW_RETURN_RANDOM_DATA_REF in such a way that the anonymous data object created by the random generator can be elementary, structured, or an internal table. To do this, set the IV_SIMPLE_TYPES, IV_STRUCTURED_TYPES and IV_TABLE_TYPES parameters to 'X'. As a result of this, you have to extend the CASE evaluation. To do this, compare the *kind* attribute of the *go_type* variable within a CASE statement with relevant constants of the CL_ABAP_TYPEDESCR class.

If the object actually is an internal table, this fact should be output with the WRITE command, together with the table type. Search for a suitable attribute in the class CL_ABAP_TABLEDESCR. Consider using a downcast again to access this attribute.

2. Activate and execute your program.
- a) Select **Ctrl + F3** and **F8**.

Result

SAPDNW7AW_RTTI_S1

```

REPORT  sapdnw7aw_rtti_s1.
DATA:
  gr_data          TYPE REF TO data,
  go_type          TYPE REF TO cl_abap_typedescr,
  go_elem_type     TYPE REF TO cl_abap_elemdescr,
  go_struc_type   TYPE REF TO cl_abap_structdescr,
  go_table_type    TYPE REF TO cl_abap_tabledescr,
  gt_components    TYPE abap_component_tab,
  gv_name          TYPE string,
  gv_output_length TYPE i,
  gv_lines         TYPE sy-tabix.

TRY.
  CALL FUNCTION 'DNW7AW_RETURN_RANDOM_DATA_REF'

```

Continued on next page

```

EXPORTING
    iv_simple_types      = 'X'
    iv_structured_types = 'X'
    iv_table_types       = 'X'
IMPORTING
    pr_data              = gr_data.
IF gr_data IS NOT INITIAL.
    CALL METHOD cl_abap_typedescr->describe_by_data_ref
        EXPORTING
            p_data_ref   = gr_data
        RECEIVING
            p_descr_ref = go_type.

    gv_name = go_type->get_relative_name( ).
    WRITE: / gv_name.

CASE go_type->kind.
    WHEN cl_abap_typedescr->kind_elem.
        WRITE: / 'It is an elementary type'(ele).
        go_elem_type ?= go_type.
        gv_output_length = go_elem_type->output_length.
        WRITE: / 'The output length:'(out), gv_output_length.
    CASE go_type->type_kind.
        WHEN cl_abap_typedescr->typekind_char.
            WRITE: / 'It is a character type'(cha).
        WHEN cl_abap_typedescr->typekind_packed.
            WRITE: / 'It is a packed type'(pac).
        WHEN cl_abap_typedescr->typekind_date.
            WRITE: / 'It is a date type'(dat).
    ENDCASE.
    WHEN cl_abap_typedescr->kind_struct.
        WRITE: / 'It is a structured type'(str).
        go_struct_type ?= go_type.
        gt_components = go_struct_type->get_components( ).
        gv_lines = LINES( gt_components ).
        WRITE: / 'The number of fields:'(fie), gv_lines.

    WHEN cl_abap_typedescr->kind_table.
        WRITE: / 'It is a table type'(tab).
        go_table_type ?= go_type.
    CASE go_table_type->table_kind.
        WHEN cl_abap_tabledescr->tablekind_hashed.

```

Continued on next page

```
        WRITE: / 'It is a hashed table'(has).
        WHEN cl_abap_tabledescr=>tablekind_sorted.
          WRITE: / 'It is a sorted table'(sor).
          WHEN cl_abap_tabledescr=>tablekind_std.
            WRITE: / 'It is a standard table'(std).
          ENDCASE.
        ENDCASE.
      ENDIF.
    CATCH cx_dnw7aw_no_data.
  ENDTRY.
```



Lesson Summary

You should now be able to:

- Query type attributes at runtime
- Create types dynamically



Unit Summary

You should now be able to:

- Access class components and object components dynamically
- Define field symbols
- Define data references
- Dereference data references
- Generate data objects dynamically
- Query type attributes at runtime
- Create types dynamically



Test Your Knowledge

1. Which of the following statements regarding dynamic programming is correct?

Choose the correct answer(s).

- A Field symbols contain the addresses of data objects at runtime.
- B When you change the content of a field symbol, this changes the address of the assigned data object.
- C When you change the content of a field symbol, this changes an address. That is, the field symbol then no longer refers to the data object.
- D You can use field symbols to access data objects as if they had a different type than their static type.
- E There are no data references in ABAP; field symbols are merely dereferenced pointers.
- F When you change the content of an assigned data reference, this changes an address. That is, the reference no longer points to the data object.
- G In ABAP, you can define the type of an object when you define the reference, or wait until it is generated at runtime.
- H In ABAP, you can define the type of a data object when you define it, or wait until it is generated at runtime.
- I A globally typed data reference points to a data object that has already been assigned a different global type. In addition, a generically typed field symbol was used to dereference the data reference. If this field reference is used to access the data object, the type attributes of the data object will apply, and not those of the data reference.
- J If a data reference variable is typed statically, it passes on its type attributes when it is assigned to a non-typed data reference.
- K You can use RTTI classes in ABAP to determine all of the type properties of a (data) object at runtime.



Answers

1. Which of the following statements regarding dynamic programming is correct?

Answer: A, D, F, G, H, J, K

Feedback



Course Summary

You should now be able to:

- Use fundamental elements of object-oriented modeling in UML
- Create ABAP Objects programs that contain all useful object-oriented programming techniques
- Use the relevant tools to create object-oriented Repository objects
- Describe and exploit the range of applications of ABAP Objects
- Define, raise, and handle class-based exceptions
- Query type and class attributes at runtime
- Make specialized changes to the SAP standard system
- Evaluate the different methods for modification and choose the right one for any given situation
- Explain the architecture of a Web Dynpro component
- Describe the parts of a Web Dynpro controller
- Create context elements in the Web Dynpro controller
- Explain how navigation and data transfer in and between Web Dynpro components can be implemented
- Define the UI of a Web Dynpro component
- Internationalize a Web Dynpro application
- Define and send messages in a Web Dynpro component
- Define input help and semantic help for UI elements in a Web Dynpro component

Related Information

- Use a URL or a cross-reference tag to point out additional information that the participants may find useful such as Web sites or White Papers. Delete this if it is not relevant.

Index

Numerics/Symbols

?=, 371
->*, 367, 369

A

abstraction, 17
ACTIVATION, 190
aggregation, 23, 54
alias name, 149, 157
area, 332
area classes, 332
area instance, 332
area root class, 335
ASSIGN, 363, 369
CASTING, 364
CASTING TYPE, 364
association, 21
binary and recursive, 21
attribute, 39
KERNEL_ERRID, 296
PREVIOUS, 308
private, 41
public, 41
static, 43

C

CALL METHOD, 56–57
EXCEPTIONS, 46
EXPORTING, 45
IMPORTING, 45
cardinality, 21
cast assignment operator, 371
CHANGING, 45
class, 15
abstract, 266

CL_ABAP_CLASSDE-
SCR, 386
CL_ABAP_STRUCTDE-
SCR, 384
CL_ABAP_TYPEDESCR,
382, 384
definition part, 39
final, 267
friendship, 270
instantiation, 268
Class
Definition, 38
CLASS, 227
ABSTRACT, 266
CREATE, 268
Definition, 38
FINAL, 267
FRIENDS, 270
INHERITING FROM, 106
PRIVATE SECTION, 42
PUBLIC SECTION, 42
class attribute, 43
Class Builder
display settings, 239
editing class components,
232
class diagram, 20
CLASS EVENTS, 187
class method, 48
CLASS_CONSTRUCTOR, 63
CLASS-METHODS, 48
classes
without multiple
instantiation, 269
component interface, 156

- composition, 23
- compound interface, 156
- constructor, 61, 109
 - static, 63
- C**
- CONSTRUCTOR, 61
- CREATE DATA, 368
 - TYPE, 368
- CREATE OBJECT, 52
- D**
- DATA
 - TYPE REF TO, 51, 366, 371
 - TYPE REF TO data, 366
- data reference, 366
- data reference variable, 366
- DEFAULT, 45
- DEFINITION, 39
- definition part, 39
- delegation, 26
- delegation principle, 11
- dereference, 371
- dereferencing, 367, 369, 373
- dereferencing operator, 367, 369, 371
- DESCRIBE FIELD, 381
- DESCRIBE TABLE, 381
- destructor, 61
- down cast, 118
- Down-Cast, 153
- Down-cast assignment operator, 153
- downcast, 118, 299
- E**
- encapsulation, 6, 41
- event
 - static, 184
- event control, 12
- event handler method, 185
- EVENTS, 187
- exception
 - class-based, 292
 - definition, 292
 - mapping, 308–309
 - propagate, 296, 303
 - exception chain, 294, 308
 - exception class, 294
 - CX_DYNAMIC_CHECK, 295, 307
 - CX_NO_CHECK, 295, 307
 - CX_ROOT, 295, 308
 - CX_STATIC_CHECK, 295, 307
 - CX_SY_ARITH-METIC_ERROR, 299
 - CX_SY_ARITH-METIC_OVERFLOW, 298
 - CX_SY_MOVE_CAST_ER-ROR, 120, 154
 - CX_SY_ZERODIVIDE, 296
 - global, 294
 - inheritance hierarchy, 307
 - local, 294
 - superclass, 307
 - exception handling, 294
 - CATCH, 296
 - CLEANUP, 297, 304
 - static check, 307
 - syntax check, 307
 - TRY, ENDTRY, 296
- exception instance, 294
- exception texts, 296
- F**
- field symbol, 362
- FIELD-SYMBOLS, 363
- friend concept, 270
- friendship relationship, 270
 - inheritance, 272
- function group
 - data management, 6

G

Garbage Collector, 53
generalization, 24, 104
global class
 print, 232
 test, 234

H

handler table, 191

I

information hiding, 41
inheritance, 12
instance, 16
instance attribute, 43
instance constructor, 61, 109
 visibility section, 268

instance event, 184
instance method, 48
instantiation

 class, 50
 data type, 368

interface

 class, 41
 compound, 156
 method, 45

INTERFACE, 148

interface resolution operator,
 148–149

INTERFACES, 148

IS ASSIGNED, 364
IS BOUND, 367
IS INITIAL, 53, 368

M

ME, 63
MESSAGE, 298
message class, 301
method
 abstract, 266
 call, 56–57
 definition, 45

DESCRIBE_BY_DATA,
 384

final, 267

functional, 58

GET_SOURCE_POSITION, 295

GET_TEXT, 295, 298, 302,
 304

parameter, 45

private, 47

public, 47

redefinition, 107

signature, 45

static, 48, 267

METHODS, 48

 ABSTRACT, 266
 FINAL, 267
 FOR EVENT, 188

MOVE, 52, 153

 down cast assignment
 operator, 118

 ?TO, 371

multiple inheritance, 106

multiple instantiation, 10

multiplicity, 21

N

namespace

 class, 47

narrowing cast assignment
 operator, 118

null reference, 51

O

object

 behavior, 19
 classification, 15
 identity, 16
 status, 16, 19

object diagram, 25

object link, 21

Object Management Group
 (OMG), 18

- OMG, 18
- Online Text Repository, 301
- OPTIONAL, 45
- OTR, 302
- P**
 - package, 19
 - pointer
 - dereferenced, 362
 - polymorphism, 12, 116
 - Polymorphism, 152
 - PRIVATE SECTION, 42
 - programming model
 - object-oriented, 10
 - procedural, 5
 - PROTECTED SECTION, 110
 - PUBLIC SECTION, 42
- R**
 - RAISE EVENT, 187
 - RAISING, 303
 - READ-ONLY, 40, 42
 - redefinition, 107
 - Refactoring Assistant, 242
 - reference
 - independent, 53
 - validity, 367
 - reference semantics, 366
 - reference types, 371
 - reference variable, 51, 366
 - data type, 371
 - Registration, 189, 191
 - RETURNING, 45
 - role, 21
 - role design pattern, 122
 - RTTC, 386
 - RTTI, 381
 - RTTS, 381
 - runtime error
 - BCD_ZERODIVIDE, 296
 - catchable, 294
 - Runtime Type Creation, 386
- Runtime Type Identification, 381
- S**
 - self-reference, 63
 - SENDER, 188
 - sequence diagram, 26
 - SET HANDLER, 190
 - ACTIVATION, 190
 - FOR ALL INSTANCES, 190
 - shared memory, 328
 - shared objects, 328
 - SHMA, 332
 - SHMM, 343
 - shopping cart, 328
 - Signature, 45
 - singleton, 269
 - singleton class, 271
 - specialization, 24, 103–104
 - static attribute, 43
 - static constructor, 63
 - static method, 48
 - subclass, 103–104
 - SUPER, 107
 - superclass, 103–104
- T**
 - TABLE_LINE, 53
 - text ID, 304
 - type
 - dynamic, 113
 - static, 113
 - TYPE REF TO, 51
 - TYPE REF TO data, 366
 - TYPES**
 - TYPE REF TO, 366, 371
 - TYPE REF TO data, 366
- U**
 - UML, 18
 - UNASSIGN, 364
 - Unified Modeling Language, 18
 - Up-Cast, 150

upcast, 113

V

value semantics, 362
visibility section

protected, 110

Z

zero reference, 367

Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.