

BC401

ABAP Objects

SAP NetWeaver

Date _____
Training Center _____
Instructors _____
Education Website _____

Participant Handbook

Course Version: 92
Course Duration: 5 Day(s)
Material Number: 50099907



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

Copyright

Copyright © 2011 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 and external.
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.....	33
Unit 2: Object-Oriented Concepts and Programming Techniques	101
Inheritance and Casting.....	103
Interfaces and Casting	153
Events	198
Unit 3: Object-Oriented Repository Objects.....	243
Global Classes and Interfaces	245
Special Object-Oriented Programming Techniques	316
Persistent Objects and OO-Transactions	355
Unit 4: Exception Handling and RTTS	369
Exception Handling in ABAP Objects	370
RTTS (Run Time Type Services)	420
Unit 5: Preview and Case Study	439
Preview of Later Topics	440
Case Study	452
Index	483

Course Overview

This course provides a comprehensive and detailed introduction to the programming techniques and the associated syntax of the ABAP Objects language. The only information that is not covered here belongs to separate subject areas like dialog programming or database access, which are discussed in detail in other courses.

This main objective of this course is to provide an introduction to the concepts and programming techniques of object-oriented development with ABAP Objects. This also includes an introduction to modeling.

Furthermore, you will become familiar with the tools in the ABAP Workbench that are used for object-oriented programming with ABAP Objects.

In another part of the course, you will be introduced to programming techniques that sometimes use object-oriented elements of ABAP Objects but can also be used in purely procedural contexts.

SAP's technical component, the **SAP NetWeaver Application Server**, referred to in this course as **SAP NW AS** for short, is embedded in the **SAP NetWeaver** solution and therefore provides a technical basis for almost all SAP solutions. This means that you can use the knowledge and skills you gain from this course not only for developing within **SAP ERP** but also in all solutions that have been developed with ABAP Objects.

Most of the programming techniques, tools, and syntax elements in this course are already available as of *SAP Web AS 6.10*. Only a few concepts were introduced in later releases or enhancement packs. Wherever such concepts are discussed in this course material, we tried to mention the release or enhancement pack as off which they are available.

Target Audience

This course is intended for the following audiences:

- Development project managers
- ABAP Objects developers

Course Prerequisites

Required Knowledge

- Programming experience
- SAPTEC (SAP NetWeaver: Application Platform Fundamentals)
- BC400 (ABAP Workbench: Foundation)



Course Goals

This course will prepare you to:

- Understand the details of procedural and object-oriented syntax elements in ABAP Objects and use these effectively
- Create object-oriented models of business applications
- Use the tools in the *ABAP Workbench* to develop object-oriented applications



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

SAP Software Component Information

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

- 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 is an introduction to 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 the later unit deals with concepts that are entirely new to you. The syntax will play a much smaller part 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	27
Lesson: Fundamental Object-Oriented Syntax Elements	33
Exercise 2: Local Classes.....	61
Exercise 3: Objects	67
Exercise 4: Method Calls	71
Exercise 5: Constructors	77
Exercise 6: Private Methods	85

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.

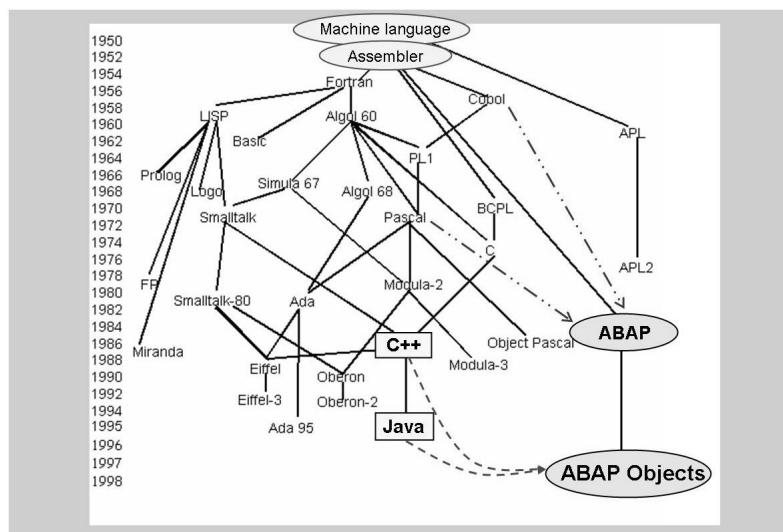
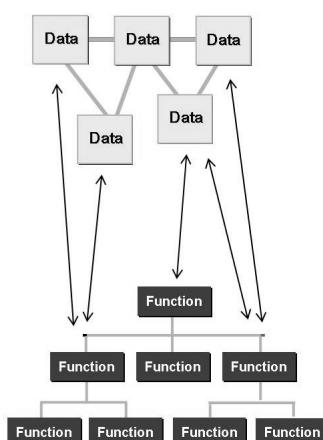


Figure 1: History of Selected Programming Languages

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. It is possible to read and change global variables from anywhere in the program.

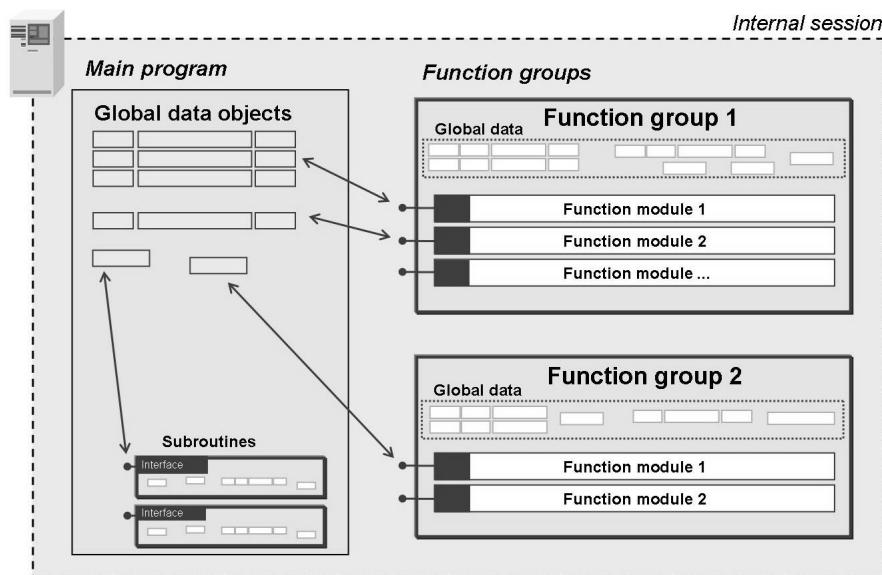


Figure 4: Encapsulating Data Using Function Groups

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

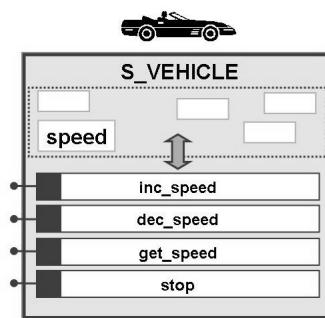


Figure 5: Example of a Function Group

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

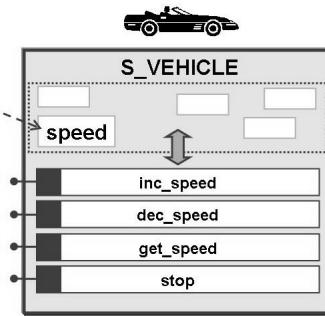


Figure 6: Example of Using the Function Group

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

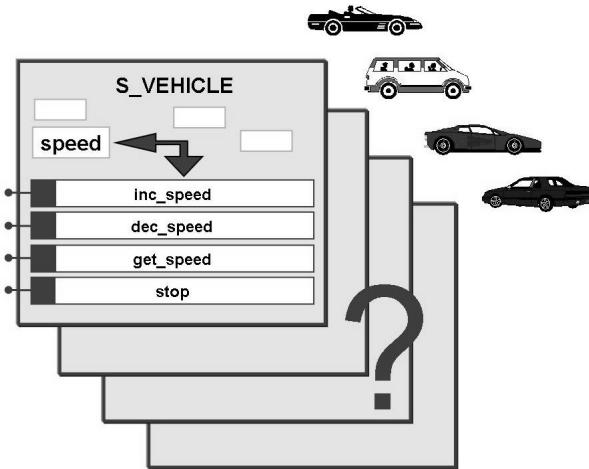


Figure 7: Several Instances of One Function Group?

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.

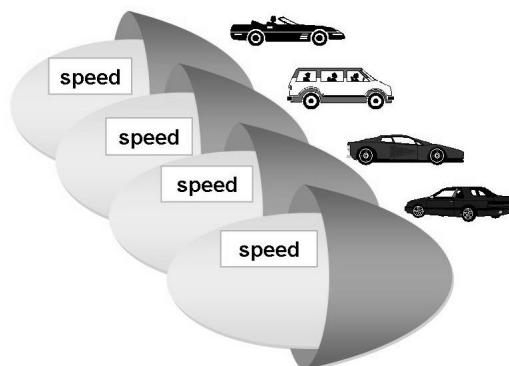


Figure 8: Multiple Instantiation in Object-Oriented Programming

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.

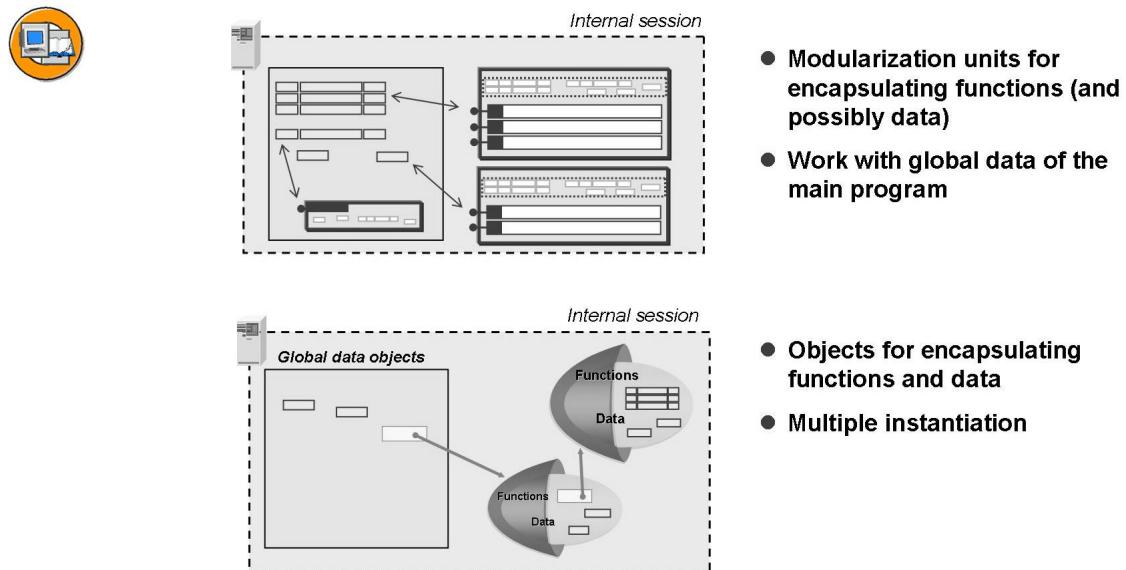


Figure 9: ABAP Main Memory and Encapsulation

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.

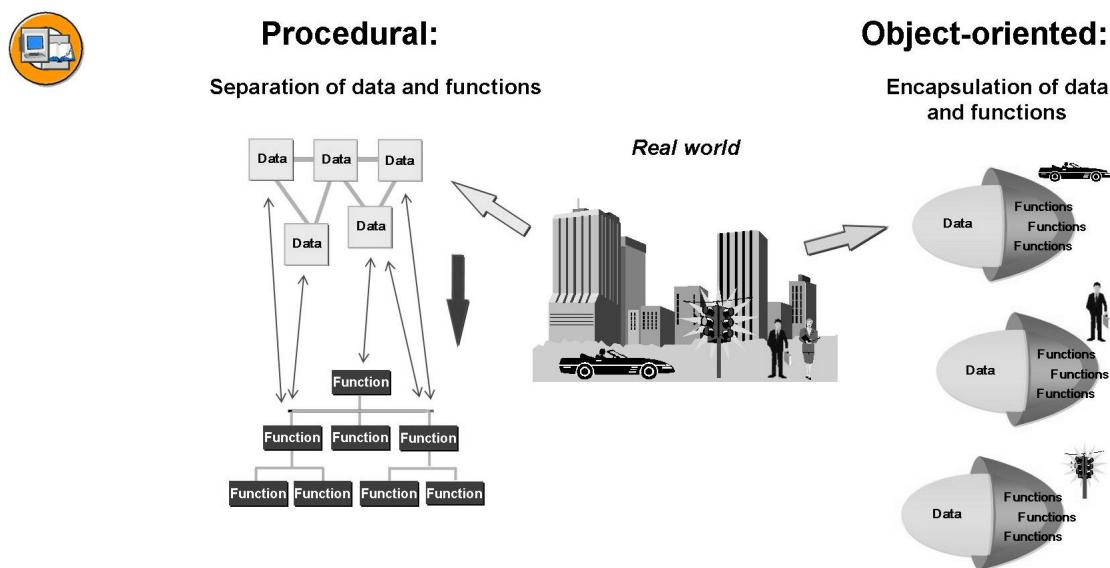


Figure 10: Data Management in Procedural and Object-Oriented Models – Summary

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 kna1.
      ...

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

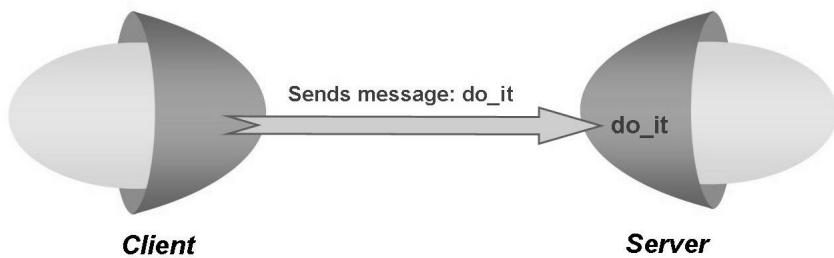


Figure 12: Client/Server Relationships Between Objects

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

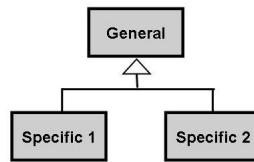
In object-oriented programming, the services are distributed amongst the objects in such a way as to avoid redundancies and so that each object offers exactly those services that are within its area of responsibility. If an object needs any other services, it requests these from other objects. This is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

Example:

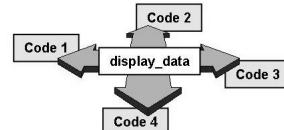


The common task “data retrieval and output” should be distributed over at least two objects. One is responsible for data retrieval and one for output. As long as the data retrieval object does not change its protocol, its internal implementation can be altered, without any changes to the output object 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**

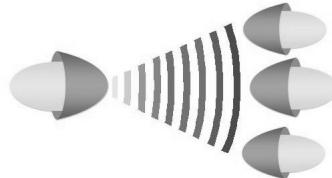


Figure 13: Additional Concepts of the Object-Oriented Programming Model

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.

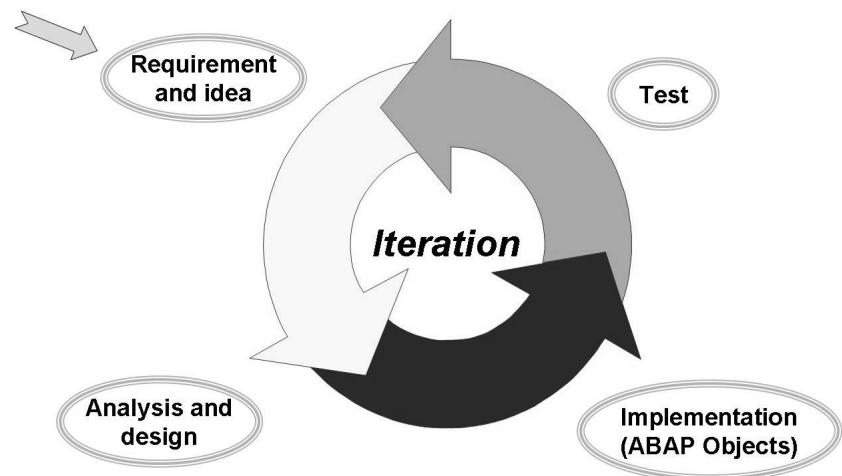


Figure 14: The Software Development Process

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

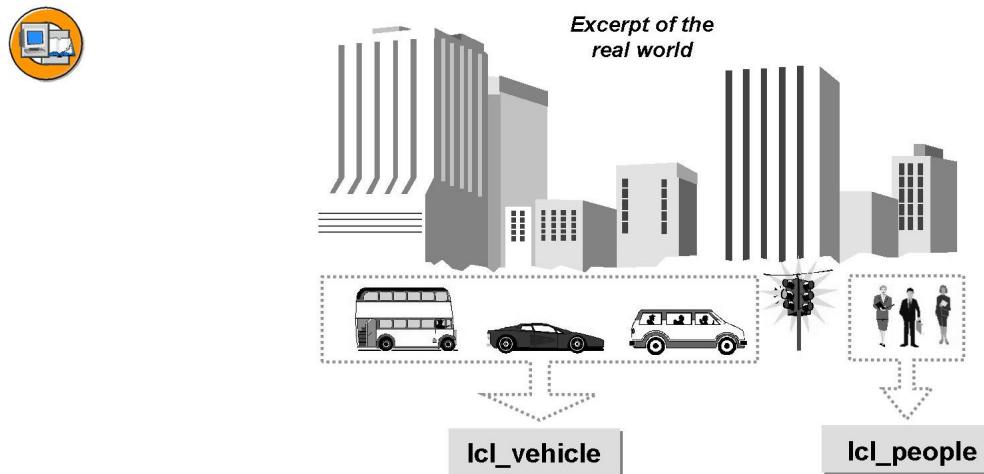
Business Example

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.

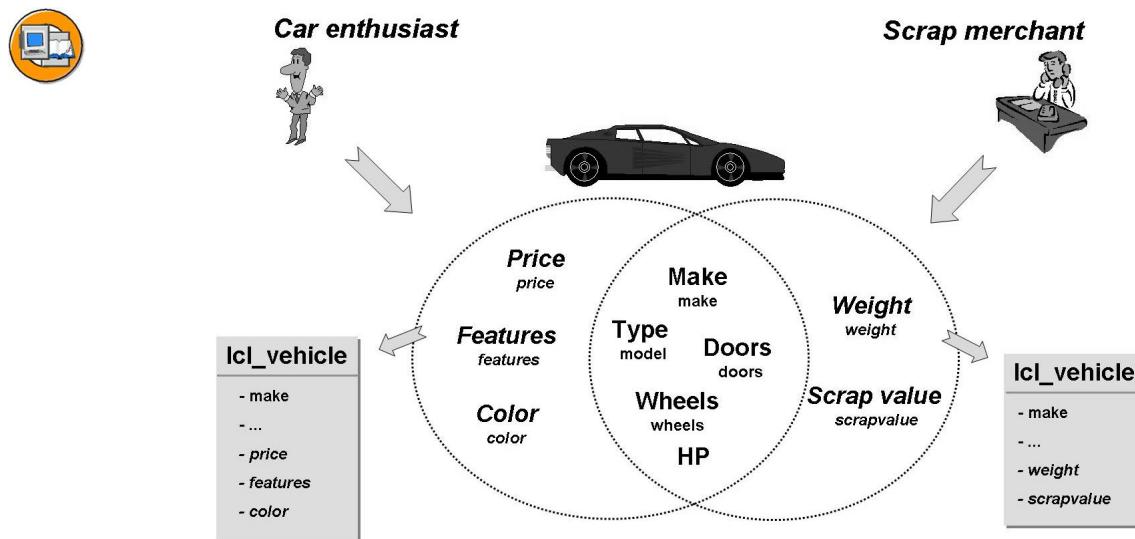
Consequently, an object has an identity, a status (number of characteristic instances), and a behavior. Do not confuse the concepts of identity and status. Identity is an attribute that distinguishes each object from all other objects of its class. Two different objects can have identical attribute values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

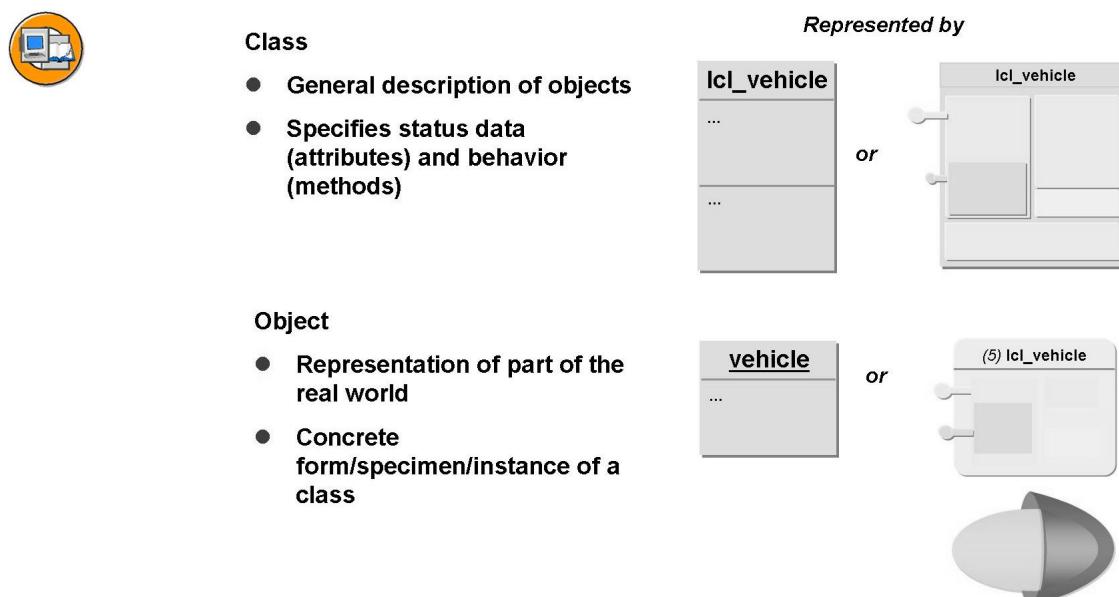
→ **Note:** In the literal sense of the word instance, the meaning is slightly more specific. It means a concrete – that is, uniquely identifiable – instance of a class.

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.

**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

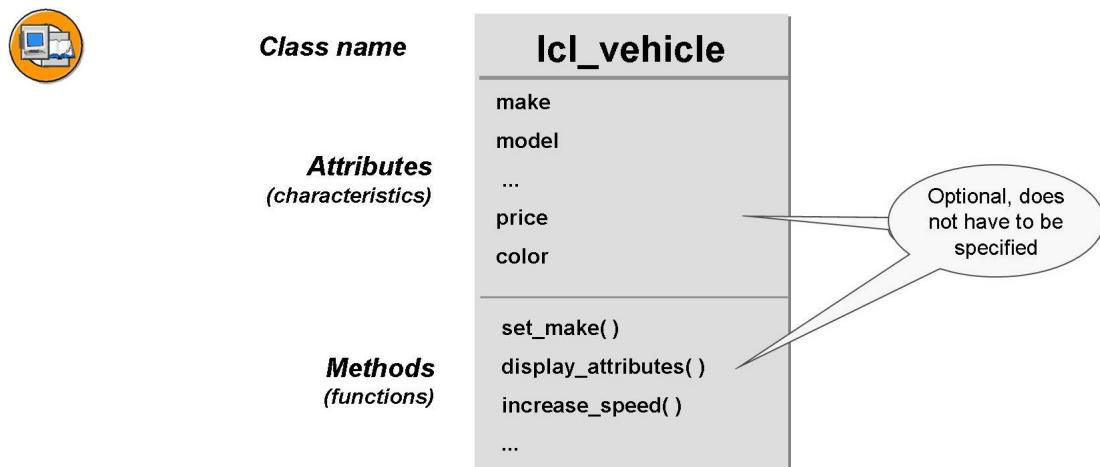
Pay particular attention to the **sequence** in which the **objects** relate to each other.

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.

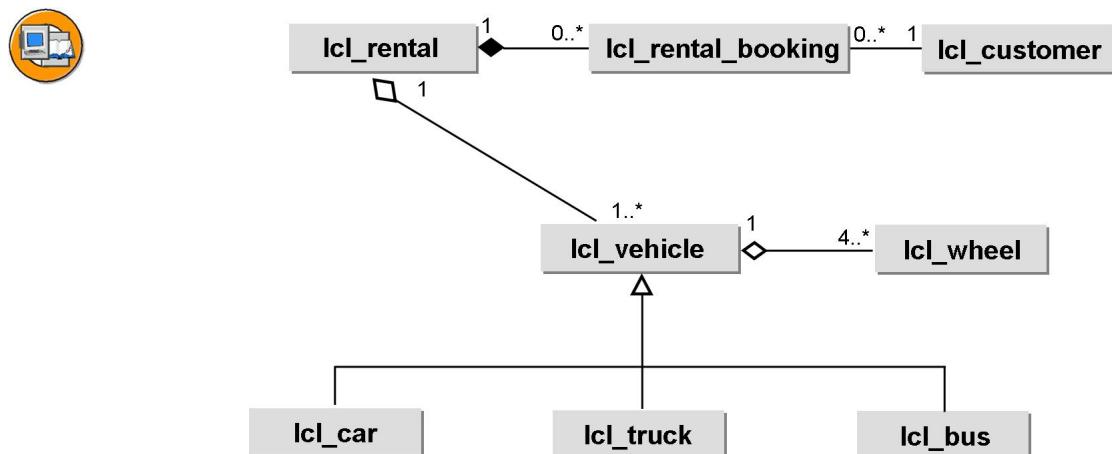
The third aspect is realized in the Repository object package. In other words: Packages can be used to realize the structure of software components within SAP Web AS according to the structure's specification in a component diagram.

**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.

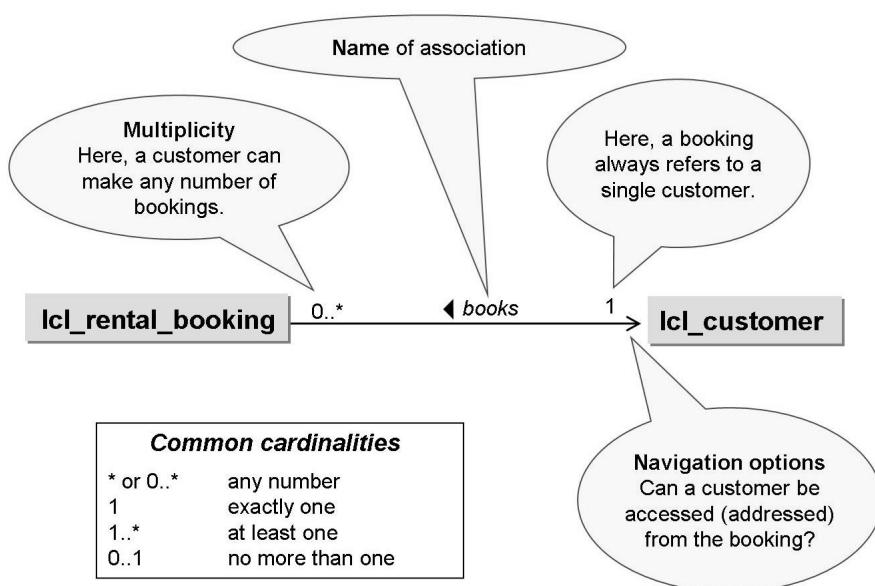


Figure 20: association

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.

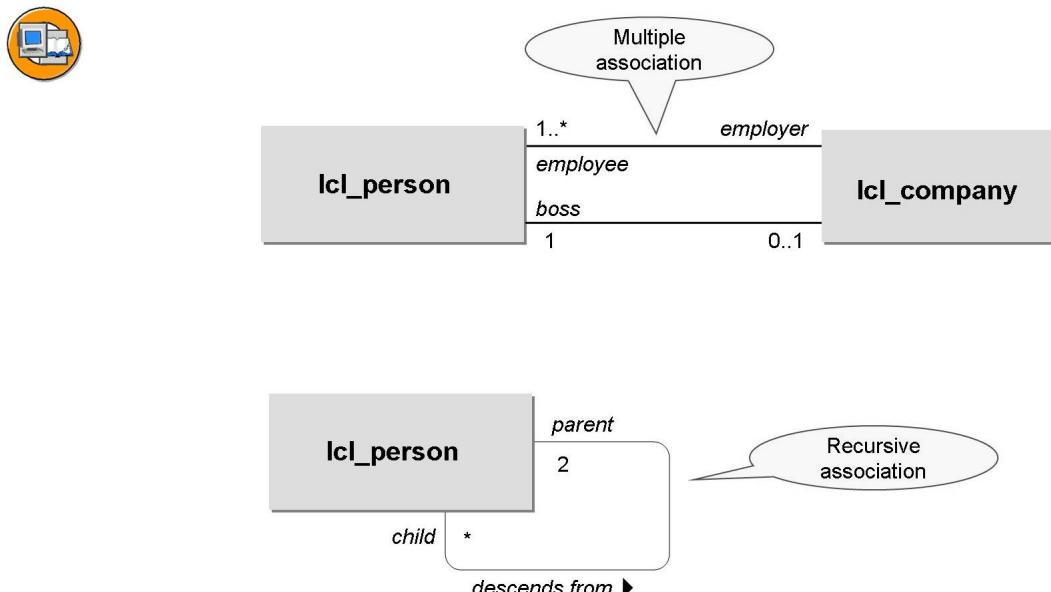
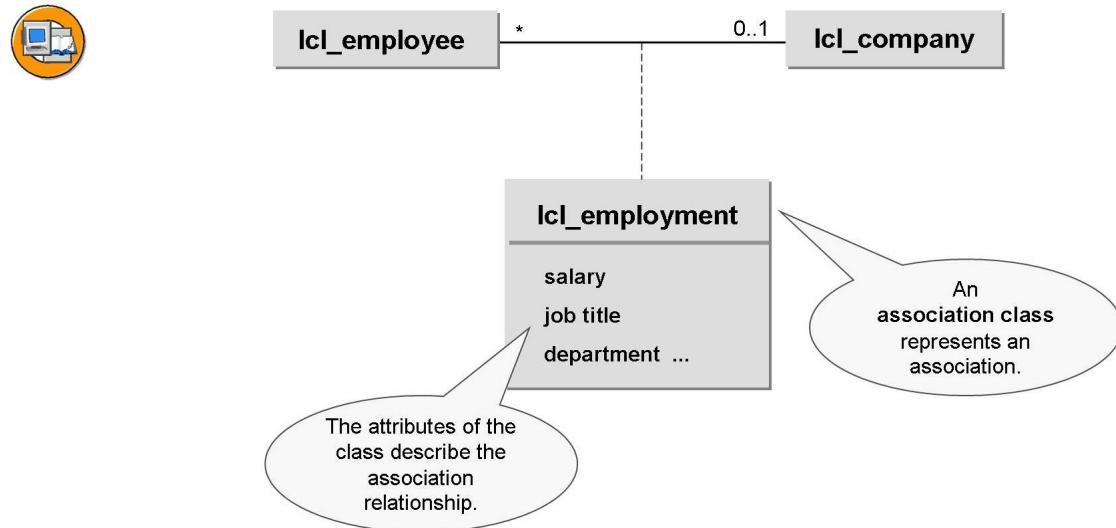


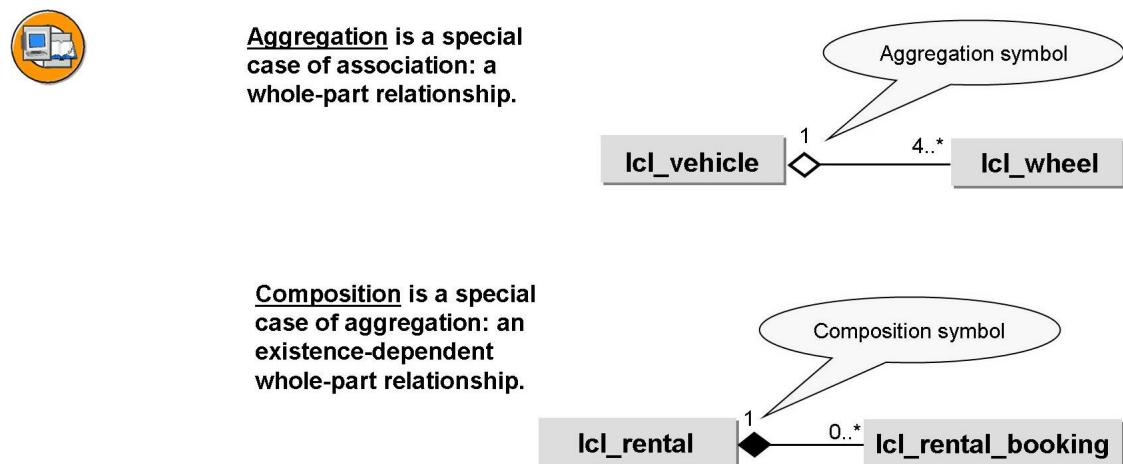
Figure 21: Association with Roles – Examples

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.

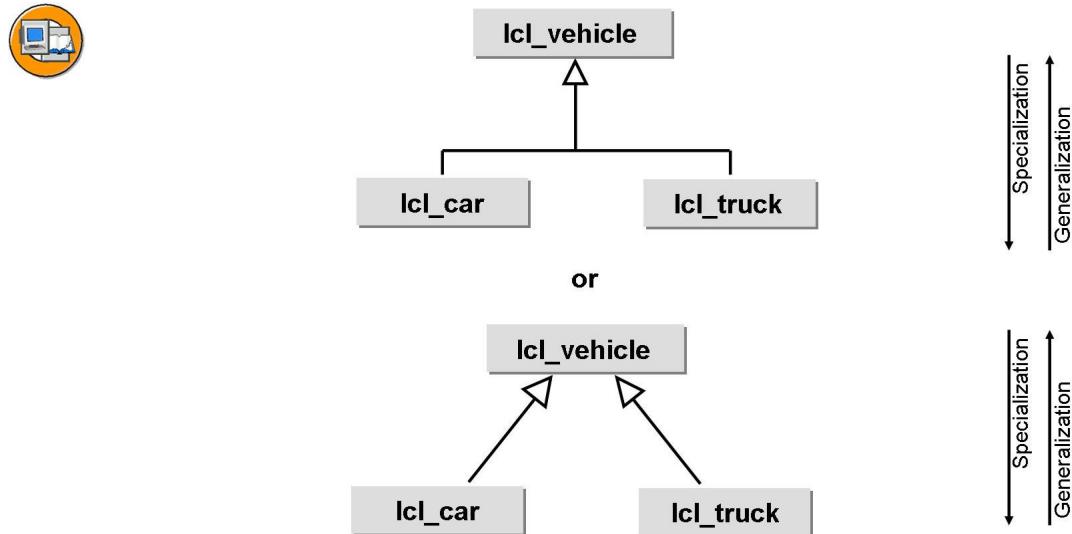
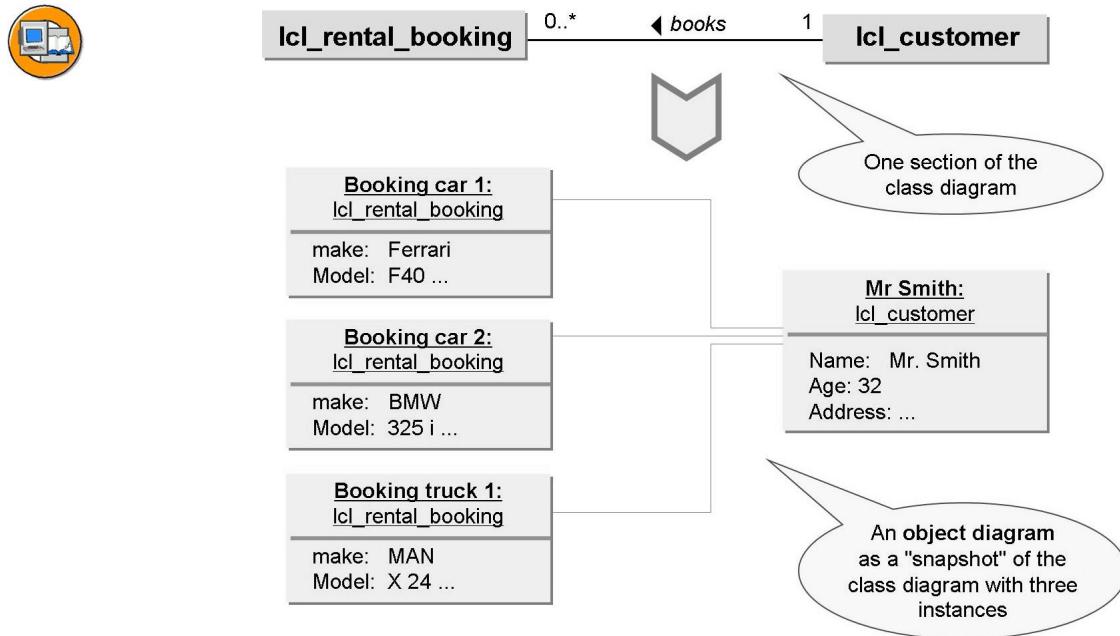


Figure 24: Generalization and Specialization

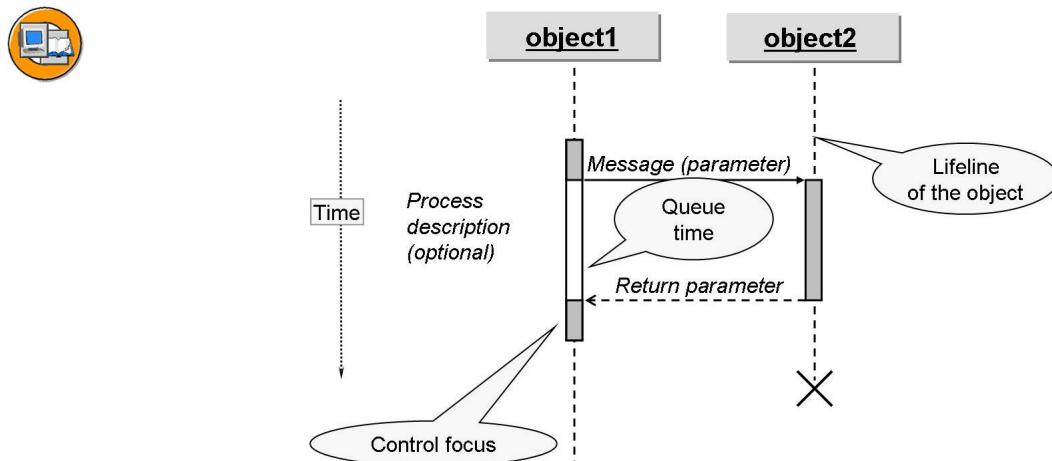
Generalization and specialization relationships are always bidirectional.
Generalization can be described with the words “is a special.”

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.

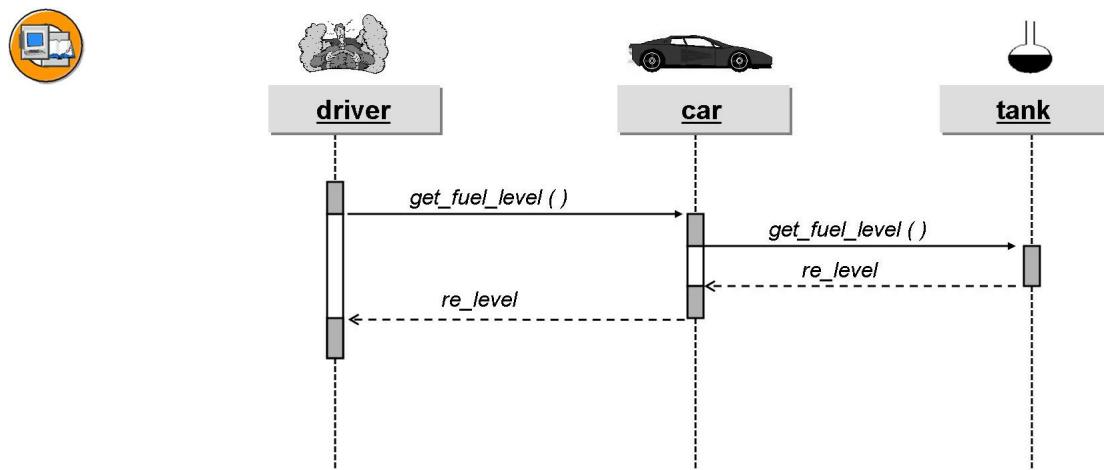


Figure 27: Delegation Principle in a Sequence Diagram

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

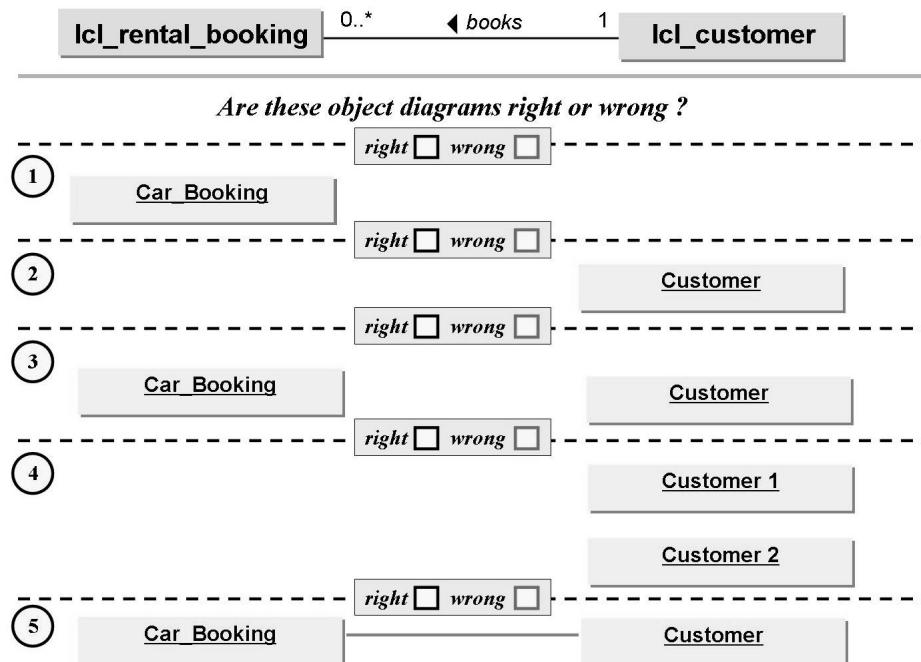


Figure 28: Possible Object Diagrams? (1)

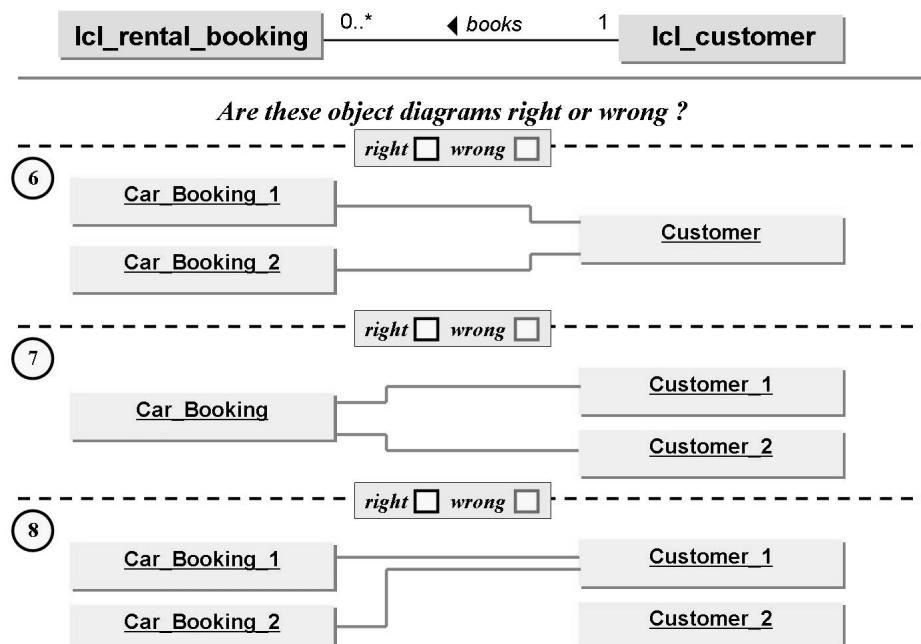


Figure 29: Possible Object Diagrams? (2)

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

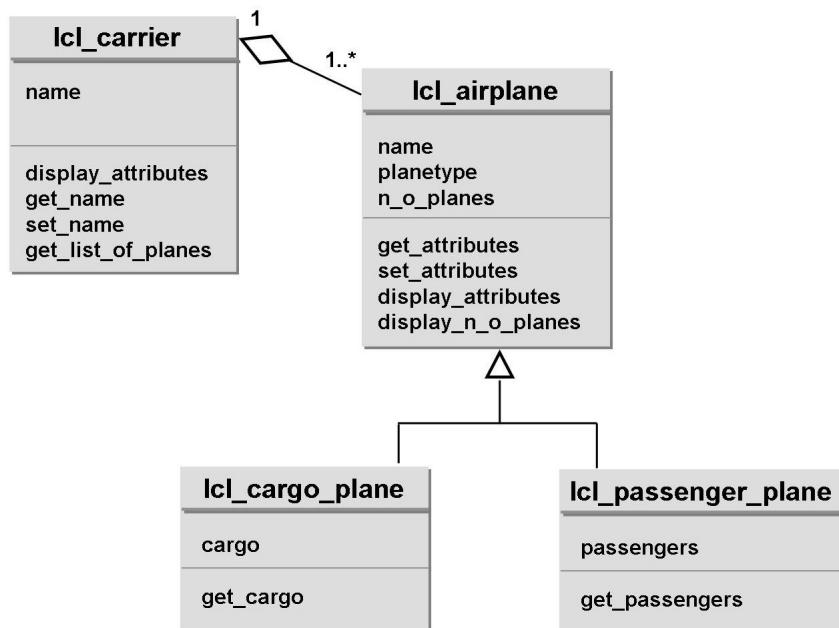


Figure 30: Class Diagram for Exercise: CARRIER/AIRPLANE

Task 2: Optional: Object Diagrams

Decide whether the following object diagrams are correct.

- 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

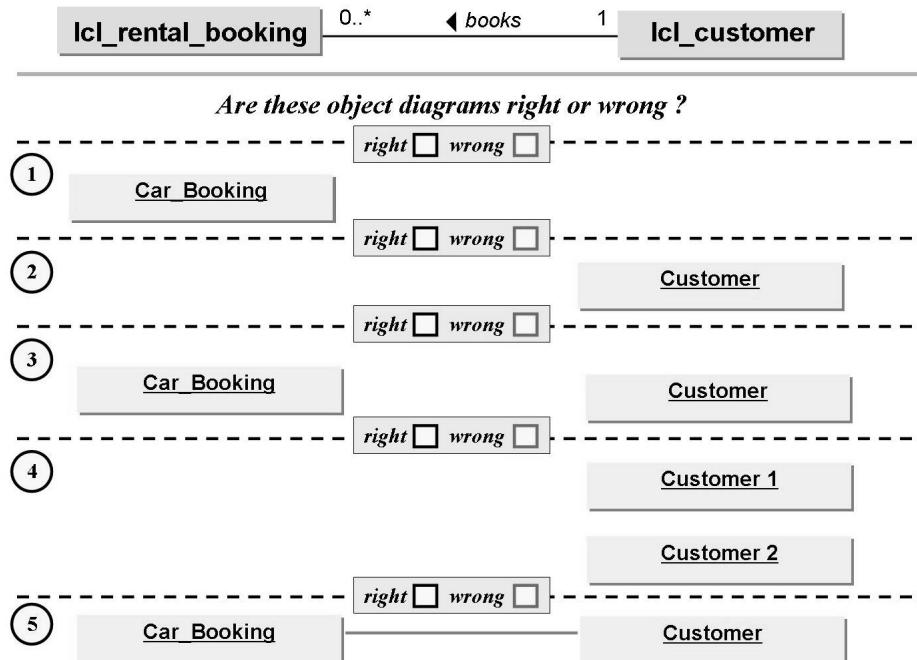


Figure 31: Possible Object Diagrams? (1)

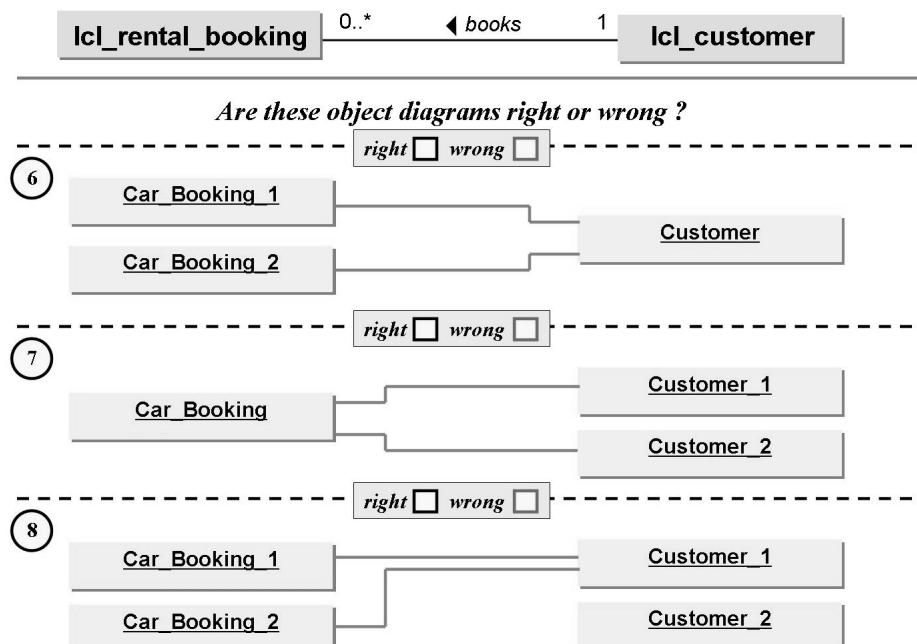


Figure 32: Possible Object Diagrams? (2)

- a) The object diagrams numbered 2, 4, 5 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:

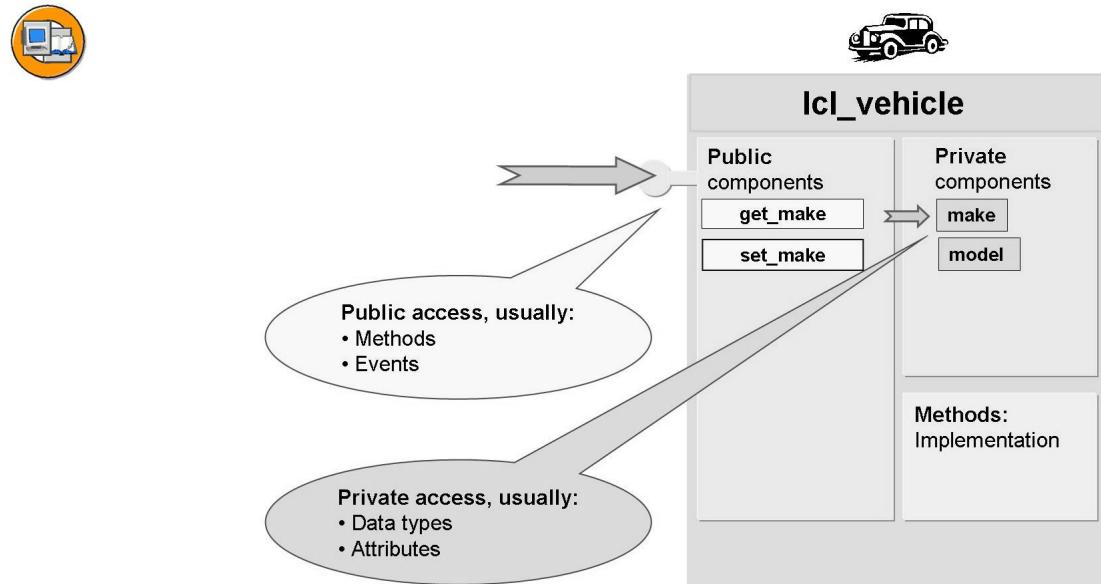
- Define classes
- Generate and delete objects
- Access attributes
- Call methods

Business Example

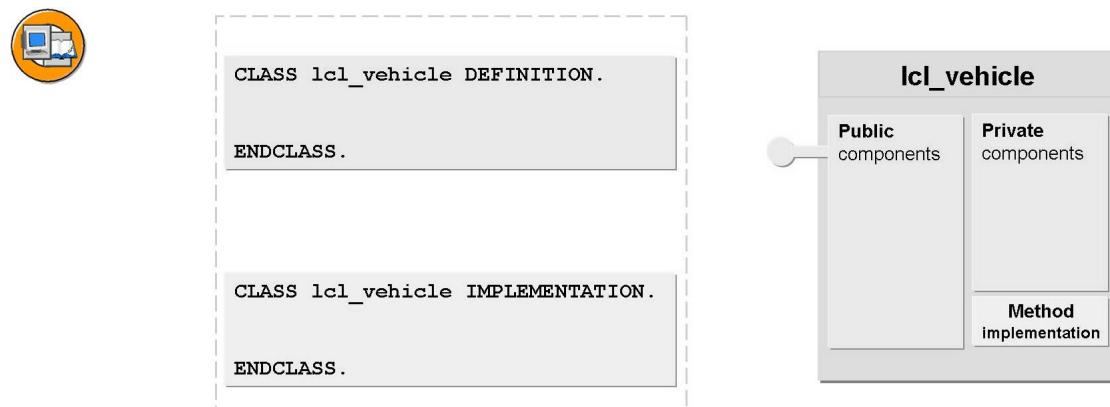
You want to implement classes, objects, and associations of your model in ABAP Objects.

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.

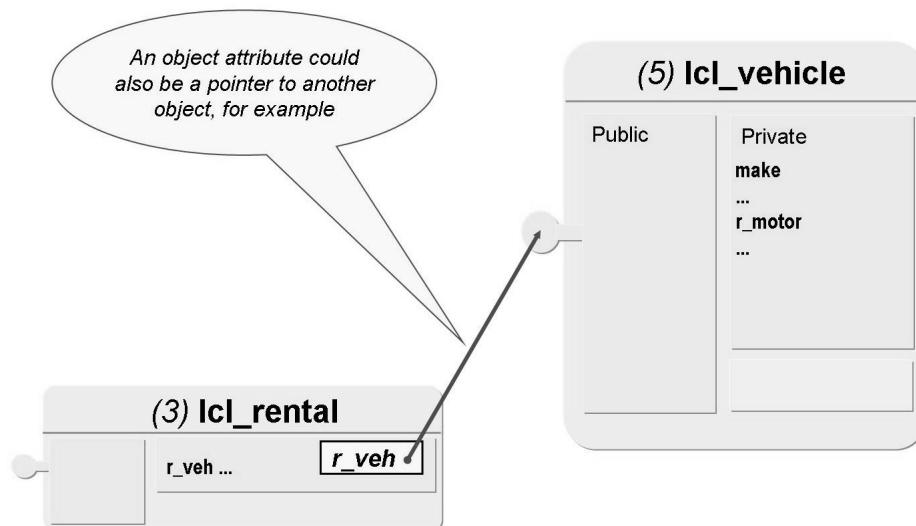


Figure 35: Example of Attributes

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.  
  
  ENDCLASS.
```

Reference variables
reference classes,
interfaces or types

Figure 36: Definition of Attributes, Types, and Constants

In DATA statements in classes, you can only use the **TYPE** addition to refer to data types. The **LIKE** addition is only 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 a reference. This will be discussed in more detail later.

The statement **CONSTANTS** is used within the class definition to define data objects that have a constant value.

If you use the statement **TYPES** within the class definition you declare a local type that is related especially to this class – for example, a type to be used for one or more attributes.

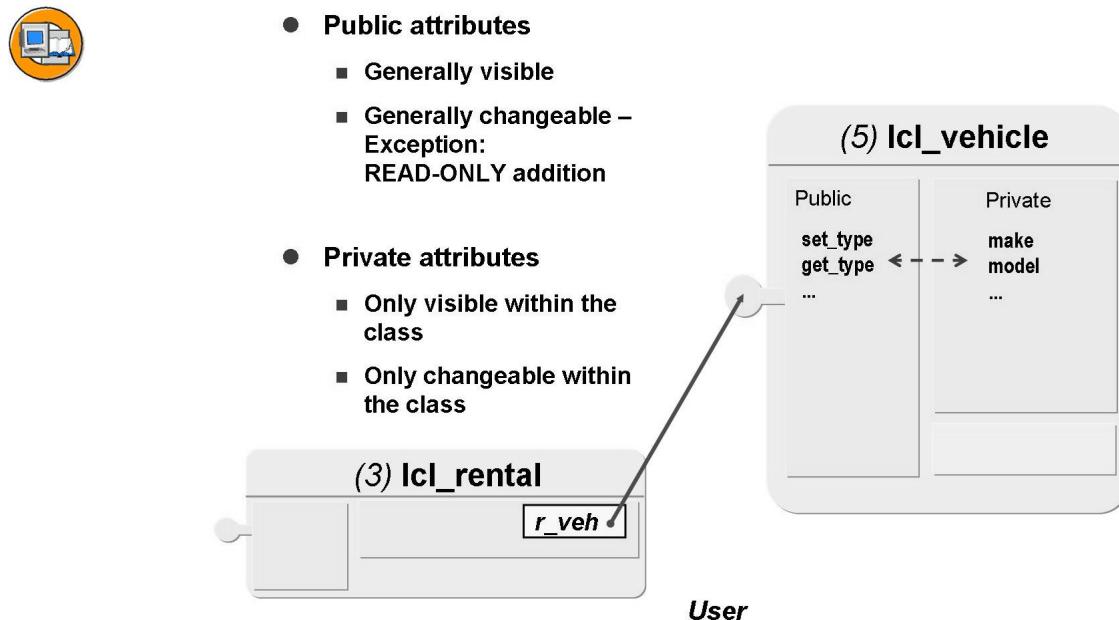


Figure 37: Visibility Sections of Attributes

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.

In a similar way the constants and types defined by a class can be either private, only to be used inside the class, or public, i.e. accessible from outside the class. 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: mv_make TYPE string,  
      mv_model TYPE string.  
  
PRIVATE SECTION.  
...  
ENDCLASS.
```



```
CLASS lcl_vehicle DEFINITION.  
  
PUBLIC SECTION.  
...  
  
PRIVATE SECTION.  
DATA: mv_make TYPE string,  
      mv_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 attributes 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

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

```
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION.
...
PRIVATE SECTION.
DATA:
  mv_make TYPE string,
  ...
CLASS-DATA:
  gv_n_o_vehicles TYPE i,
  ...
ENDCLASS.
```

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**.

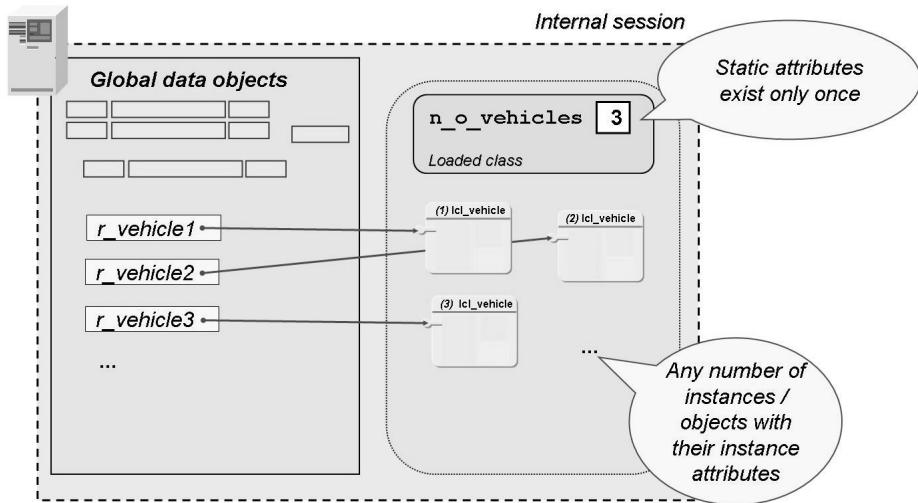


Figure 40: Instance Attributes and Static Attributes in the Program Context

The figure 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.

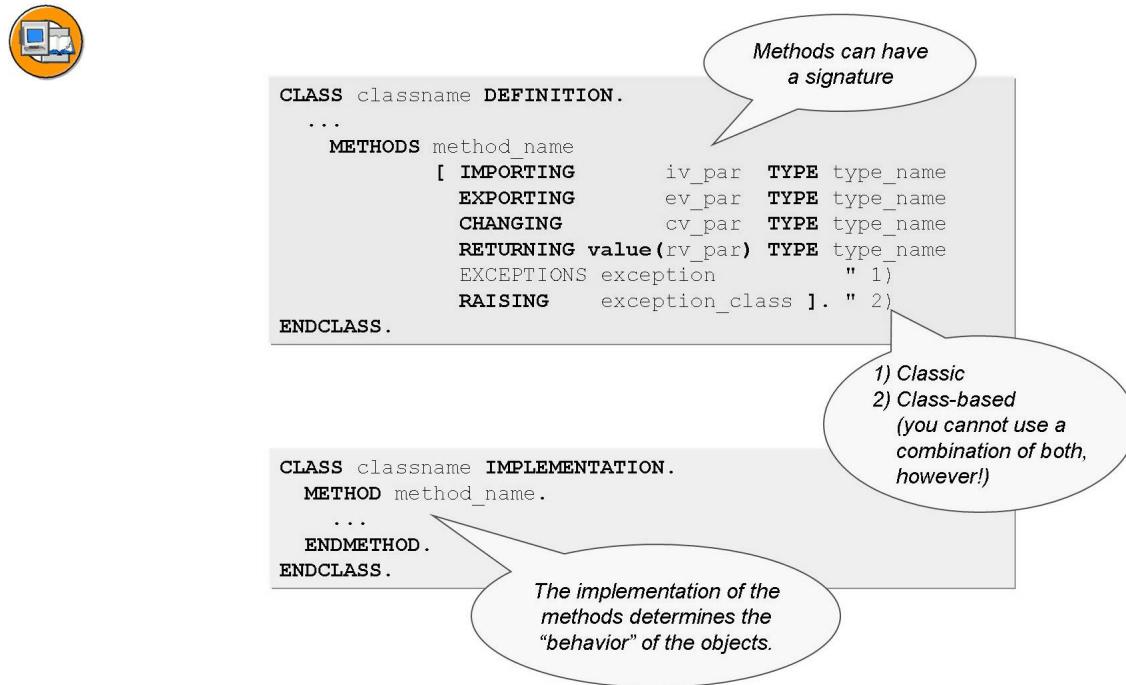


Figure 41: Syntax for Methods

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!

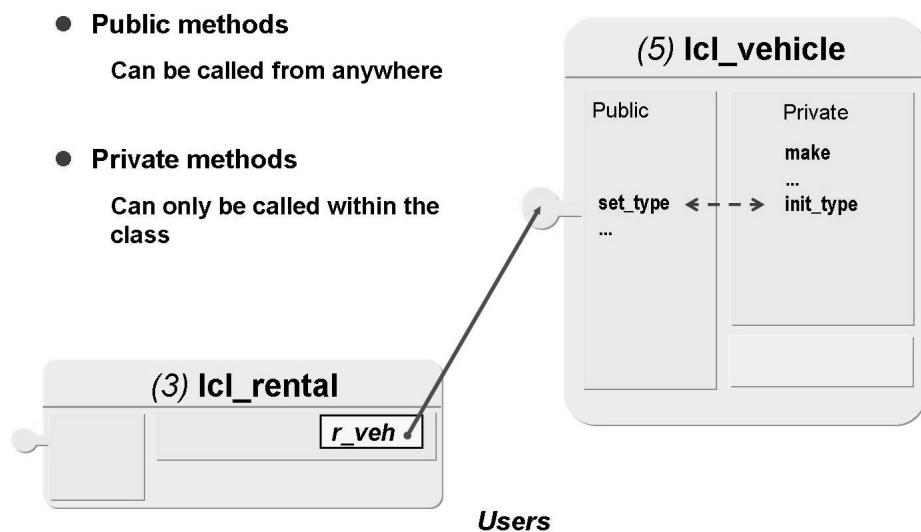


Figure 42: Visibility Sections of Methods

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 iv_make TYPE string  
           iv_model TYPE string.  
  PRIVATE SECTION.  
    METHODS init_type.  
    DATA: mv_make TYPE string,  
          mv_model TYPE string.  
  ENDCLASS.
```

```
CLASS lcl_vehicle IMPLEMENTATION.  
  Method init_type.  
    CLEAR: mv_make, mv_model.  
  ENDMETHOD.  
  Method set_type.  
    IF iv_make IS INITIAL.  
    *      calling Method init_type ...  
    ELSE.  
      mv_make = iv_make.  
      mv_model = iv_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 ev_count TYPE i.

  PRIVATE SECTION.
    DATA:
      mv_make TYPE string ,
      mv_model TYPE string .
    CLASS-DATA:
      gv_n_o_vehicles TYPE i.
ENDCLASS.
```

```
CLASS lcl_vehicle IMPLEMENTATION.
  ...
  METHOD get_n_o_vehicles .
    ev_count = gv_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**.

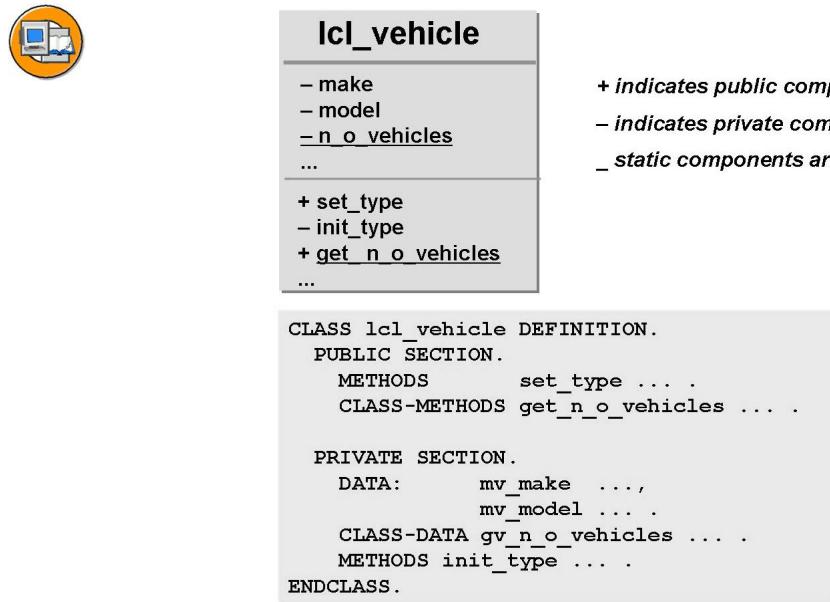


Figure 45: Visibility Sections and UML Notation

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

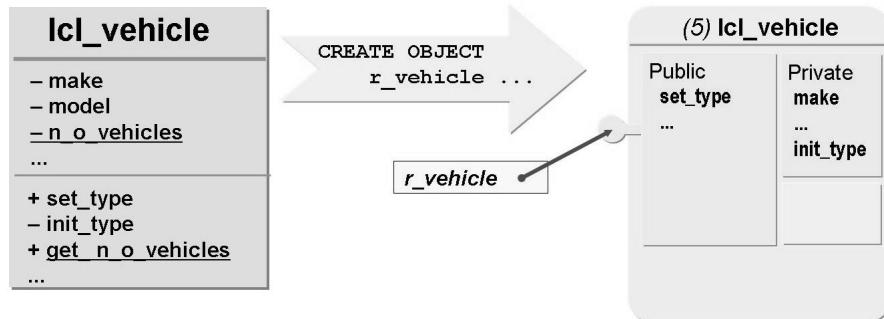


Figure 46: Overview of Instances of Classes

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: go_vehicle1 TYPE REF TO lcl_vehicle,
      go_vehicle2 LIKE go_vehicle1.

START-OF-SELECTION.
  ...

```

go_vehicle1 →

go_vehicle2 ←

Figure 47: Definition of Reference Variables

DATA go_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: go_vehicle1 TYPE REF TO lcl_vehicle,
      go_vehicle2 LIKE r_vehicle1.

CREATE OBJECT go_vehicle1.
CREATE OBJECT go_vehicle2.
```

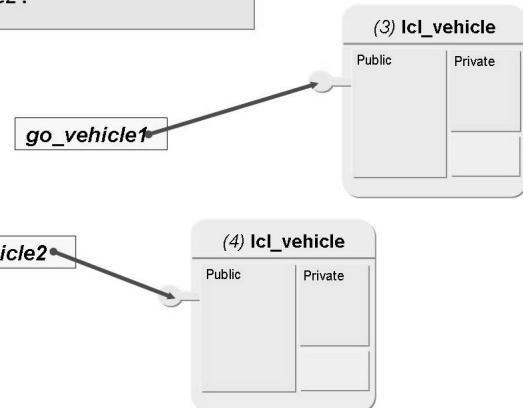


Figure 48: Creating Objects

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



```
DATA: go_vehicle1 TYPE REF TO lcl_vehicle,
      go_vehicle2 LIKE go_vehicle1,
      go_vehicle3 LIKE go_vehicle1.

CREATE OBJECT go_vehicle1.
go_vehicle2 = go_vehicle1.

CREATE OBJECT go_vehicle3.
```

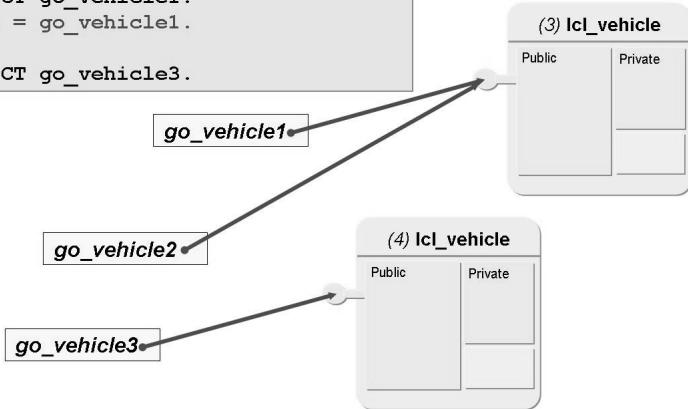


Figure 49: Reference Semantics of Object References

Reference variables can also be assigned to each other.

For the above example, this would mean that after the MOVE statement, GO_VEHICLE1 and GO_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.

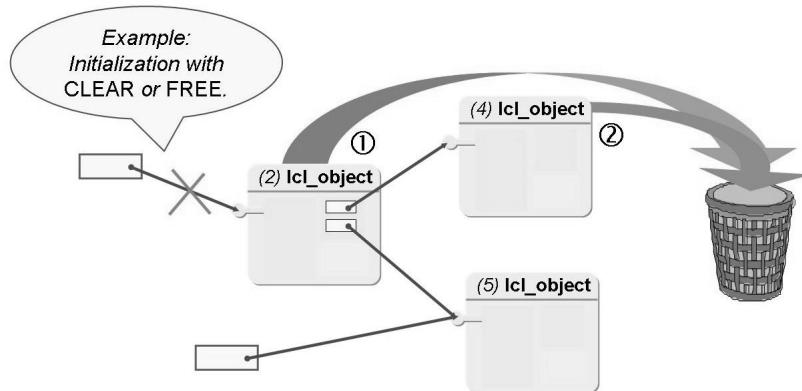


Figure 50: Garbage Collector

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 any more, so it is deleted as well.

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



```
DATA:  
  go_vehicle  TYPE REF TO lcl_vehicle,  
  gt_vehicles TYPE TABLE OF REF TO lcl_vehicle.
```

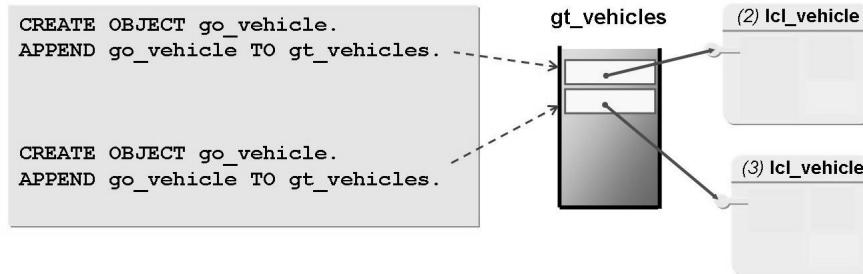


Figure 51: Reference Administration with Multiple Instantiation

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.



```
DATA: go_vehicle  TYPE REF TO lcl_vehicle,  
      go_wheel    TYPE REF TO lcl_wheel.  
  
CREATE OBJECT go_vehicle.  
DO ... TIMES.  
  CREATE OBJECT go_wheel.  
  * add wheel to vehicle  
  ...  
ENDDO.
```

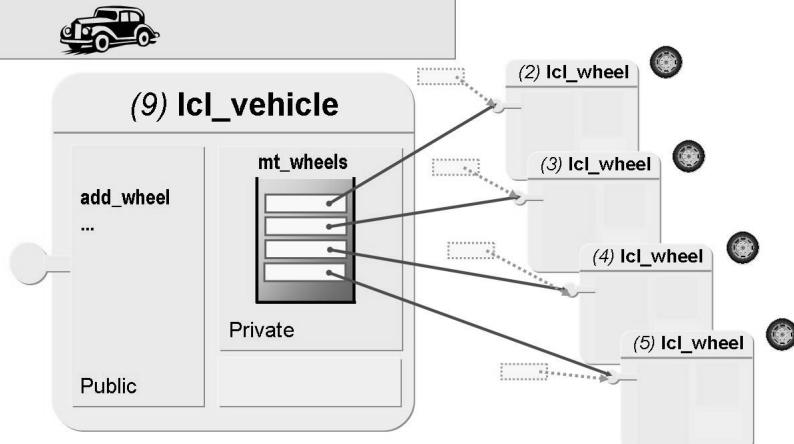


Figure 52: Example of Aggregation

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.

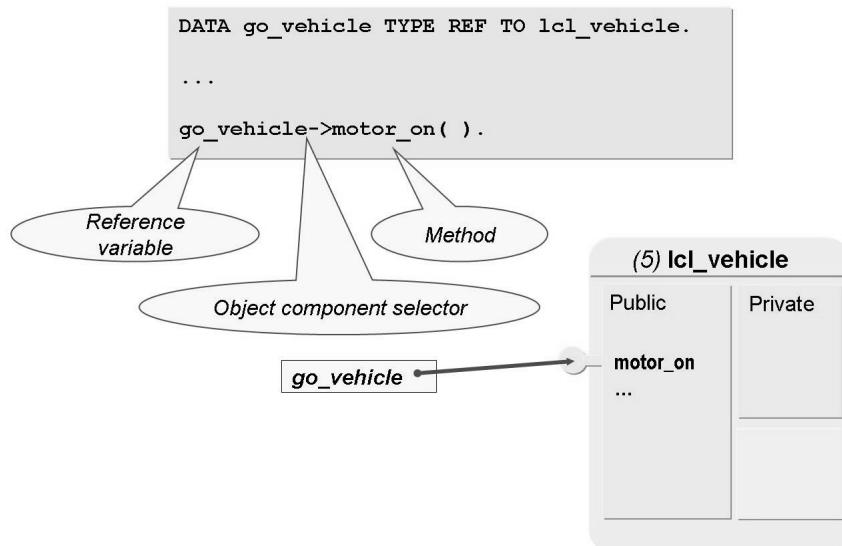


Figure 53: Calling Methods

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 iv_par      = val_ex ...
  IMPORTING ev_par      = val_im ...
  CHANGING cv_par      = val_chg ...
  RECEIVING rv_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 iv_par      = val_ex ...
  IMPORTING ev_par      = val_im ...
  CHANGING cv_par      = val_chg ...
  RECEIVING rv_par      = val_res ...
  EXCEPTIONS exception = val_rc ... ).
```

*See
functional
methods*

Example:

```
DATA: go_vehicle TYPE REF TO lcl_vehicle,
      gv_make TYPE string, gv_model TYPE string.
...
go_vehicle->get_type( IMPORTING ev_make = gv_make
                       ev_model = gv_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 iv_par      = val_ex ...
  IMPORTING ev_par      = val_im ...
  CHANGING cv_par      = val_chg ...
  RECEIVING rv_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 iv_par      = val_ex ...
  IMPORTING ev_par      = val_im ...
  CHANGING cv_par      = val_chg ...
  RECEIVING rv_par      = val_res
  EXCEPTIONS exception = val_rc ... ).
```

Refer to
functional
methods!

Example:

```
DATA gv_number TYPE i.
...
lcl_vehicle=>get_n_o_vehicles( IMPORTING ev_count = gv_number ).
```

Figure 55: Calling Static Methods - Syntax

Static methods (also referred to as class methods) are called using **CALL METHOD classname=>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( iv_par_1 = val_ex_1
                                ...
                                iv_par_n = val_ex_n ).
```

```
result = class_name=>func_method_name( iv_par_1 = val_ex_1
                                         ...
                                         iv_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 iv_distance      TYPE s_distance
              iv_fuel            TYPE s_capacity
      RETURNING value(rv_fuel)   TYPE s_consum.
  ENDCLASS.

DATA: go_veh1 TYPE REF TO lcl_vehicle,
      go_veh2 TYPE REF TO lcl_vehicle,
      gv_avg_fuel TYPE s_consum.
...
gv_avg_fuel =
  go_veh1->get_average_fuel( iv_distance = 500 iv_fuel = '50.0' )
+ go_veh2->get_average_fuel( iv_distance = 600 iv_fuel = '60.0' ).
```

```

DATA gv_number TYPE i.
...
gv_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 gv_number TYPE i.
...
CALL METHOD lcl_vehicle=>get_n_o_vehicles
  RECEIVING rv_count = gv_number.
```

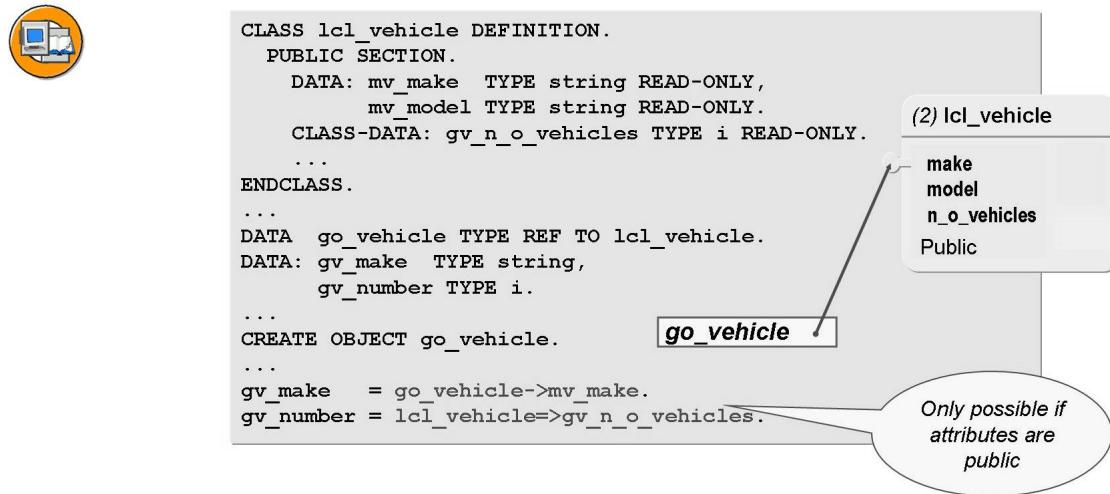


Figure 58: Accessing Public Attributes

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.

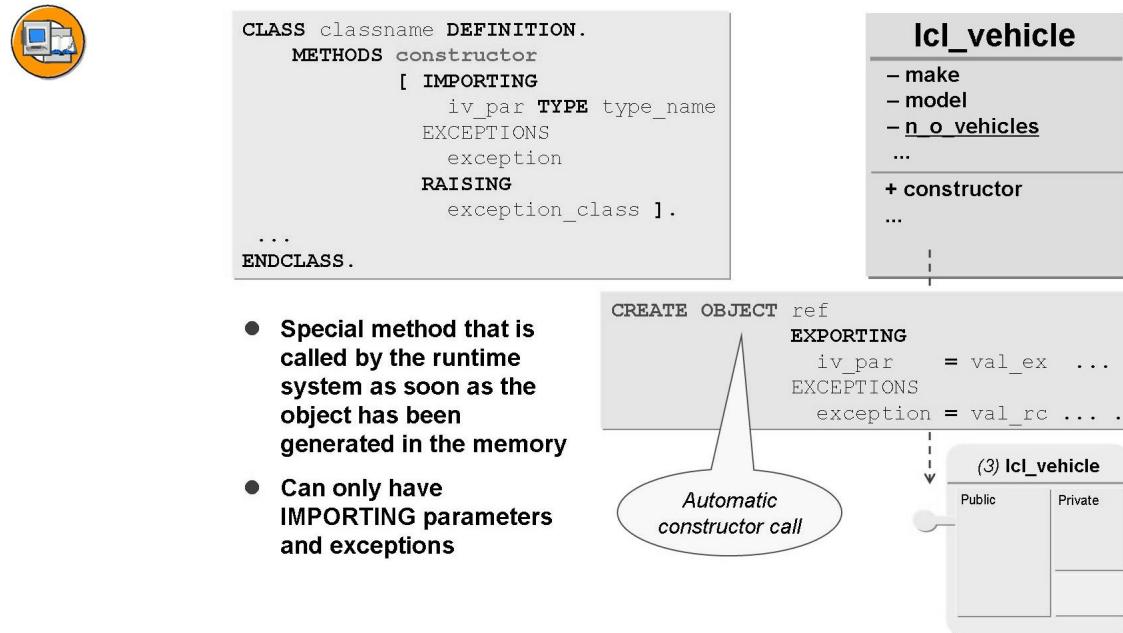


Figure 59: (Instance) Constructor

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 has not 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 iv_make TYPE string  
           iv_model TYPE string.  
  PRIVATE SECTION.  
    DATA: mv_make TYPE string, mv_model TYPE string.  
    CLASS-DATA gv_n_o_vehicles TYPE i.  
  ENDCLASS.
```

```
CLASS lcl_vehicle IMPLEMENTATION.  
  METHOD constructor.  
    mv_make = iv_make.  
    mv_model = iv_model.  
    ADD 1 TO gv_n_o_vehicles.  
  ENDMETHOD.  
ENDCLASS.
```

```
DATA go_vehicle TYPE REF TO lcl_vehicles.  
...  
CREATE OBJECT go_vehicle  
  EXPORTING iv_make = 'Ferrari'  
           iv_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 gv_n_o_vehicles TYPE i.
ENDCLASS.

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

```

```
gv_number = 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 only 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 not 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.

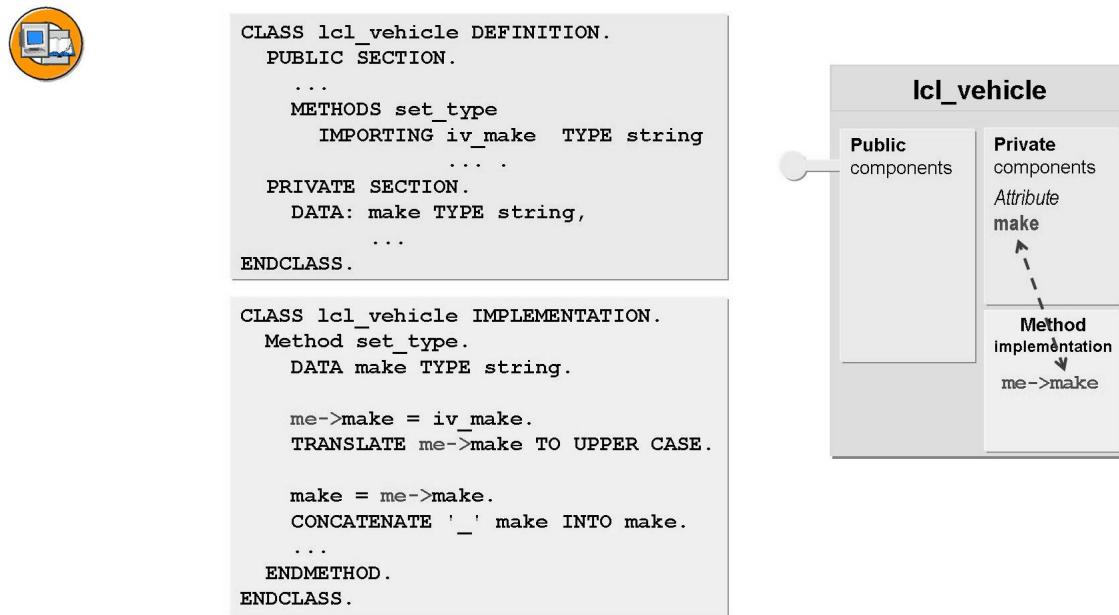


Figure 62: Self Reference

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.

Template:

None

Solution:

SAPBC401_BAS_S1

Task 1:

Create a new program.

1. Create an executable program without a TOP include. (Suggested name: **ZBC401_##_MAIN** (where ## is your two-digit group number.)

Task 2:

Declare and implement a class for airplanes.

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

MV_NAME (name of airplane), data type STRING

mv_planetype (type of airplane), data type SAPLANE-PLANETYPE
and the private static attribute

MV_N_O_AIRPLANES (instance counter), data type I.

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.

Continued on next page

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.

6. In order to improve the list output, arrange the output in two columns: One for the descriptive text and one for the attribute values.



Hint: Use addition AT <number> of the write-statement to place the output at a given column.

To make later adjustments easier don't use a literal for the number. Define the private constant C_POS_1 instead.

Solution 2: Local Classes

Task 1:

Create a new program.

1. Create an executable program without a TOP include. (Suggested 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.

Task 2:

Declare and implement a class for airplanes.

1. Within your program, declare the local class LCL_AIRPLANE.
 - a) See the source code extract from the model solution.
2. Define the two private instance attributes
MV_NAME (name of airplane), data type STRING
mv_planetype (type of airplane), data type SAPLANE-PLANETYPE
and the private static attribute
MV_N_O_AIRPLANES (instance counter), data type I.
 - a) See the source code extract from the model solution.
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.

Continued on next page

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.
6. In order to improve the list output, arrange the output in two columns: One for the descriptive text and one for the attribute values.



Hint: Use addition `AT <number>` of the write-statement to place the output at a given column.

To make later adjustments easier don't use a literal for the number. Define the private constant `C_POS_1` instead.

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

Result

Source code extract:

Continued on next page

SAPBC401_BAS_S1

```

REPORT  sapbc401_bas_s1.

TYPE-POOLS icon.

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

PUBLIC SECTION.

METHODS:
    set_attributes
        IMPORTING
            iv_name      TYPE string
            iv_planetype TYPE saplane-planetype,
        display_attributes.

CLASS-METHODS:
    display_n_o_airplanes.

PROTECTED SECTION.

PRIVATE SECTION.

CONSTANTS:
    c_pos_1 TYPE i VALUE 30.

DATA:
    mv_name      TYPE string,
    mv_planetype TYPE saplane-planetype.

CLASS-DATA:
    gv_n_o_airplanes TYPE i.

ENDCLASS.          "lcl_airplane DEFINITION

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

```

Continued on next page

```
METHOD set_attributes.  
  
    mv_name      = iv_name.  
    mv_planetype = iv_planetype.  
  
    * doesn't make sense so much -  
    * only in order to get an effect  
    * after calling display_n_o_airplanes:  
        gv_n_o_airplanes = gv_n_o_airplanes + 1.  
  
ENDMETHOD.           "set_attributes  
  
METHOD display_attributes.  
    DATA: lv_weight TYPE saplane-weight,  
          lv_cap     TYPE saplane-tankcap.  
  
    WRITE:  
        / icon_ws_plane AS ICON,  
        / 'Name of Airplane'(001) ,   AT c_pos_1 mv_name,  
        / 'Type of Airplane:'(002) ,   AT c_pos_1 mv_planetype.  
  
ENDMETHOD.           "display_attributes  
  
METHOD display_n_o_airplanes.  
    SKIP.  
    WRITE:  
        / 'Number of airplanes:'(cal),  
        AT c_pos_1 gv_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 the references to the objects are not lost.

Template:

SAPBC401_BAS_S1

Solution:

SAPBC401_BAS_S2

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.



Hint: Make sure to use the ABAP keyword START-OF-SELECTION to specify, where the executable part of the main program begins. Otherwise the executable statements will be classified “unreachable” by the syntax check.

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) 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.



Hint: Make sure to use the ABAP keyword START-OF-SELECTION to specify, where the executable part of the main program begins. Otherwise the executable statements will be classified “unreachable” by the syntax check.

- a) See the source code extract from the model solution.
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_BAS_S2

```
REPORT  sapbc401_bas_s2.
```

```
TYPE-POOLS icon.
```

```
*-----*
```

Continued on next page

```
*      CLASS lcl_airplane DEFINITION          *
*-----*
CLASS lcl_airplane DEFINITION.

...
ENDCLASS.           "lcl_airplane DEFINITION

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

...
ENDCLASS.           "lcl_airplane IMPLEMENTATION

DATA:
go_airplane TYPE REF TO lcl_airplane.

DATA:
gt_airplanes TYPE TABLE OF REF TO lcl_airplane.

START-OF-SELECTION.
*****CREATE OBJECT go_airplane.
APPEND go_airplane TO gt_airplanes.

CREATE OBJECT go_airplane.
APPEND go_airplane TO gt_airplanes.

CREATE OBJECT go_airplane.
APPEND go_airplane TO gt_airplanes.
```


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.

Template:

SAPBC401_BAS_S2

Solution:

SAPBC401_BAS_S3

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').
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 RV_COUNT, which must be an integer.

Continued on next page

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) 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').
 - 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 RV_COUNT, which must be an integer.
 - 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:

SAPBC401_BAS_S3

c401_bas_s3.

Continued on next page

```

TYPE-POOLS icon.

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

PUBLIC SECTION.

METHODS:
    set_attributes
        IMPORTING
            iv_name      TYPE string
            iv_planetype TYPE saplane-planetype,
    display_attributes.

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

PROTECTED SECTION.

PRIVATE SECTION.

CONSTANTS:
    c_pos_1 TYPE i VALUE 30.

DATA:
    mv_name      TYPE string,
    mv_planetype TYPE saplane-planetype.

CLASS-DATA:
    gv_n_o_airplanes TYPE i.
ENDCLASS.           "lcl_airplane DEFINITION

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

METHOD set_attributes.

```

Continued on next page

```

mv_name      = iv_name.
mv_planetype = iv_planetype.

*   doesn't make sense so much -
*   only in order to get an effect
*   after calling display_n_o_airplanes:
gv_n_o_airplanes = gv_n_o_airplanes + 1.

ENDMETHOD.          "set_attributes

METHOD display_attributes.
DATA: lv_weight TYPE saplane-weight,
      lv_cap    TYPE saplane-tankcap.

WRITE:
/ icon_ws_plane AS ICON,
/ 'Name of Airplane'(001) , AT c_pos_1 mv_name,
/ 'Type of Airplane:'(002) , AT c_pos_1 mv_planetype.

ENDMETHOD.          "display_attributes

METHOD display_n_o_airplanes.
SKIP.
WRITE:
/ 'Number of airplanes:'(ca1),
  AT c_pos_1 gv_n_o_airplanes LEFT-JUSTIFIED.
ENDMETHOD.          "display_n_o_airplanes

METHOD get_n_o_airplanes.
rv_count = gv_n_o_airplanes.
ENDMETHOD.          "get_n_o_airplanes

ENDCLASS.           "lcl_airplane IMPLEMENTATION

DATA:
go_airplane  TYPE REF TO lcl_airplane,
gt_airplanes TYPE TABLE OF REF TO lcl_airplane,
gv_count     TYPE i.

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

CREATE OBJECT go_airplane.

```

Continued on next page

```
APPEND go_airplane TO gt_airplanes.  
go_airplane->set_attributes( iv_name      = 'LH Berlin'  
                             iv_planetype = 'A321' ).  
  
CREATE OBJECT go_airplane.  
APPEND go_airplane TO gt_airplanes.  
go_airplane->set_attributes( iv_name      = 'AA New York'  
                             iv_planetype = '747-400' ).  
  
CREATE OBJECT go_airplane.  
APPEND go_airplane TO gt_airplanes.  
go_airplane->set_attributes( iv_name      = 'US Hercules'  
                             iv_planetype = '747-200F' ).  
  
LOOP AT gt_airplanes INTO go_airplane.  
  go_airplane->display_attributes( ).  
ENDLOOP.  
  
* long syntax for functional call:  
* CALL METHOD lcl_airplane=>get_n_o_airplanes  
*   RECEIVING  
*     rv_count = gv_count.  
  
* a little bit shorter:  
* lcl_airplane=>get_n_o_airplanes( RECEIVING rv_count = gv_count ).  
  
* the shortest syntax for functional call:  
gv_count = lcl_airplane=>get_n_o_airplanes( ).  
  
SKIP 2.  
WRITE: / 'Number of airplanes'(cal), gv_count.
```

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, raise and handle classical, non-class-based exceptions

Business Example

Make your program more realistic: The airplane objects receive their attributes as soon as they are created. Also make sure that airplane objects can only be created with plane types stored in database table SAPLANE.

Template:

SAPBC401_BAS_S3

Solution:

SAPBC401_BAS_S4

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 constructor with a suitable signature for the instances of your class LCL_AIRPLANE.
Implement it in a way that the two instance attributes are set and the instance counter N_O_AIRPLANES is increased by one.
2. Remove the definition and implementation of method SET_ATTRIBUTES to ensure that an instance's attributes can only be set once during its creation.

Task 2:

Create airplane objects.

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

Continued on next page

Task 3:

Define additional instance attributes with technical information about the airplane. Set these attributes in your constructor by accessing database table SAPLANE. Raise an exception if the specified plane type is not found on the database.

1. In your class, define the private instance attributes MV_WEIGHT (data type SAPLANE-WEIGHT) and MV_TANKCAP (data type SAPLANE_TANKCAP).
2. Extend the definition of the instance constructor by adding a classical, non-class based exception WRONG_PLANETYPE.
3. Extend the implementation of the constructor. Read a single record from database table SAPLANE for the specified plane type and set the new attributes accordingly.
Raise the exception if no data is found for the given plane type.
4. Adjust the CREATE OBJECT statements so that you handle the exception. Only store the object reference in your table if the creation was successful.
5. Follow the execution of the program in the *ABAP Debugger*. For test reasons, specify a non-existing plane type to see, how the exception handling works.

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 constructor with a suitable signature for the instances of your class LCL_AIRPLANE.

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

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

2. Remove the definition and implementation of method SET_ATTRIBUTES to ensure that an instance's attributes can only be set once during its creation.

- 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. 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 additional instance attributes with technical information about the airplane. Set these attributes in your constructor by accessing database table SAPLANE. Raise an exception if the specified plane type is not found on the database.

1. In your class, define the private instance attributes MV_WEIGHT (data type SAPLANE-WEIGHT) and MV_TANKCAP (data type SAPLANE_TANKCAP).

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

Continued on next page

2. Extend the definition of the instance constructor by adding a classical, non-class based exception `WRONG_PLANETYPE`.
 - a) See the source code extract from the model solution.
3. Extend the implementation of the constructor. Read a single record from database table `SAPLANE` for the specified plane type and set the new attributes accordingly.
Raise the exception if no data is found for the given plane type.
 - a) See the source code extract from the model solution.
4. Adjust the `CREATE OBJECT` statements so that you handle the exception. Only store the object reference in your table if the creation was successful.
 - a) See the source code extract from the model solution.
5. Follow the execution of the program in the *ABAP Debugger*. For test reasons, specify a non-existing plane type to see, how the exception handling works.
 - a) Carry out this step in the usual manner. Additional information on the ABAP Debugger is available in the SAP Library.

Result

SAPBC401_BAS_S4

```

REPORT  sapbc401_bas_s4.

TYPE-POOLS icon.

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

PUBLIC SECTION.

METHODS:
  constructor
  IMPORTING
    iv_name      TYPE string
    iv_planetype TYPE saplane-planetype
  EXCEPTIONS
    wrong_planetype,
    display_attributes.

```

Continued on next page

Internal Use SAP Partner Only

```

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

PRIVATE SECTION.

CONSTANTS:
    c_pos_1 TYPE i VALUE 30.

DATA:
    mv_name      TYPE string,
    mv_planetype TYPE saplane-planetype,
    mv_weight    TYPE saplane-weight,
    mv_tankcap   TYPE saplane-tankcap.

CLASS-DATA:
    gv_n_o_airplanes TYPE i.

ENDCLASS.          "lcl_airplane DEFINITION

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

METHOD constructor.

DATA: ls_planetype TYPE saplane.

mv_name      = iv_name.
mv_planetype = iv_planetype.

SELECT SINGLE * FROM saplane
    INTO ls_planetype
    WHERE planetype = iv_planetype.

IF sy-subrc <> 0.
    RAISE wrong_planetype.
ELSE.
    mv_weight  = ls_planetype-weight.
    mv_tankcap = ls_planetype-tankcap.
    gv_n_o_airplanes = gv_n_o_airplanes + 1.
ENDIF.

```

Continued on next page

```

ENDMETHOD.          "constructor

METHOD display_attributes.

WRITE:
/ icon_ws_plane AS ICON,
/ 'Name of Airplane'(001), AT c_pos_1 mv_name,
/ 'Type of Airplane:'(002), AT c_pos_1 mv_planetype,
/ 'Weight:'(003),           AT c_pos_1 mv_weight LEFT-JUSTIFIED,
/ 'Tank capacity:'(004),    AT c_pos_1 mv_tankcap LEFT-JUSTIFIED.

ENDMETHOD.          "display_attributes

METHOD display_n_o_airplanes.

SKIP.

WRITE:
/ 'Number of airplanes:'(cal),
AT c_pos_1 gv_n_o_airplanes LEFT-JUSTIFIED.
ENDMETHOD.          "display_n_o_airplanes

METHOD get_n_o_airplanes.
rv_count = gv_n_o_airplanes.
ENDMETHOD.          "get_n_o_airplanes
ENDCLASS.          "lcl_airplane IMPLEMENTATION

DATA:
go_airplane  TYPE REF TO lcl_airplane,
gt_airplanes TYPE TABLE OF REF TO lcl_airplane,
gv_count     TYPE i.

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

CREATE OBJECT go_airplane
EXPORTING
  iv_name      = 'LH Berlin'
  iv_planetype = 'A321'
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc = 0.
  APPEND go_airplane TO gt_airplanes.
ENDIF.

```

Continued on next page

```
CREATE OBJECT go_airplane
  EXPORTING
    iv_name      = 'AA New York'
    iv_planetype = '747-400'
  EXCEPTIONS
    wrong_planetype = 1.
  IF sy-subrc = 0.
    APPEND go_airplane TO gt_airplanes.
  ENDIF.

CREATE OBJECT go_airplane
  EXPORTING
    iv_name      = 'US Hercules'
    iv_planetype = '747-200F'
  EXCEPTIONS
    wrong_planetype = 1.
  IF sy-subrc = 0.
    APPEND go_airplane TO gt_airplanes.
  ENDIF.

LOOP AT gt_airplanes INTO go_airplane.
  go_airplane->display_attributes( ).
ENDLOOP.

gv_count = lcl_airplane=>get_n_o_airplanes( ).

SKIP 2.
WRITE: / 'Number of airplanes'(ca1), gv_count.
```


Exercise 6: Private Methods

Exercise Objectives

After completing this exercise, you will be able to:

- Define and implement static constructors
- Use the delegation principle
- Structure and call private methods

Business Example

For performance reasons, you do not want to access the database each time an instance is created. Therefore, you read all the data just once in the static constructor and store it in a static attribute.

Template:

SAPBC401_BAS_S4

Solution:

SAPBC401_BAS_S5

Task 1:

Define an internal table as a private static attribute. Define a static constructor in which you fill the attribute with the complete content of database table SAPLANE.

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

Define the internal table GT_PLANETYPES as a private class attribute.

To specify the type for this internal table, define the private table type TY_PLANETYPES in the class definition.

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

2. Define a static constructor in the LCL_AIRPLANE class.

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

Continued on next page

3. Use the *ABAP Debugger* to make sure 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?

Task 2:

Declare and implement a private static method to derive the weight and tank capacity of one plane type from the GT_PLANETYPES table.

1. 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.

The signature should also contain a classical, non-class based exception which is raised if no data is found for the specified airplane type.

2. Implement the method in such a way that the values for the export parameters are determined through a single-record access to the internal table GT_PLANETYPES.

If the table does not contain any values for the airplane type, the method should raise an exception.

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

Continued on next page

Task 3:

Replace the database access in the **instance** constructor by a call of method GET_TECHNICAL_ATTRIBUTES.

1. Is it possible to call the new method from the main program? Why not?

2. Call the method GET_TECHNICAL_ATTRIBUTES from the constructor to obtain additional technical data. Depending on the outcome, set the corresponding private attributes or raise the exception of the constructor.
3. Observe the execution of the program in the *ABAP Debugger*.
4. Which alternative solutions could be used to solve the tasks?

Solution 6: Private Methods

Task 1:

Define an internal table as a private static attribute. Define a static constructor in which you fill the attribute with the complete content of database table SAPLANE.

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

Define the internal table GT_PLANETYPES as a private class attribute.

To specify the type for this internal table, define the private table type TY_PLANETYPES in the class definition.

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

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

2. Define a static constructor in the LCL_AIRPLANE class.

Implement the constructor in such a way that the GT_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. Use the *ABAP Debugger* to make sure 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?

Answer: Before the class is accessed for the first time (before DISPLAY_N_O_AIRPLANES in this case).

Task 2:

Declare and implement a private static method to derive the weight and tank capacity of one plane type from the GT_PLANETYPES table.

1. 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.

The signature should also contain a classical, non-class based exception which is raised if no data is found for the specified airplane type.

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

Continued on next page

2. Implement the method in such a way that the values for the export parameters are determined through a single-record access to the internal table GT_PLANETYPES.

If the table does not contain any values for the airplane type, the method should raise an exception.

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

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

Task 3:

Replace the database access in the **instance** constructor by a call of method GET_TECHNICAL_ATTRIBUTES.

1. Is it possible to call the new method from the main program? Why not?

Answer: This would only be possible if the method was public. As it is a private method, it can only be called from within the LCL_AIRPLANE class.

2. Call the method GET_TECHNICAL_ATTRIBUTES from the constructor to obtain additional technical data. Depending on the outcome, set the corresponding private attributes or raise the exception of the constructor.

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

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:

- **Define GET_TECHNICAL_ATTRIBUTES as instance method without the PLANETYPE import parameter.**

If the method was defined as an instance method, it could access the MV_PLANETYPE attribute directly (as ME->MV_PLANETYPE). In that case the import parameter would not be needed.

- **Do not use a method to read GT_PLANETYPES**

In the instance constructor you could access the class attribute GT_PLANETYPES directly (as ME->GT_PLANETYPES).

Continued on next page

Result

SAPBC401_BAS_S5

```

REPORT  sapbc401_bas_s5.

TYPE-POOLS icon.

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

PUBLIC SECTION.

METHODS:
constructor
IMPORTING
    iv_name      TYPE string
    iv_planetype TYPE saplane-planetype
EXCEPTIONS
    wrong_planetype,
display_attributes.

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

PRIVATE SECTION.

TYPES:
ty_planetypes TYPE STANDARD TABLE OF saplane
    WITH NON-UNIQUE KEY planetype.

CONSTANTS:
c_pos_1 TYPE i VALUE 30.

DATA:
mv_name      TYPE string,
mv_planetype TYPE saplane-planetype,
mv_weight    TYPE saplane-weight,

```

Continued on next page

```

mv_tankcap    TYPE saplane-tankcap.

CLASS-DATA:
  gv_n_o_airplanes TYPE i,
  gt_planetypes    TYPE ty_planetypes.

CLASS-METHODS:
  get_technical_attributes
    IMPORTING
      iv_type      TYPE saplane-planetype
    EXPORTING
      ev_weight    TYPE saplane-weight
      ev_tankcap   TYPE saplane-tankcap
    EXCEPTIONS
      wrong_planetype.

ENDCLASS.          "lcl_airplane DEFINITION

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

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

METHOD constructor.

*  DATA: ls_planetype TYPE saplane.

  mv_name      = iv_name.
  mv_planetype = iv_planetype.

*  SELECT SINGLE * FROM saplane
*    INTO ls_planetype
*    WHERE planetype = iv_planetype.

get_technical_attributes(
  EXPORTING
    iv_type = iv_planetype
  IMPORTING
    ev_weight  = mv_weight
    ev_tankcap = mv_tankcap
  EXCEPTIONS

```

Continued on next page

```

wrong_planetype = 1 ).

IF sy-subrc <> 0.
  RAISE wrong_planetype.
ELSE.
*   mv_weight = ls_planetype-weight.
*   mv_tankcap = ls_planetype-tankcap.
  gv_n_o_airplanes = gv_n_o_airplanes + 1.
ENDIF.

ENDMETHOD.           "constructor

METHOD display_attributes.

WRITE:
  / icon_ws_plane AS ICON,
  / 'Name of Airplane'(001), AT c_pos_1 mv_name,
  / 'Type of Airplane:'(002), AT c_pos_1 mv_planetype,
  / 'Weight:'(003),          AT c_pos_1 mv_weight LEFT-JUSTIFIED,
  / 'Tank capacity:'(004),   AT c_pos_1 mv_tankcap LEFT-JUSTIFIED.

ENDMETHOD.           "display_attributes

METHOD display_n_o_airplanes.
SKIP.
WRITE:
  / 'Number of airplanes:'(cal),
  AT c_pos_1 gv_n_o_airplanes LEFT-JUSTIFIED.
ENDMETHOD.           "display_n_o_airplanes

METHOD get_n_o_airplanes.
rv_count = gv_n_o_airplanes.
ENDMETHOD.           "get_n_o_airplanes

METHOD get_technical_attributes.
DATA: ls_planetype TYPE saplane.

READ TABLE gt_planetypes INTO ls_planetype
      WITH TABLE KEY planetype = iv_type
      TRANSPORTING weight tankcap.

IF sy-subrc = 0.
  ev_weight = ls_planetype-weight.
  ev_tankcap = ls_planetype-tankcap.
ELSE.
  RAISE wrong_planetype.
ENDIF.

```

Continued on next page

```

ENDIF.

ENDMETHOD.          "get_technical_attributes

ENDCLASS.           "lcl_airplane IMPLEMENTATION

DATA:
go_airplane  TYPE REF TO lcl_airplane,
gt_airplanes TYPE TABLE OF REF TO lcl_airplane,
gv_count      TYPE i.

START-OF-SELECTION.
*****



lcl_airplane=>display_n_o_airplanes( ).

CREATE OBJECT go_airplane
EXPORTING
  iv_name      = 'LH Berlin'
  iv_planetype = 'A321'
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc = 0.
  APPEND go_airplane TO gt_airplanes.
ENDIF.

CREATE OBJECT go_airplane
EXPORTING
  iv_name      = 'AA New York'
  iv_planetype = '747-400'
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc = 0.
  APPEND go_airplane TO gt_airplanes.
ENDIF.

CREATE OBJECT go_airplane
EXPORTING
  iv_name      = 'US Hercules'
  iv_planetype = '747-200F'
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc = 0.
  APPEND go_airplane TO gt_airplanes.
ENDIF.

```

Continued on next page

```
LOOP AT gt_airplanes INTO go_airplane.  
  go_airplane->display_attributes( ).  
ENDLOOP.  
  
gv_count = lcl_airplane=>get_n_o_airplanes( ).  
  
SKIP 2.  
WRITE: / 'Number of airplanes'(ca1), gv_count.
```



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. Do you always have to define a constructor, too?



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. Do you always have to define a constructor, too?

Answer: No.

Refer to the relevant section of the lesson.

Internal Use SAP Partner Only

International Use SAP Partner Only

Internal Use SAP Partner Only

International Use SAP Partner Only

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	135
Exercise 9: Aggregation and Generic Calls.....	141
Lesson: Interfaces and Casting	153
Exercise 10: Interface Definition und Implementation	169
Exercise 11: Use of Interfaces	179
Exercise 12: Integrate Trainer's Object Model	187
Lesson: Events.....	198
Exercise 13: Events in Superclasses	207
Exercise 14: (Optional) Events in Interfaces	223

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.

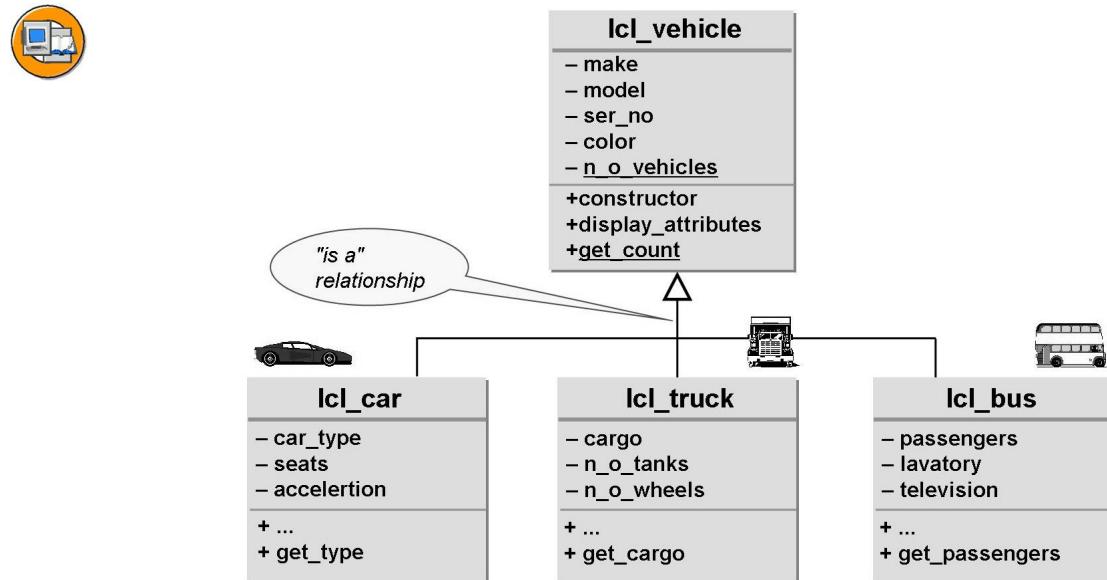


Figure 63: Example of Generalization/Specialization

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**.

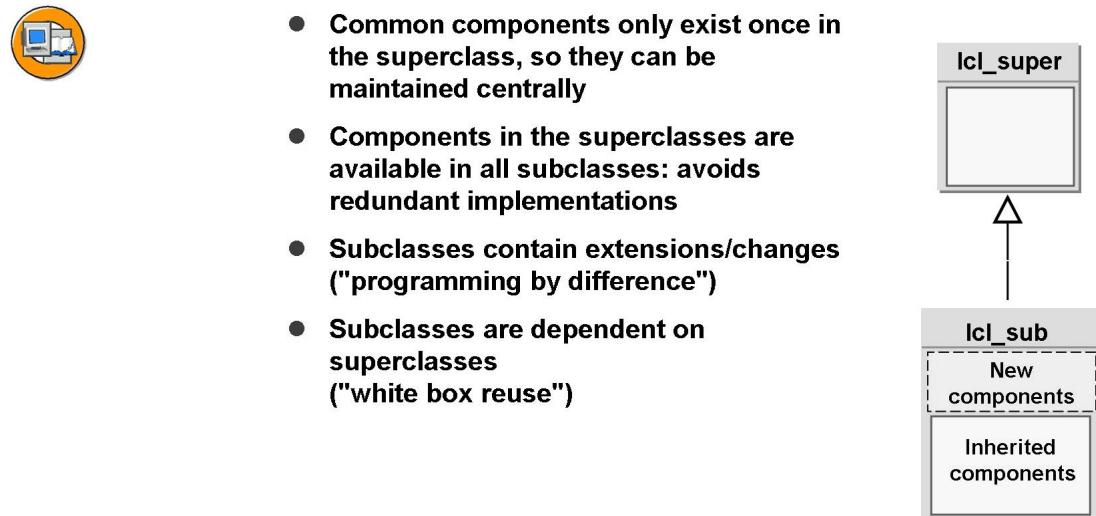


Figure 64: Characteristics of Generalization/Specialization

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 set_type IMPORTING iv_make      TYPE string.

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



```
CLASS lcl_truck DEFINITION INHERITING FROM lcl_vehicle.

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

PRIVATE SECTION.
  DATA mv_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 SET_TYPE method. The subclass also defines the GET_CARGO method.



```
CLASS lcl_vehicle DEFINITION.
  PUBLIC SECTION.
    METHODS display_attributes.

ENDCLASS.
```

```
CLASS lcl_truck DEFINITION INHERITING FROM lcl_vehicle.
  PUBLIC SECTION.
    METHODS display_attributes REDEFINITION.
  ...
ENDCLASS.
```



Signature must be retained!

```
CLASS lcl_truck IMPLEMENTATION.
  METHOD display_attributes.
  ...
  super->display_attributes(...)

ENDMETHOD.
ENDCLASS.
```

New implementation of inherited method.

Refers to the superclass component.

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. Since 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.

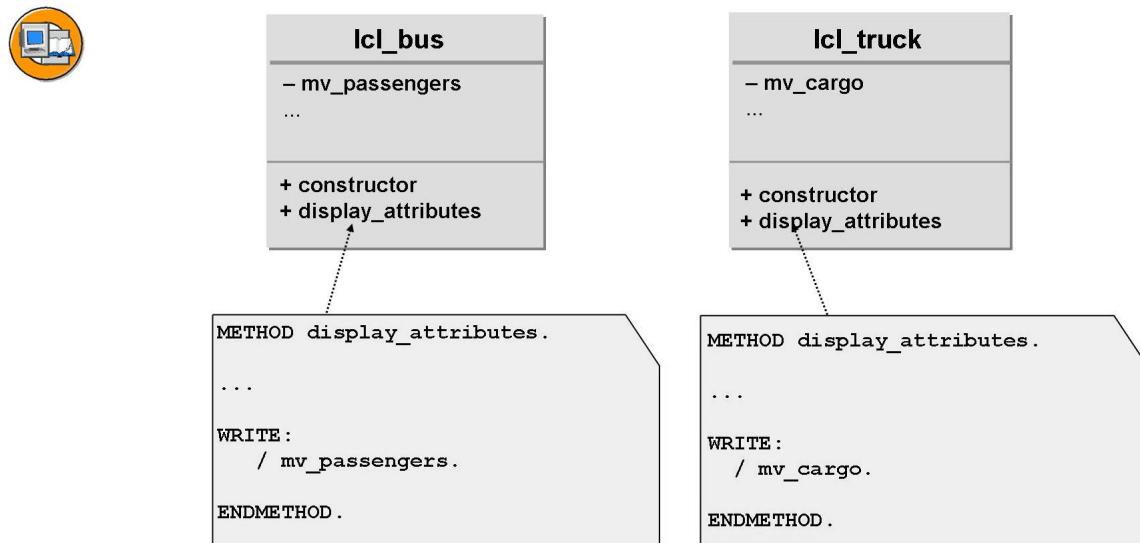


Figure 67: Preserving Semantics During Redefinition

In this example, both redefined methods output completely different information. The important point is that the method's semantics stay the same.

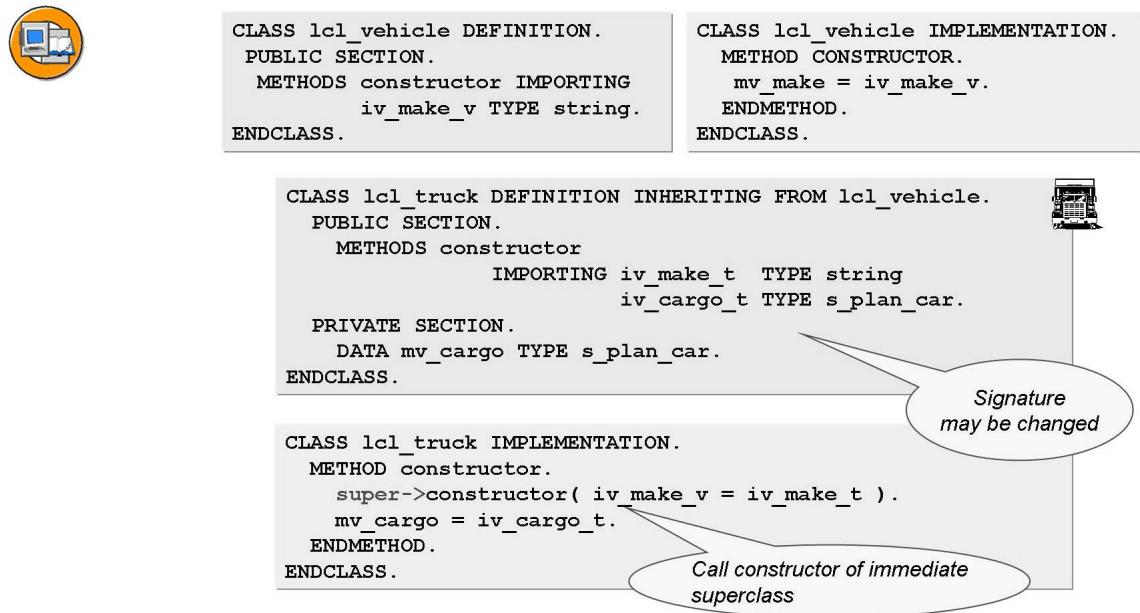


Figure 68: Definition of the constructor in subclasses

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 is not inherited like normal methods. Any class can define its own constructor – fully independent from the definition of the constructor in its superclass. A subclass can even define a constructor if there's no constructor in the superclass.

However, when **implementing** the subclass constructor it is **mandatory** to call the constructor of the immediate superclass. Through this it is ensured that the constructor of class is always executed - whether the created object is an instance of the class itself or an instance of one of its subclasses.

Because of this enforced call of the superclass constructor, the subclass constructor will normally just add parameters to the signature of the superclass constructor rather than be completely different.



Hint: In this context, the concept of **overloading** is of relevance. With overloading, a method can have several definitions with different signatures and thus also different implementations. The concept of overloading is **not supported in ABAP Objects**. The usual workaround is one signature with different sets of optional parameters.

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: go_2 TYPE REF TO lcl_2,
      go_3 TYPE REF TO lcl_3.

* case 2:
CREATE OBJECT go_2 EXPORTING iv_a1 = 100.

* case 1:
CREATE OBJECT go_3 EXPORTING iv_a1 = 100
                  iv_a2 = 1000.
```

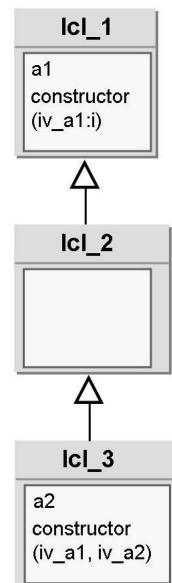


Figure 69: Rules for Calling the Constructor

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.

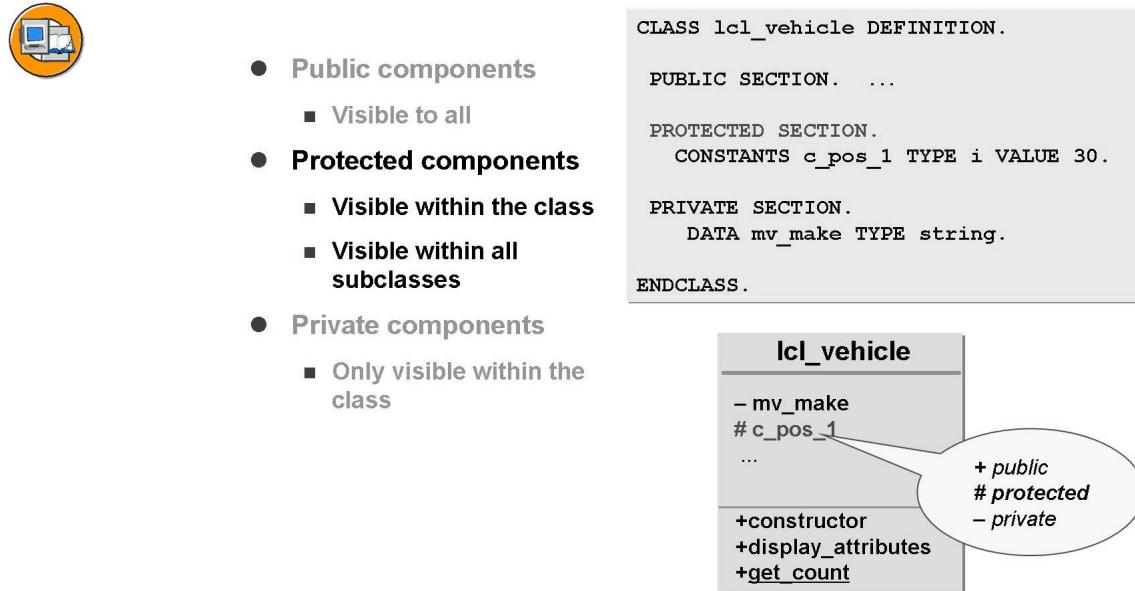


Figure 70: Inheritance and Visibility

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.

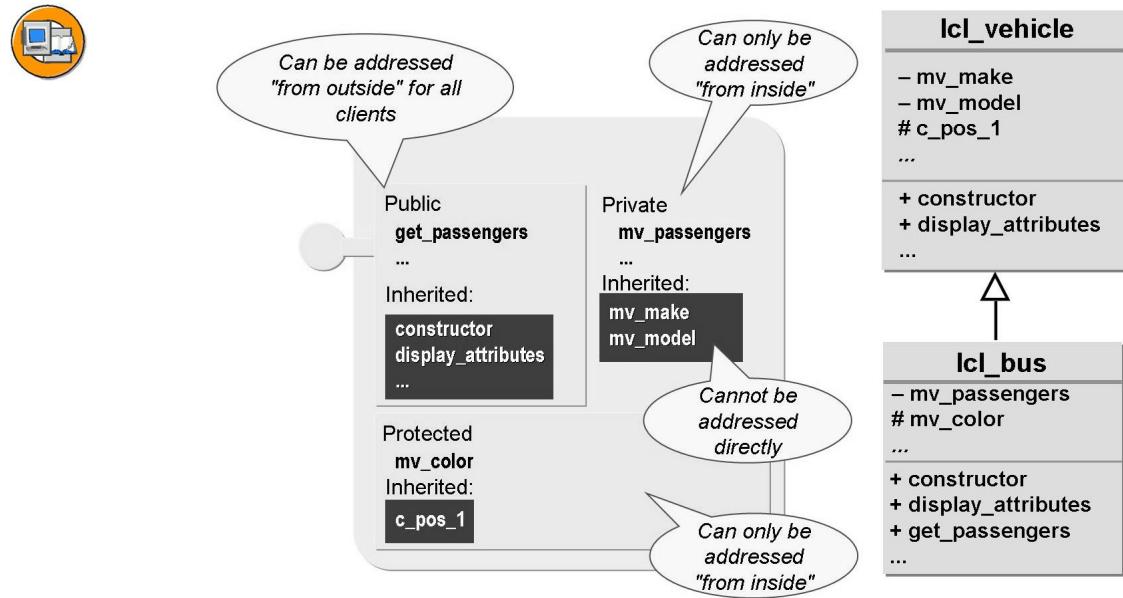


Figure 71: Protected Versus Private Section

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 constant `C_POS_1` in superclass `LCL_VEHICLE` directly from its subclass `LCL_BUS`. On the other hand, the only way to access the private `MV_MAKE` and `MV_MODEL` attributes from the subclasses of `LCL_VEHICLE` is to call methods of `LCL_VEHICLE`. These methods have to be public or protected.

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.
- The static constructor of a superclass is executed when the superclass or one of its subclasses is accessed for the first time.
- A subclass can always have a static constructor – whether the superclass has a static constructor or not.
- If a subclass and its superclass have a static constructor they are both executed when the first access to the subclass is also the first access to the superclass.

Up-Cast (Widening Cast)

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



```
DATA: go_vehicle TYPE REF TO lcl_vehicle,
      go_truck     TYPE REF TO lcl_truck.
```

```
CREATE OBJECT go_truck.
```

`go_truck`

`go_vehicle`

(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.

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



Hint: With redefined methods 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 GO_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 GO_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. Since 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 go_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

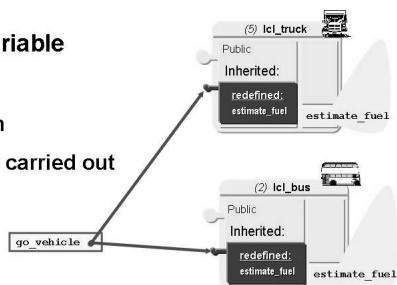


Figure 73: Static and Dynamic Types of References

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

In the example, LCL_VEHICLE is the static type of the variable GO_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.

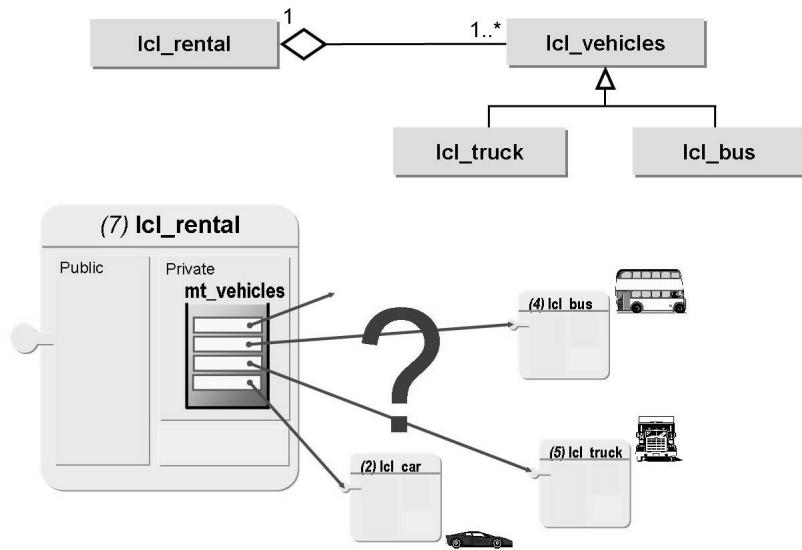


Figure 74: Generic Access After Up-Cast Assignments

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 `DISPLAY_ATTRIBUTES` method is defined in the superclass `LCL_VEHICLE` and is redefined in all subclasses.

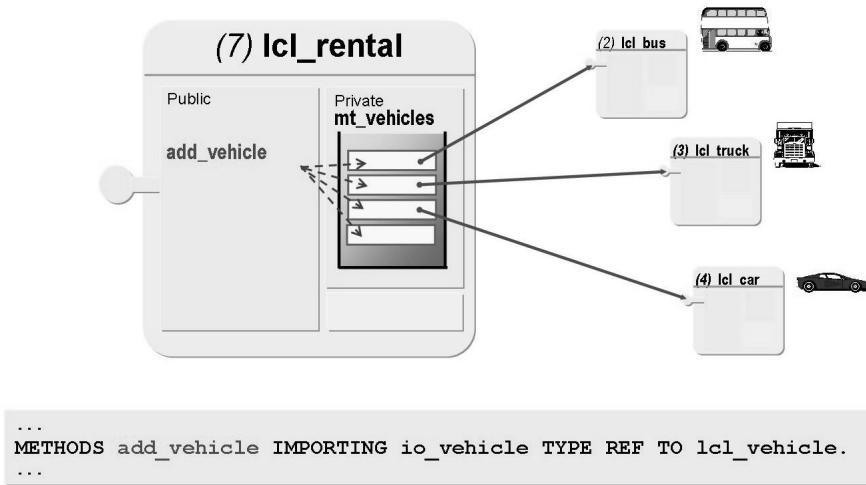


Figure 75: Row Type of the Internal Table in the Application Example

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.



```

CLASS lcl_rental DEFINITION.
  PUBLIC SECTION.
    METHODS add_vehicle IMPORTING io_vehicle TYPE REF TO lcl_vehicle.
    METHODS display_attributes.

  PROTECTED SECTION.
    DATA mt_vehicles TYPE TABLE OF REF TO lcl_vehicle.
  ENDCCLASS.

  CLASS lcl_rental IMPLEMENTATION.
    METHOD add_vehicle.
      APPEND io_vehicle TO mt_vehicles.
    ENDMETHOD.

    METHOD display_attributes.
      DATA lo_vehicle TYPE REF TO lcl_vehicle.
      LOOP AT mt_vehicles INTO lo_vehicle.
        lo_vehicle->display_attributes( ).
      ENDLOOP.
    ENDMETHOD.
  ENDCCLASS.
  
```

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 DISPLAY_ATTRIBUTES method was inherited from the LCL_VEHICLE superclass and may have been redefined.

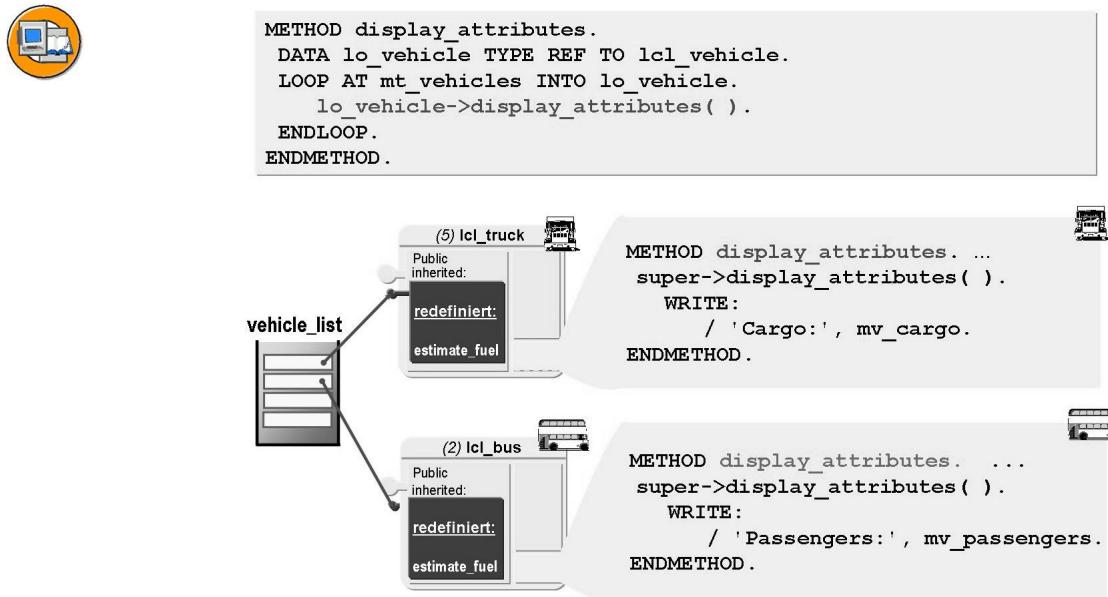


Figure 77: Polymorphism: Generic Access Using the Superclass Reference

Which implementation is executed when DISPLAY_ATTRIBUTES is called now depends on which object the superclass reference GO_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 go_vehicle->display_attributes is called, the implementation is not executed from LCL_VEHICLE (static type of GO_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 DISPLAY_ATTRIBUTES. 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 display_truck.
...
ENDFUNCTION.

FUNCTION display_bus.
...
ENDFUNCTION.

...
DATA: gt_vehicles  TYPE TABLE OF vehicle_type,
      gs_vehicle   LIKE LINE OF vehicle_list.

gt_vehicles
name ... func_name
truck1 ... DISPLAY_TRUCK
bus1 ... DISPLAY_BUS
bus2 ... DISPLAY_BUS
truck2 ... DISPLAY_TRUCK
truck3 ... DISPLAY_TRUCK
      ...
LOOP AT gt_vehicles INTO gs_vehicle.
  CALL FUNCTION gs_vehicle-func_name.
ENDLOOP.
...

```

Figure 78: Generic Calls in the Procedural Programming Model

Using dynamic function module calls, you can program generically in ABAP, 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 model 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”.

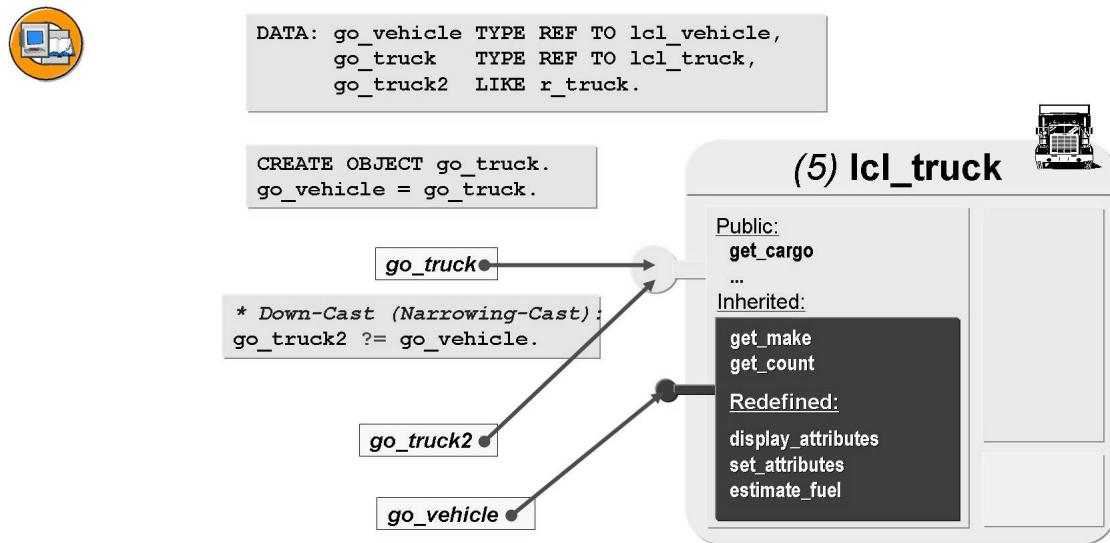
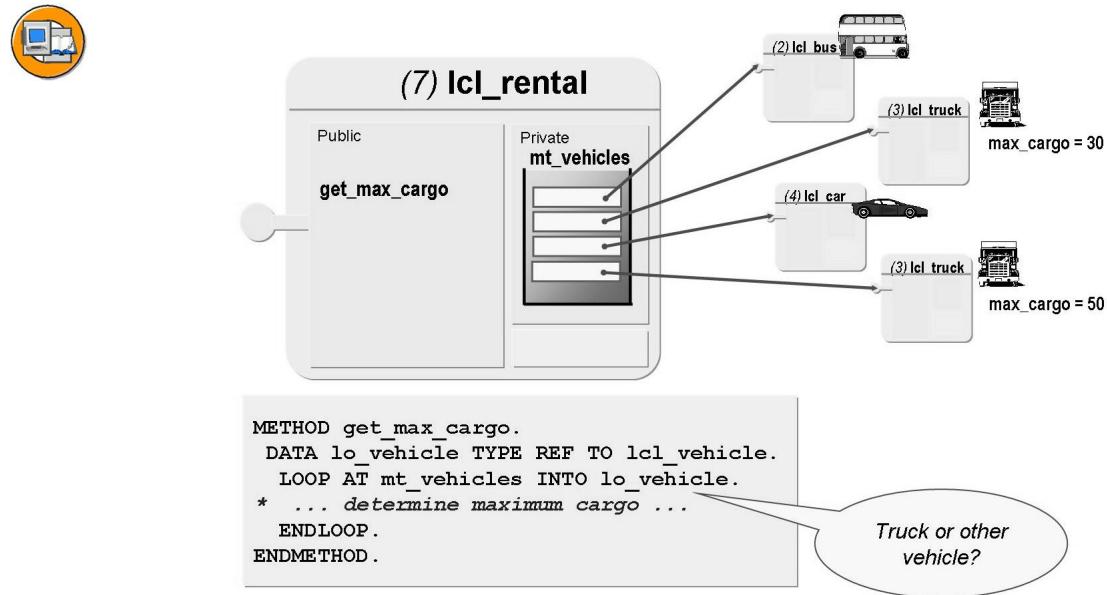


Figure 79: Down Cast (Narrowing Cast) with Object References

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 GO_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 GO_VEHICLE at runtime but the down cast assignment operator tries to copy the reference to the now invalid reference GO_TRUCK?

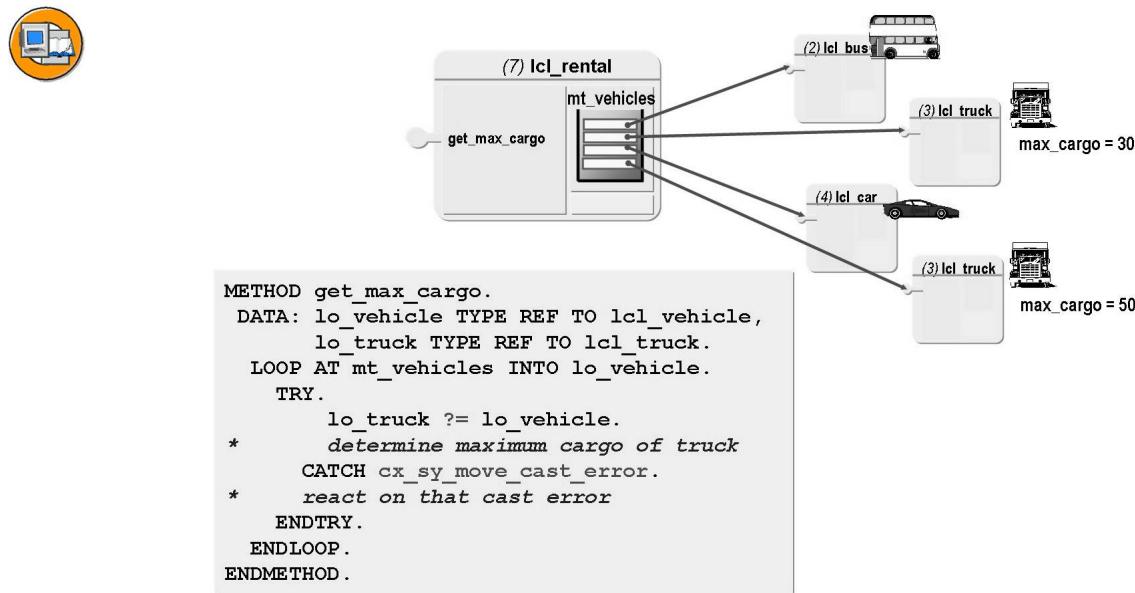


Figure 81: Down Cast and Exception Handling in the Application Example

In contrast to the up cast assignment, it is possible that the static type of the target variable (GO_TRUCK) is neither more general than nor the same as the dynamic type of the source variable (GO_VEHICLE), namely when GO_VEHICLE contains bus or sports car references. That is the reason 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: Usage 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

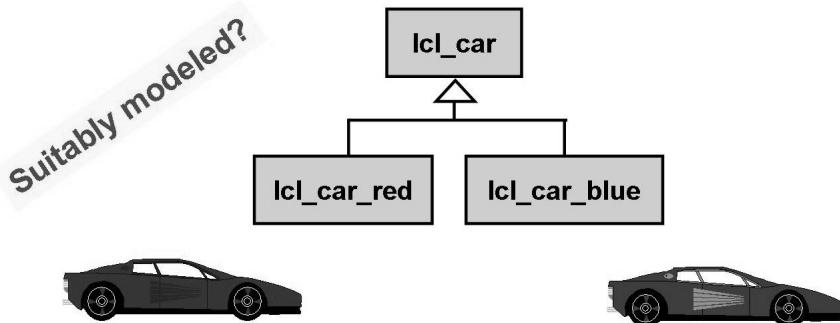


Figure 83: Examples: Misuse of Inheritance

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
- Define Instance constructors in subclasses
- Effectively use visibility sections

Business Example

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

Template:

SAPBC401_BAS_S5

Solution:

SAPBC401_INH_S1

Task 1:

Define the local class LCL_PASSENGER_PLANE for passenger planes. Define it as a subclass of LCL_AIRPLANE. Add specific attributes and methods and redefine the DISPLAY_ATTRIBUTES method.

1. Complete your program ZBC401_##_MAIN (where ## is your two-digit group number) or copy the template program.
Define a new local class LCL_PASSENGER_PLANE and make it a subclass of the already existing LCL_AIRPLANE.
2. Is there a specific sequence in which the classes have to be defined in the source code?

3. The class must have a private instance attribute, MV_SEATS, with the same type as the table field SFLIGHT-SEATSMAX.

Continued on next page

4. Define and implement an instance constructor that assigns values to **all** instance attributes in the class.
5. Redefine method DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.

Make sure the output of specific attributes is aligned with the output from the superclass. Therefore, change the visibility of the C_POS_1 constant so that you can use it in your subclass.

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, MV_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.
3. Redefine DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.

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. Before instantiating any objects, call the static method DISPLAY_N_O_AIRPLANES.
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:

Debug your program.

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

Continued on next page

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:

Define the local class LCL_PASSENGER_PLANE for passenger planes. Define it as a subclass of LCL_AIRPLANE. Add specific attributes and methods and redefine the DISPLAY_ATTRIBUTES method.

1. Complete your program ZBC401##_MAIN (where ## is your two-digit group number) or copy the template program.
Define a new local class LCL_PASSENGER_PLANE and make it a subclass of the already existing LCL_AIRPLANE.

- a) See the source code extract from the model solution.
2. Is there a specific sequence in which the classes have to be defined in the source code?

Answer: The superclass has to be defined **first**. Otherwise the syntax check will not know it when we refer to it in the INHERITING FROM part of the subclass definition.

3. The class must have a private instance attribute, MV_SEATS, with the same type as the table field SFLIGHT-SEATSMAX.
 - a) See the source code extract from the model solution.
4. Define and implement an instance constructor that assigns values to **all** instance attributes in the class.
 - a) See the source code extract from the model solution.
5. Redefine method DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.

Make sure the output of specific attributes is aligned with the output from the superclass. Therefore, change the visibility of the C_POS_1 constant so that you can use it in your subclass.

- a) See the source code extract from 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, MV_CARGO, with the same type as the table field SCPLANE-CARGOMAX.
 - a) See the source code extract from the model solution.

Continued on next page

2. Define and implement an instance constructor that assigns values to **all** instance attributes in the class.
 - a) See the source code extract from the model solution.
3. Redefine DISPLAY_ATTRIBUTES so that **all** instance attributes are displayed using the WRITE statement.
 - a) See the source code extract from 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) See the source code extract from the model solution.
2. Before instantiating any objects, call the static method DISPLAY_N_O_AIRPLANES.
 - a) See the source code extract from 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) See the source code extract from the model solution.
4. Call the DISPLAY_ATTRIBUTES method for both instances.
 - a) See the source code extract from the model solution.
5. Call the static method DISPLAY_ATTRIBUTES a second time.
 - a) See the source code extract from the model solution.

Task 4:

Debug 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 method is private in the superclass.

Continued on next page

Result

SAPBC401_INH_S1

```

REPORT  sapbc401_inh_s1.

TYPE-POOLS icon.

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

PUBLIC SECTION.

METHODS:
constructor
IMPORTING
    iv_name      TYPE string
    iv_planetype TYPE saplane-planetype
EXCEPTIONS
    wrong_planetype,
display_attributes.

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

PROTECTED SECTION.

CONSTANTS:
c_pos_1 TYPE i VALUE 30.

PRIVATE SECTION.

TYPES:
ty_planetypes TYPE STANDARD TABLE OF saplane
    WITH NON-UNIQUE KEY planetype.

DATA:
mv_name      TYPE string,

```

Continued on next page

```

mv_planetype TYPE saplane-planetype,
mv_weight      TYPE saplane-weight,
mv_tankcap     TYPE saplane-tankcap.

CLASS-DATA:
gv_n_o_airplanes TYPE i,
gt_planetypes    TYPE ty_planetypes.

CLASS-METHODS:
get_technical_attributes
  IMPORTING
    iv_type      TYPE saplane-planetype
  EXPORTING
    ev_weight    TYPE saplane-weight
    ev_tankcap   TYPE saplane-tankcap
  EXCEPTIONS
    wrong_planetype.

ENDCLASS.          "lcl_airplane DEFINITION

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

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

METHOD constructor.

  mv_name      = iv_name.
  mv_planetype = iv_planetype.

  get_technical_attributes(
    EXPORTING
      iv_type = iv_planetype
    IMPORTING
      ev_weight  = mv_weight
      ev_tankcap = mv_tankcap
    EXCEPTIONS
      wrong_planetype = 1 ).

  IF sy-subrc <> 0.
    RAISE wrong_planetype.

```

Continued on next page

```

ELSE.
  gv_n_o_airplanes = gv_n_o_airplanes + 1.
ENDIF.

ENDMETHOD.           "constructor

METHOD display_attributes.

WRITE:
  / icon_ws_plane AS ICON,
  / 'Name of Airplane'(001), AT c_pos_1 mv_name,
  / 'Type of Airplane:'(002), AT c_pos_1 mv_planetype,
  / 'Weight:'(003),          AT c_pos_1 mv_weight LEFT-JUSTIFIED,
  / 'Tank capacity:'(004),   AT c_pos_1 mv_tankcap LEFT-JUSTIFIED.

ENDMETHOD.           "display_attributes

METHOD display_n_o_airplanes.

SKIP.
WRITE:
  / 'Number of airplanes:'(cal),
  AT c_pos_1 gv_n_o_airplanes LEFT-JUSTIFIED.
ENDMETHOD.           "display_n_o_airplanes

METHOD get_n_o_airplanes.
  rv_count = gv_n_o_airplanes.
ENDMETHOD.           "get_n_o_airplanes

METHOD get_technical_attributes.
  DATA: ls_planetype TYPE saplane.

  READ TABLE gt_planetypes INTO ls_planetype
    WITH TABLE KEY planetype = iv_type
    TRANSPORTING weight tankcap.
  IF sy-subrc = 0.
    ev_weight = ls_planetype-weight.
    ev_tankcap = ls_planetype-tankcap.
  ELSE.
    RAISE wrong_planetype.
  ENDIF.
ENDMETHOD.           "get_technical_attributes

ENDCLASS.           "lcl_airplane IMPLEMENTATION

```

Continued on next page

```

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

PUBLIC SECTION.

METHODS:
constructor
IMPORTING
  iv_name      TYPE string
  iv_planetype TYPE saplane-planetype
  iv_cargo     TYPE s_plan_car
EXCEPTIONS
  wrong_planetype,
display_attributes REDEFINITION.

PRIVATE SECTION.

DATA:
  mv_cargo TYPE s_plan_car.

ENDCLASS.          "lcl_cargo_plane DEFINITION

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

METHOD constructor.
super->constructor(
EXPORTING
  iv_name      = iv_name
  iv_planetype = iv_planetype
EXCEPTIONS
  wrong_planetype = 1 .
IF sy-subrc <> 0.
  RAISE wrong_planetype.
ENDIF.
mv_cargo = iv_cargo.
ENDMETHOD.          "constructor

```

Continued on next page

```

METHOD display_attributes.
super->display_attributes( ).
WRITE:
/ 'Max Cargo:' (005), AT c_pos_1 mv_cargo LEFT-JUSTIFIED.
ULINE.
ENDMETHOD.           "display_attributes

ENDCLASS.           "lcl_cargo_plane IMPLEMENTATION

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

PUBLIC SECTION.

METHODS:
constructor
IMPORTING
iv_name      TYPE string
iv_planetype TYPE saplane-planetype
iv_seats     TYPE s_seatsmax
EXCEPTIONS
wrong_planetype,
display_attributes REDEFINITION.

PRIVATE SECTION.

DATA:
mv_seats TYPE s_seatsmax.
ENDCLASS.           "lcl_passenger_plane DEFINITION

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

METHOD constructor.
super->constructor(
EXPORTING
iv_name      = iv_name
iv_planetype = iv_planetype
EXCEPTIONS
wrong_planetype = 1 ).


```

Continued on next page

```

IF sy-subrc <> 0.
  RAISE wrong_planetype.
ENDIF.
mv_seats = iv_seats.
ENDMETHOD.           "constructor

METHOD display_attributes.
  super->display_attributes( ).
  WRITE:
    / 'Max Seats:' (006), AT c_pos_1 mv_seats LEFT-JUSTIFIED.
  ULINE.
ENDMETHOD.           "display_attributes
ENDCLASS.            "lcl_passenger_plane IMPLEMENTATION

DATA:
  go_airplane  TYPE REF TO lcl_airplane,
  go_cargo      TYPE REF TO lcl_cargo_plane,
  go_passenger  TYPE REF TO lcl_passenger_plane,
  gt_airplanes  TYPE TABLE OF REF TO lcl_airplane,
  gv_count      TYPE i.

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

CREATE OBJECT go_passenger
  EXPORTING
    iv_name        = 'LH BERLIN'
    iv_planetype   = '747-400'
    iv_seats       = 345
  EXCEPTIONS
    wrong_planetype = 1.

  IF sy-subrc = 0.
* do nothing just now
  ENDIF.

CREATE OBJECT go_cargo
  EXPORTING
    iv_name        = 'US Hercules'
    iv_planetype   = '747-200F'
    iv_cargo       = 533
  EXCEPTIONS
    wrong_planetype = 1.

```

Continued on next page

```
IF sy-subrc = 0.  
* do nothing just now  
ENDIF.  
  
go_passenger->display_attributes( ).  
go_cargo->display_attributes( ).  
  
gv_count = lcl_airplane=>get_n_o_airplanes( ).  
  
SKIP 2.  
WRITE: / 'Number of airplanes'(cal), gv_count.
```

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.

Template:

SAPBC401_INH_S1

Solution:

SAPBC401_INH_S2

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.
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

SAPBC401_INH_S2

```
REPORT  sapbc401_inh_s2.

TYPE-POOLS icon.

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

...
ENDCLASS.          "lcl_airplane DEFINITION

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

...
ENDCLASS.          "lcl_airplane IMPLEMENTATION

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

...
ENDCLASS.          "lcl_cargo_plane DEFINITION

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

...

```

Continued on next page

Internal Use SAP Partner Only

Internal Use SAP Partner Only

```

ENDCLASS.           "lcl_cargo_plane IMPLEMENTATION

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

...

ENDCLASS.           "lcl_passenger_plane DEFINITION

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

...

ENDCLASS.           "lcl_passenger_plane IMPLEMENTATION

DATA:
go_airplane  TYPE REF TO lcl_airplane,
go_cargo      TYPE REF TO lcl_cargo_plane,
go_passenger  TYPE REF TO lcl_passenger_plane,
gt_airplanes  TYPE TABLE OF REF TO lcl_airplane,
gv_count      TYPE i.

START-OF-SELECTION.
*****



lcl_airplane=>display_n_o_airplanes( ).

CREATE OBJECT go_passenger
EXPORTING
    iv_name      = 'LH BERLIN'
    iv_planetype = '747-400'
    iv_seats     = 345
EXCEPTIONS
    wrong_planetype = 1.

IF sy-subrc = 0.
    APPEND go_passenger TO gt_airplanes.
ENDIF.

CREATE OBJECT go_cargo

```

Continued on next page

```
EXPORTING
  iv_name      = 'US Hercules'
  iv_planetype = '747-200F'
  iv_cargo     = 533
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc = 0.
  APPEND go_cargo TO gt_airplanes.
ENDIF.

LOOP AT gt_airplanes INTO go_airplane.
  go_airplane->display_attributes( ).
ENDLOOP.

gv_count = lcl_airplane=>get_n_o_airplanes( ).

SKIP 2.
WRITE: / 'Number of airplanes'(ca1), gv_count.
```

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.

Template:

SAPBC401_INH_T3

Solution:

SAPBC401_INH_S3 (Task 1 and 2)

SAPBC401_INH_S3_OPT (Task 3)

Task 1:

Define a local class for airlines.

1. Complete your program ZBC401_##_MAIN (where ## is your two-digit group number) or copy the template program.

If you copy the template program go directly to the next step. Otherwise copy local class LCL_CARRIER (definition and implementation) from the template program and paste it into your own program.

2. Define a private instance attribute MT_AIRPLANES to store references to airplanes there.
3. Define and implement a public method ADD_AIRPLANE so that airplane references can be added to the previously defined list MT_AIRPLANES. The method should have one import parameter IO_PLANE.
4. Define and implement a private instance method DISPLAY_AIRPLANES. In this method, loop at the list of airplanes and call method DISPLAY_ATTRIBUTES for each airplane.
5. Extend the implementation of the method DISPLAY_ATTRIBUTES in a way that it not only displays information about the carrier itself but also about all its airplanes. Use the previously defined DISPLAY_AIRPLANES_METHOD.

Continued on next page

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.
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 each airplane instance to the carrier. You may also create and transfer additional airplanes.
5. Display the attributes of the airline by calling its method DISPLAY_ATTRIBUTES.

Task 3: OPTIONAL: Downcast

Determining the highest cargo value.

1. In class lcl_cargo_plane, define and implement the public functional method GET_CARGO to return the cargo value. Define a returning parameter RV_CARGO.
2. In class lcl_carrier, define and implement the private functional method GET_MAX_CARGO to calculate the highest cargo value (load capacity) of all cargo planes. Loop at the list of airplanes and use the down-cast technique to identify cargo planes. For those airplanes for which the down cast was successful, you can then call the GET_CARGO method you defined and implemented in the previous step.
3. Call the GET_MAX_CARGO method from within the DISPLAY_ATTRIBUTES method class LCL_CARRIER and output the result.

Solution 9: Aggregation and Generic Calls

Task 1:

Define a local class for airlines.

1. Complete your program ZBC401_##_MAIN (where ## is your two-digit group number) or copy the template program.

If you copy the template program go directly to the next step. Otherwise copy local class LCL_CARRIER (definition and implementation) from the template program and paste it into your own program.

- a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
- b) See the source code extract from the model solution.
2. Define a private instance attribute MT_AIRPLANES to store references to airplanes there.
 - a) See the source code extract from the model solution.
3. Define and implement a public method ADD_AIRPLANE so that airplane references can be added to the previously defined list MT_AIRPLANES. The method should have one import parameter IO_PLANE.
 - a) See the source code extract from the model solution.
4. Define and implement a private instance method DISPLAY_AIRPLANES. In this method, loop at the list of airplanes and call method DISPLAY_ATTRIBUTES for each airplane.
 - a) See the source code extract from the model solution.
5. Extend the implementation of the method DISPLAY_ATTRIBUTES in a way that it not only displays information about the carrier itself but also about all its airplanes. Use the previously defined DISPLAY_AIRPLANES_METHOD.
 - 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.

Continued on next page

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.
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 each airplane instance to the carrier. 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: Downcast

Determining the highest cargo value.

1. In class lcl_cargo_plane, define and implement the public functional method GET_CARGO to return the cargo value. Define a returning parameter RV_CARGO.
 - a) See the source code extract from the model solution.
2. In class lcl_carrier, define and implement the private functional method GET_MAX_CARGO to calculate the highest cargo value (load capacity) of all cargo planes. Loop at the list of airplanes and use the down-cast technique to identify cargo planes. For those airplanes for which the down cast was successful, you can then call the GET_CARGO method you defined and implemented in the previous step.
 - a) See the source code extract from the model solution.
3. Call the GET_MAX_CARGO method from within the DISPLAY_ATTRIBUTES method class LCL_CARRIER and output the result.
 - a) See the source code extract from the model solution.

Result

SAPBC401_INH_S3

```
REPORT  sapbc401_inh_s3.
```

```
TYPE-POOLS icon.
```

Continued on next page

Internal Use SAP Partner Only

```
*-----*
*      CLASS lcl_airplane DEFINITION
*-----*
CLASS lcl_airplane DEFINITION.

...
ENDCLASS.          "lcl_airplane DEFINITION

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

...
ENDCLASS.          "lcl_airplane IMPLEMENTATION

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

...
ENDCLASS.          "lcl_cargo_plane DEFINITION

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

...
ENDCLASS.          "lcl_cargo_plane IMPLEMENTATION

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

...
ENDCLASS.          "lcl_passenger_plane DEFINITION
```

Continued on next page

Internal Use SAP Partner Only

```

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

...

ENDCLASS.                      "lcl_passenger_plane IMPLEMENTATION

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

PUBLIC SECTION.

METHODS:
constructor IMPORTING iv_name TYPE string,
display_attributes,
add_airplane IMPORTING io_plane TYPE REF TO lcl_airplane.

PRIVATE SECTION.

DATA:
mv_name      TYPE string,
mt_airplanes TYPE TABLE OF REF TO lcl_airplane.

METHODS:
display_airplanes.

ENDCLASS.                      "lcl_carrier DEFINITION

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

METHOD constructor.
mv_name = iv_name.
ENDMETHOD.                      "constructor

METHOD display_attributes.

```

Continued on next page

```

SKIP 2.

WRITE: icon_flight AS ICON,
       mv_name.

ULINE.
ULINE.

me->display_airplanes( ).

ENDMETHOD.           "display_attributes

METHOD add_airplane.

APPEND io_plane TO mt_airplanes.

ENDMETHOD.           "add_airplane

METHOD display_airplanes.

DATA: lo_plane TYPE REF TO lcl_airplane.

LOOP AT mt_airplanes INTO lo_plane.
   lo_plane->display_attributes( ).
ENDLOOP.

ENDMETHOD.           "display_airplanes

ENDCLASS.           "lcl_carrier IMPLEMENTATION

DATA:
go_carrier  TYPE REF TO lcl_carrier,
go_airplane  TYPE REF TO lcl_airplane,
go_cargo      TYPE REF TO lcl_cargo_plane,
go_passenger  TYPE REF TO lcl_passenger_plane,
gv_count      TYPE i.

START-OF-SELECTION.
***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly-Travel'.

***** Passenger Plane *****
CREATE OBJECT go_passenger
EXPORTING
  iv_name      = 'LH BERLIN'
  iv_planetype = '747-400'
  iv_seats     = 345
EXCEPTIONS
  wrong_planetype = 1.

```

Continued on next page

```

IF sy-subrc = 0.
  go_carrier->add_airplane( go_passenger ).
ENDIF.

***** cargo Plane ****
CREATE OBJECT go_cargo
  EXPORTING
    iv_name      = 'US Hercules'
    iv_planetype = '747-200F'
    iv_cargo     = 533
  EXCEPTIONS
    wrong_planetype = 1.
IF sy-subrc = 0.
  go_carrier->add_airplane( go_cargo ).
ENDIF.

***** output carrier (including list of airplanes) ****
go_carrier->display_attributes( ).
```

Optional Part: SAPBC401_INH_S3_OPT

```

REPORT sapbc401_inh_s3_opt.

TYPE-POOLS icon.

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

PUBLIC SECTION.

METHODS:
  constructor
  IMPORTING
    iv_name      TYPE string
    iv_planetype TYPE saplane-planetype
    iv_cargo     TYPE s_plan_car
```

Continued on next page

```

EXCEPTIONS
    wrong_planetype,
    display_attributes REDEFINITION,
    get_cargo RETURNING value(rv_cargo) TYPE s_plan_car.

PRIVATE SECTION.

DATA:
    mv_cargo TYPE s_plan_car.

ENDCLASS.                                     "lcl_cargo_plane DEFINITION

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

METHOD constructor.
    super->constructor(
        EXPORTING
            iv_name      = iv_name
            iv_planetype = iv_planetype
        EXCEPTIONS
            wrong_planetype = 1 .
        IF sy-subrc <> 0.
            RAISE wrong_planetype.
        ENDIF.
        mv_cargo = iv_cargo.
    ENDMETHOD.                                     "constructor

METHOD display_attributes.
    super->display_attributes( ).
    WRITE:
        / 'Max Cargo:'(005), AT c_pos_1 mv_cargo LEFT-JUSTIFIED.
    ULINE.
ENDMETHOD.                                     "display_attributes

METHOD get_cargo.
    rv_cargo = mv_cargo.
ENDMETHOD.                                     "get_cargo

ENDCLASS.                                     "lcl_cargo_plane IMPLEMENTATION

```

Continued on next page

```

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

PUBLIC SECTION.

METHODS:
    constructor IMPORTING iv_name TYPE string,
    display_attributes,
    add_airplane IMPORTING io_plane TYPE REF TO lcl_airplane.

PRIVATE SECTION.

DATA:
    mv_name      TYPE string,
    mt_airplanes TYPE TABLE OF REF TO lcl_airplane.

METHODS:
    display_airplanes,
    get_max_cargo RETURNING value(rv_max_cargo) TYPE s_plan_car.

ENDCLASS.                      "lcl_carrier DEFINITION

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

METHOD constructor.
    mv_name = iv_name.
ENDMETHOD.                      "constructor

METHOD display_attributes.

DATA:
    lv_max_cargo TYPE s_plan_car.

SKIP 2.
WRITE: icon_flight AS ICON,

```

Continued on next page

```

mv_name.

ULINE.

ULINE.

me->display_airplanes( ).

lv_max_cargo = me->get_max_cargo( ).

WRITE: / 'Capacity of biggest cargo plane:' (max),
      lv_max_cargo LEFT-JUSTIFIED.

ENDMETHOD.           "display_attributes

METHOD add_airplane.
  APPEND io_plane TO mt_airplanes.
ENDMETHOD.           "add_airplane

METHOD display_airplanes.
  DATA: lo_plane TYPE REF TO lcl_airplane.
  LOOP AT mt_airplanes INTO lo_plane.
    lo_plane->display_attributes( ).
  ENDLOOP.
ENDMETHOD.           "display_airplanes

METHOD get_max_cargo.
  DATA:
    lo_plane TYPE REF TO lcl_airplane,
    lo_cargo TYPE REF TO lcl_cargo_plane.

  LOOP AT mt_airplanes INTO lo_plane.
  TRY.
    lo_cargo ?= lo_plane.

    IF rv_max_cargo < lo_cargo->get_cargo( ).
      rv_max_cargo = lo_cargo->get_cargo( ).
    ENDIF.

    CATCH cx_sy_move_cast_error.
    *   plane is not a cargo plane - do nothing
    ENDTRY.
  ENDLOOP.
ENDMETHOD.           "get_max_cargo

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:

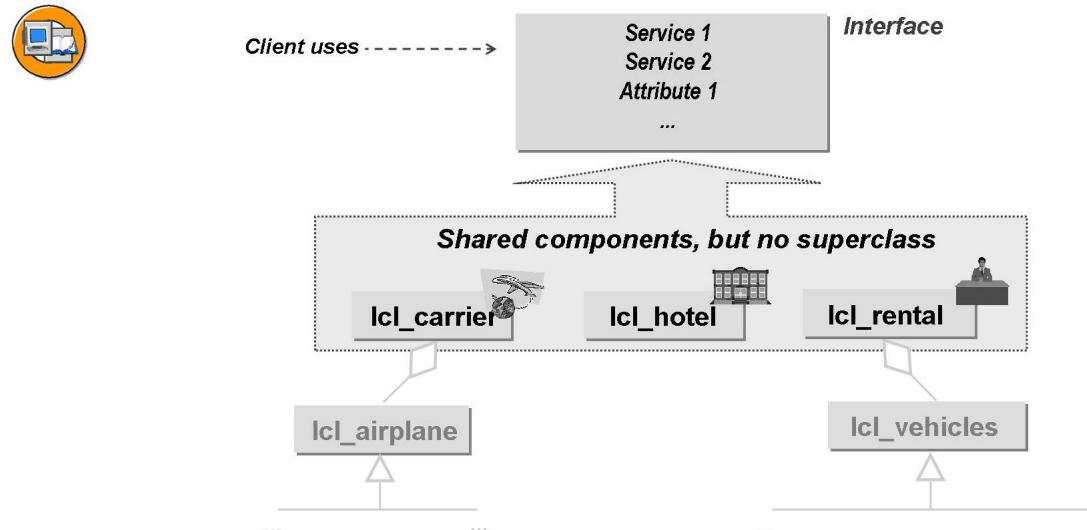


Figure 84: Central Definition of Shared Components

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.

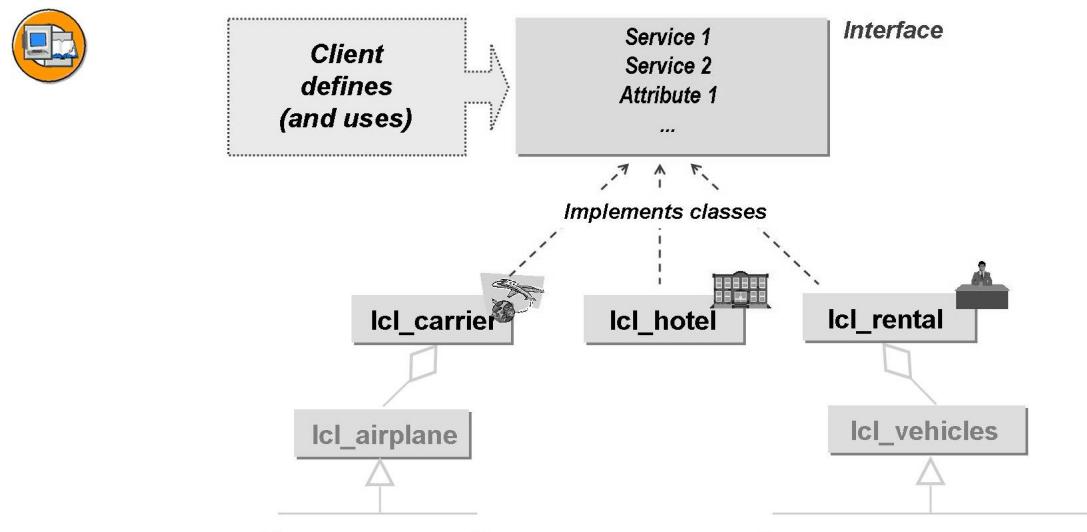


Figure 85: The Client Defines the Protocol

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 or 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.

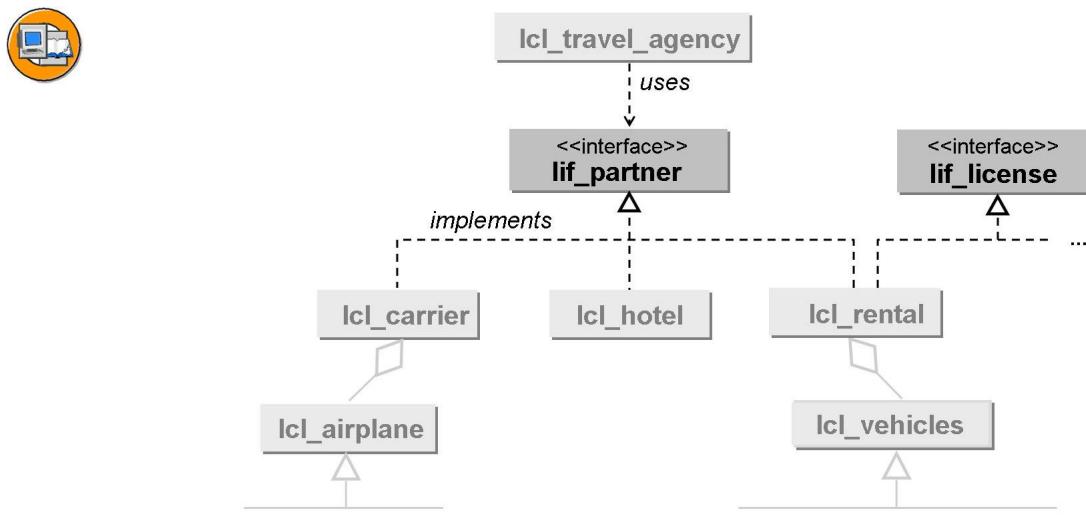


Figure 86: Interfaces in UML Notation

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.

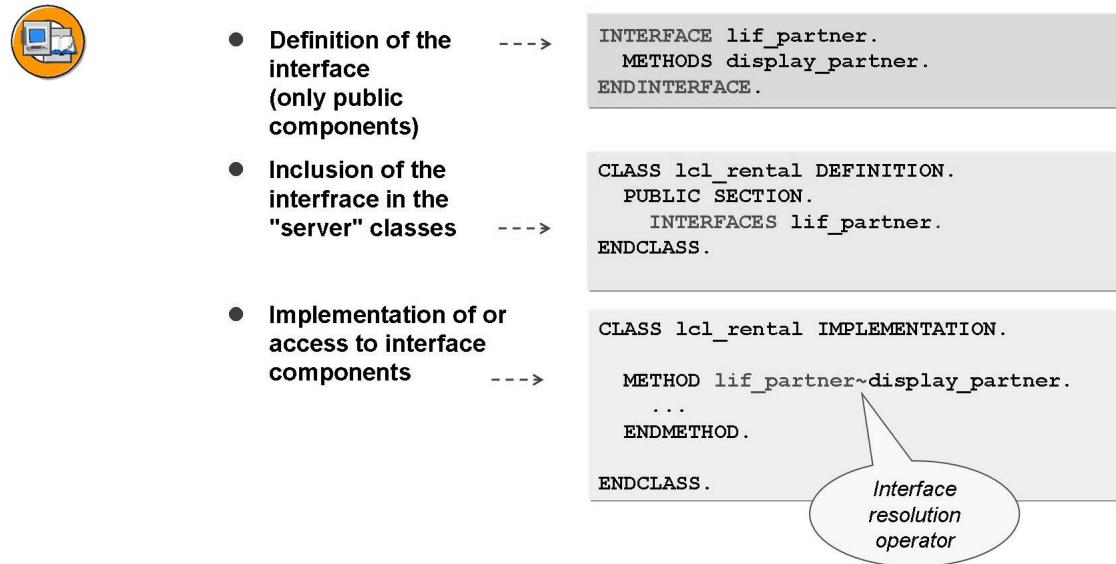


Figure 87: Defining and Implementing an Interface

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`.

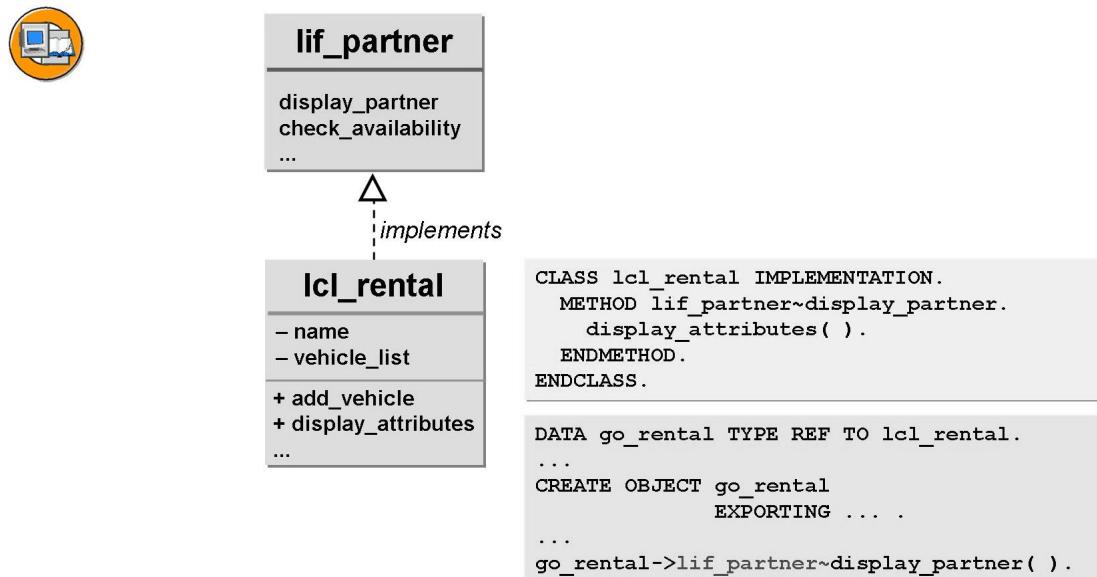


Figure 88: Addressing Interface Components Using Object References

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: go_rental  TYPE REF TO lcl_rental,
      go_partner TYPE REF TO lif_partner.
```

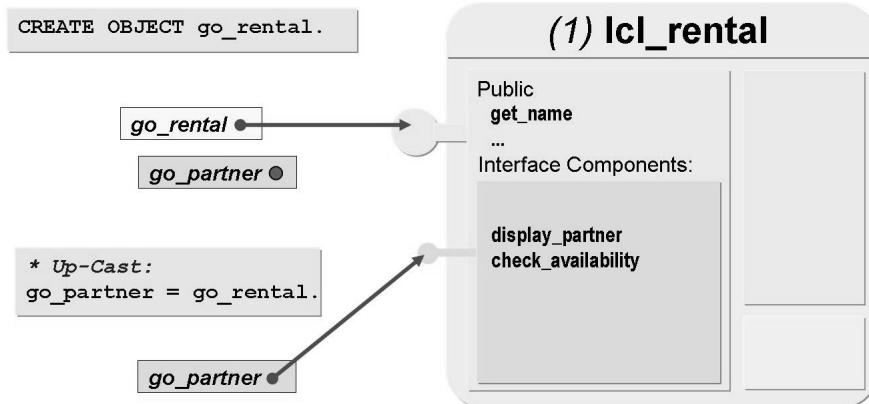


Figure 89: Up-Cast with Interface References

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 `GO_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 `GO_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.

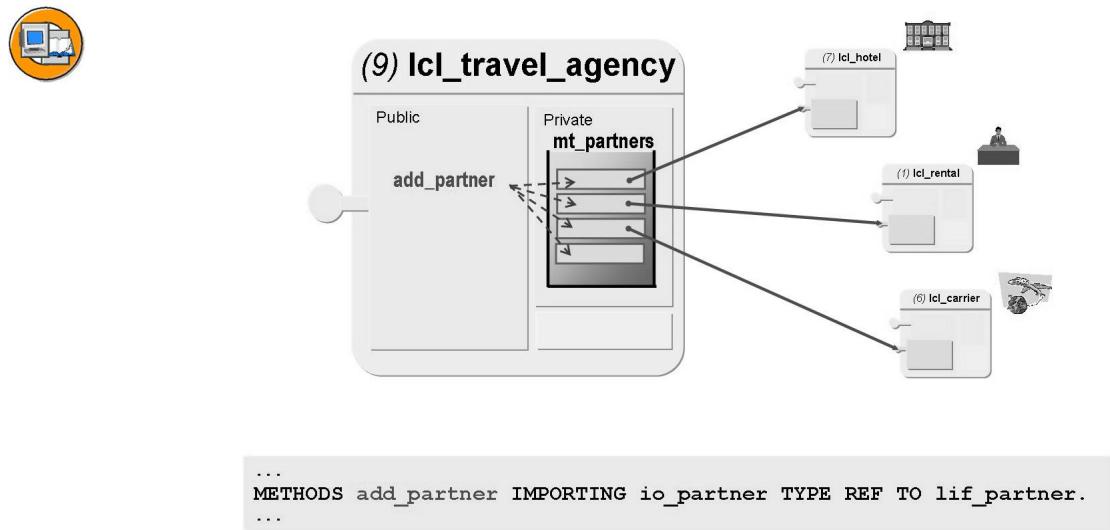


Figure 90: Row Type of the Internal Table in the Application Example

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 lo_partner TYPE REF TO lif_partner.
LOOP AT mt_partners INTO lo_partner.
  lo_partner->display_partner( ).
ENDLOOP.
ENDMETHOD.
```

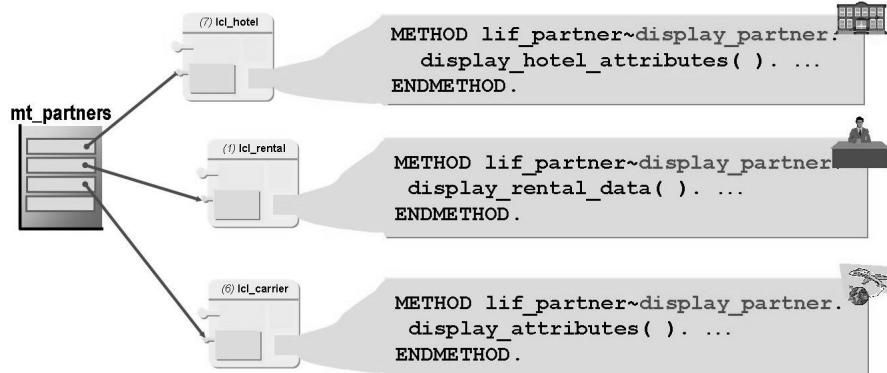


Figure 91: Polymorphism – Generic Access Using the Interface Reference

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, **go_partner->display_partner()** therefore uses the class of the instance to which **go_partner** actually refers in order to search for the implementation of **display_partner**. It does not, for example, use the static type for **go_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 **GO_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.

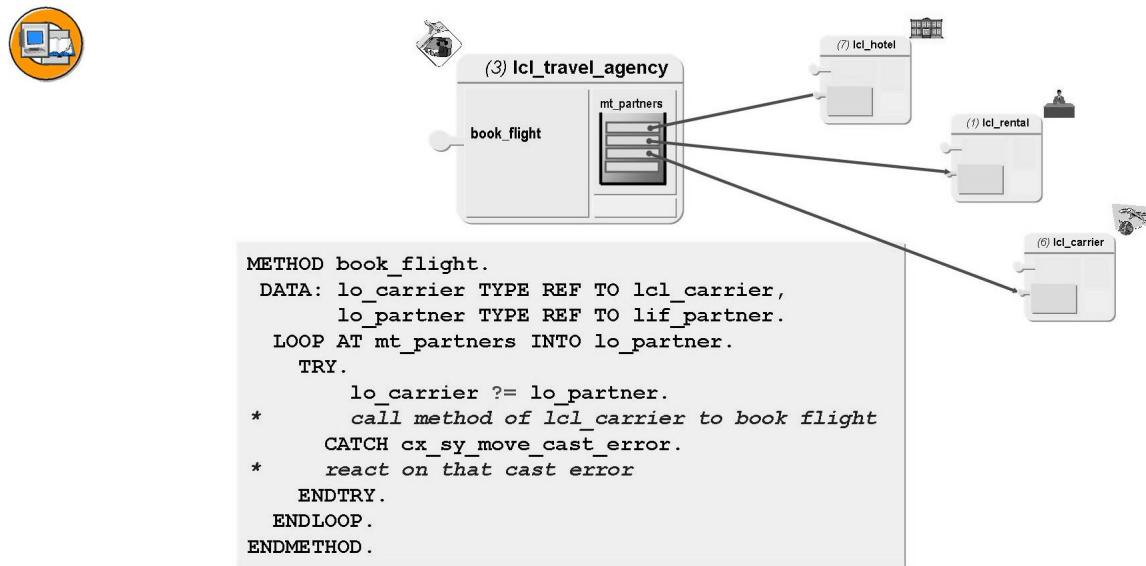


Figure 92: Down-Cast Assignment and Exception Handling in the Application Example

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 GO_PARTNER at runtime but the down-cast assignment operator is used to copy the reference to the then invalid reference GO_CARRIER? In contrast to the up-cast assignment, it is possible that the static type of the target variable (GO_CARRIER) is neither more general than nor the same as the dynamic type of the source variables (GO_PARTNER), specifically if GO_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.

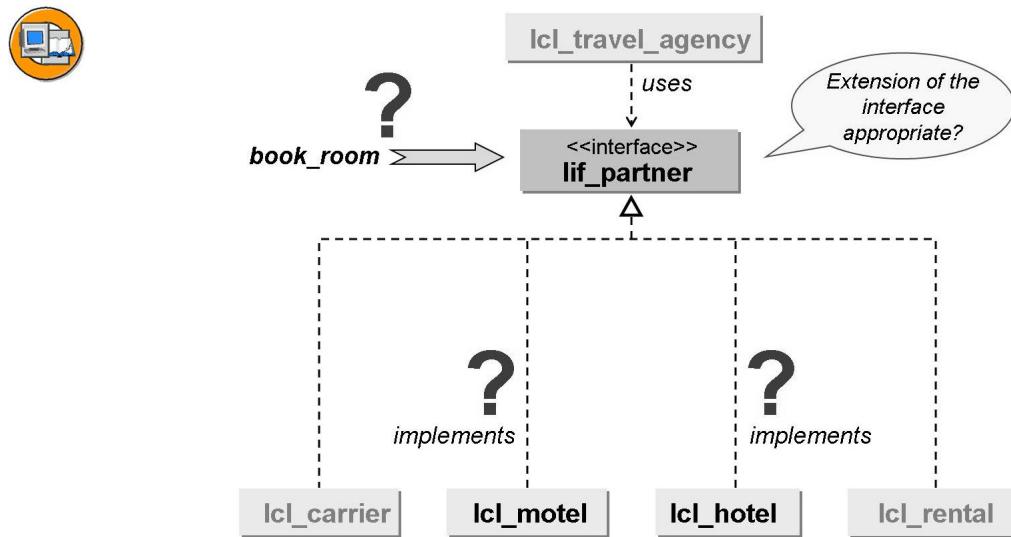


Figure 93: Interface Hierarchy in the Application Example

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.

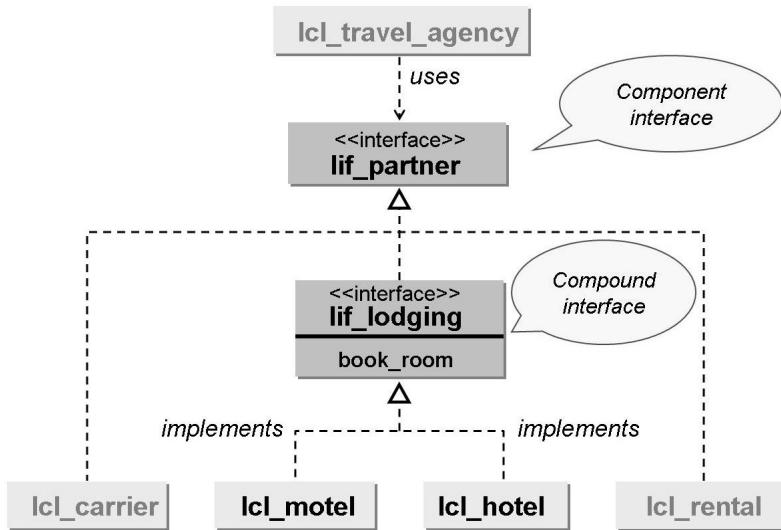


Figure 94: Compound Interface in UML Notation

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.

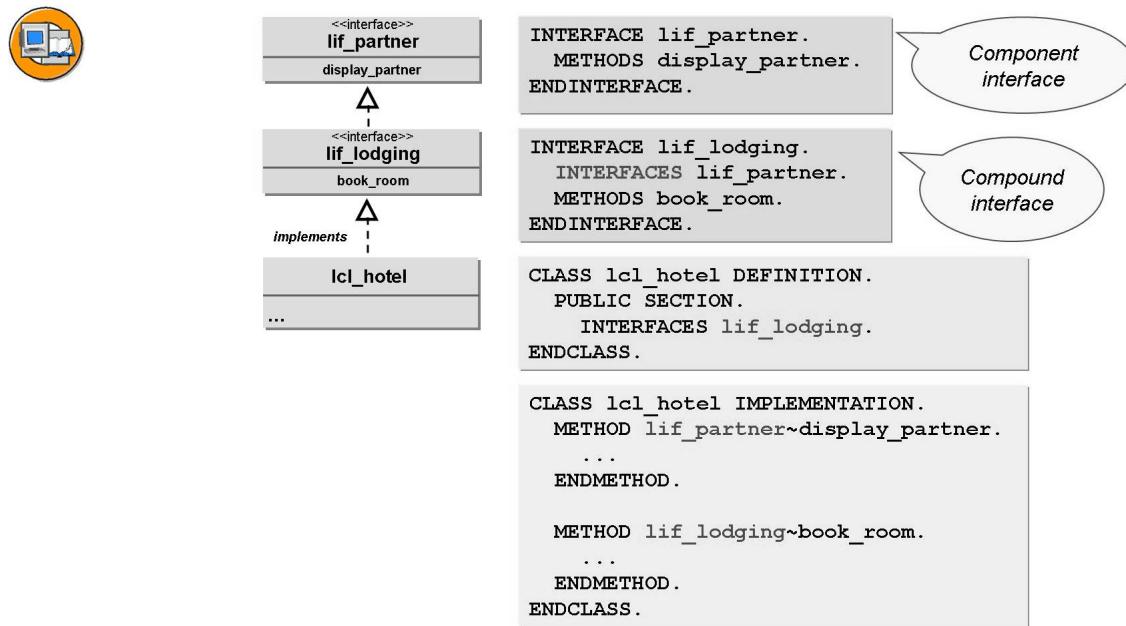


Figure 95: Definition and Implementation of Compound Interfaces: Syntax

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: go_hotel    TYPE REF TO lcl_hotel,
      go_partner  TYPE REF TO lif_partner,
      go_lodging   TYPE REF TO lif_lodging.
```

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

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

go_partner = go_hotel.
go_partner->display_partner( ).

* Down-Casts for specific access again:
go_lodging ?= go_partner.
* or:
go_hotel ?= go_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 want to add a car rental company to your program.

The car rental company – together with the already existing airline – should then be aggregated by a travel agency class as the travel agency's business partners. For such an aggregation, a generic access to airlines **and** car rental companies is needed. Instead of defining a common superclass, you decide to define an interface with generic services, which is then implemented in the classes for airlines and car rental companies.

In this first exercise, you define and implement the interface in your already existing class for airlines.

Template:

SAPBC401_INH_S3

Solution:

SAPBC401_INT_S1

Task 1:

In order to increase the legibility of your program, first split the source code into a main program and one include program. The include program should contain all the local class definitions and implementations, the declaration of reference variables and the START-OF-SELECTION event block should remain in the main program.



Note: This distribution is not according to the recommended use of includes in ABAP programs. According to the programming guidelines, the class definitions and data declarations should go to the TOP-Include, the implementations to a P-include and the START-OF-SELECTION event block to an E-include.

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

Continued on next page

2. In your main program implement the usage of a new include program (recommended name: **ZBC401_##_CARRIER** with ## being your two-digit group number). Create the include through forward navigation (double-clicking on the include's name).
3. Cut the definition and implementation sections of all four local classes from the main program and paste them into the newly created include program.
4. Where do you have to place the INCLUDE statement to avoid syntax errors?

5. Activate and test your program.

Task 2:

Define an interface with the method **DISPLAY_PARTNER** in order to later offer generic access options to potential clients (in our example a travel agency).

1. Create a new include (recommended name: **ZBC401_##_AGENCY** with ## being your two-digit group number) and define an interface **LIF_PARTNER** in this new include.
2. The interface should contain the instance method **DISPLAY_PARTNER**. The method should not have any parameters. Later on, your task will be to display business partner attributes.

Check your entire program for syntactical correctness and activate it.

Task 3:

The class **LCL_CARRIER** should make the services defined in the interface available to potential clients. To achieve this, class **LCL_CARRIER** needs to implement the interface and therefore the interface method **DISPLAY_PARTNER**.

1. In the definition of **LCL_CARRIER**, declare the newly created interface.
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.

Continued on next page

Check your entire application 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 4:

Test the interface implemented in the class LCL_CARRIER by calling the interface method from the main program.

1. In the main program, remove the call of the DISPLAY_ATTRIBUTES method of class LCL_CARRIER. Replace it with a call of the interface method DISPLAY_PARTNER.
2. 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?

3. 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.

Solution 10: Interface Definition und Implementation

Task 1:

In order to increase the legibility of your program, first split the source code into a main program and one include program. The include program should contain all the local class definitions and implementations, the declaration of reference variables and the START-OF-SELECTION event block should remain in the main program.



Note: This distribution is not according to the recommended use of includes in ABAP programs. According to the programming guidelines, the class definitions and data declarations should go to the TOP-Include, the implementations to a P-include and the START-OF-SELECTION event block to an E-include.

1. Complete your program **ZBC401##_MAIN** (## is your two-digit group number) or copy the sample solution from the previous exercise.
 - a) Carry out this step in the usual manner. Additional information on the copying of programs is available in the SAP Library.
2. In your main program implement the usage of a new include program (recommended name: **ZBC401##_CARRIER** with ## being your two-digit group number). Create the include through forward navigation (double-clicking on the include's name).
 - a) See source text excerpt from the model solution.
3. Cut the definition and implementation sections of all four local classes from the main program and paste them into the newly created include program.
 - a) See source text excerpt from the model solution.
4. Where do you have to place the INCLUDE statement to avoid syntax errors?

Answer: The INCLUDE statement has to be placed **before** the declaration of reference variables. Otherwise the class definitions are not known in the declaration of reference variables.
5. Activate and test your program.
 - a) Carry out this step in the usual manner. Additional information on the copying of programs is available in the SAP Library.

Continued on next page

Task 2:

Define an interface with the method DISPLAY_PARTNER in order to later offer generic access options to potential clients (in our example a travel agency).

1. Create a new include (recommended name: **ZBC401_##_AGENCY** with ## being your two-digit group number) and define an interface **LIF_PARTNER** in this new include.
 - a) See source text excerpt from the model solution.
2. The interface should contain the instance method **DISPLAY_PARTNER**. The method should not have any parameters. Later on, your task will be to display business partner attributes.

Check your entire program for syntactical correctness and activate it.

- a) See source text excerpt from the model solution.

Task 3:

The class LCL_CARRIER should make the services defined in the interface available to potential clients. To achieve this, class LCL_CARRIER needs to implement the interface and therefore the interface method DISPLAY_PARTNER.

1. In the definition of LCL_CARRIER, declare the newly created interface.
 - 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 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.

Continued on next page

Task 4:

Test the interface implemented in the class LCL_CARRIER by calling the interface method from the main program.

1. In the main program, remove the call of the DISPLAY_ATTRIBUTES method of class LCL_CARRIER. Replace it with a call of the interface method DISPLAY_PARTNER.
 - a) See source text excerpt from the model solution.
2. 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?

Answer: 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.

3. 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.

Result

Main program: SAPBC401_INT_S1

```
REPORT  sapbc401_int_s1.

TYPE-POOLS icon.

INCLUDE bc401_int_s1_agency.
INCLUDE bc401_int_s1_carrier.

DATA:
go_carrier    TYPE REF TO lcl_carrier,
go_airplane   TYPE REF TO lcl_airplane,
go_cargo      TYPE REF TO lcl_cargo_plane,
go_passenger  TYPE REF TO lcl_passenger_plane,
```

Continued on next page

```

gv_count      TYPE i.

START-OF-SELECTION.
***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly-Travel'.

***** Passenger Plane *****
CREATE OBJECT go_passenger
EXPORTING
  iv_name      = 'LH BERLIN'
  iv_planetype = '747-400'
  iv_seats     = 345
EXCEPTIONS
  wrong_planetype = 1.

IF sy-subrc = 0.
  go_carrier->add_airplane( go_passenger ).
ENDIF.

***** cargo Plane *****
CREATE OBJECT go_cargo
EXPORTING
  iv_name      = 'US Hercules'
  iv_planetype = '747-200F'
  iv_cargo     = 533
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc = 0.
  go_carrier->add_airplane( go_cargo ).
ENDIF.

***** output carrier (including list of airplanes) *****
* only for testing:
* call interface method for the carrier

  go_carrier->lif_partner~display_partner( ).
*  go_carrier->display_attributes( ).
```

Continued on next page

Include BC401_INT_S1_AGENCY

```
INTERFACE lif_partner.  
  METHODS display_partner.  
ENDINTERFACE.
```

Include BC401_INT_S1_CARRIER

```
*-----  
*      CLASS lcl_airplane DEFINITION  
*-----  
CLASS lcl_airplane DEFINITION.  
  
...  
  
ENDCLASS.          "lcl_airplane DEFINITION  
*-----  
*      CLASS lcl_airplane IMPLEMENTATION  
*-----  
CLASS lcl_airplane IMPLEMENTATION.  
  
...  
  
ENDCLASS.          "lcl_airplane IMPLEMENTATION  
*-----  
*      CLASS lcl_cargo_plane DEFINITION  
*-----  
*  
*-----  
CLASS lcl_cargo_plane DEFINITION INHERITING FROM lcl_airplane.  
  
...  
  
ENDCLASS.          "lcl_cargo_plane DEFINITION  
*-----  
*      CLASS lcl_cargo_plane IMPLEMENTATION  
*-----  
CLASS lcl_cargo_plane IMPLEMENTATION.  
  
...  
  
ENDCLASS.          "lcl_cargo_plane IMPLEMENTATION
```

Continued on next page

```

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

...
ENDCLASS.          "lcl_passenger_plane DEFINITION
*-----*
*      CLASS lcl_passenger_plane IMPLEMENTATION
*-----*
CLASS lcl_passenger_plane IMPLEMENTATION.

...
ENDCLASS.          "lcl_passenger_plane IMPLEMENTATION
*-----*
*      CLASS lcl_carrier DEFINITION
*-----*
CLASS lcl_carrier DEFINITION.

PUBLIC SECTION.

INTERFACES:
  lif_partner.

METHODS:
  constructor IMPORTING iv_name TYPE string,
  display_attributes,
  add_airplane IMPORTING io_plane TYPE REF TO lcl_airplane.

PRIVATE SECTION.

DATA:
  mv_name      TYPE string,
  mt_airplanes TYPE TABLE OF REF TO lcl_airplane.

METHODS:
  display_airplanes.

ENDCLASS.          "lcl_carrier DEFINITION

```

Continued on next page

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

METHOD constructor.
    mv_name = iv_name.
ENDMETHOD.           "constructor

METHOD display_attributes.
    SKIP 2.
    WRITE: icon_flight AS ICON,
           mv_name.
    ULINE.
    ULINE.
    me->display_airplanes( ).
ENDMETHOD.           "display_attributes

METHOD add_airplane.
    APPEND io_plane TO mt_airplanes.
ENDMETHOD.           "add_airplane

METHOD display_airplanes.
    DATA: lo_plane TYPE REF TO lcl_airplane.
    LOOP AT mt_airplanes INTO lo_plane.
        lo_plane->display_attributes( ).
    ENDLOOP.
ENDMETHOD.           "display_airplanes

METHOD lif_partner~display_partner.
    me->display_attributes( ).
ENDMETHOD.           "lif_partner~display_partner

ENDCLASS.           "lcl_carrier IMPLEMENTATION
```

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 class that uses the previously defined interface: Travel agencies need to manage various business partners using the interface and access the business partners' general services.

Template:

SAPBC401_INT_T2

Solution:

SAPBC401_INT_S2

Task 1:

Define a local class for travel agencies.

1. Complete your ZBC401##_MAIN program (## is your two-digit group number) or copy the template program with all its includes.
2. In the include that contains the definition of interface LIF_PARTNER, define and implement local class LCL_TRAVEL_AGENCY.



Hint: If you have not copied the template program, you can nevertheless reduce your typing load by copying the definition and implementation of class LCL_TRAVEL_AGENCY from the template SAPBC401_INT_T2, include BC401_INT_T2_AGENCY. Of course, class LCL_TRAVEL_AGENCY can also be created from scratch, without using templates.

3. As a private class attribute, add an internal table MT_PARTNER for the buffering of references to the business partners who have implemented the interface LIF_PARTNER.
4. Define and implement a public method ADD_PARTNER so that the business partner references can be added to the list MT_PARTNERS. Define a suitably typed import parameter IO_PARTNER.

Continued on next page

5. Extend the implementation of the method DISPLAY_ATTRIBUTES in a way that it not only displays information about the travel agency itself but also about all its business partners. Use the previously defined DISPLAY_PARTNERS method.

Task 2:

In the main program, generate a travel agency instance, transfer the references to the airline 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 instances to the travel agency.
4. Display the attributes of the travel agency (including information about its business partners) by calling its method DISPLAY_ATTRIBUTES.

Solution 11: Use of Interfaces

Task 1:

Define a local class for travel agencies.

1. Complete your ZBC401_##_MAIN program (## is your two-digit group number) or copy the template program with all its includes.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
2. In the include that contains the definition of interface LIF_PARTNER, define and implement local class LCL_TRAVEL_AGENCY.



Hint: If you have not copied the template program, you can nevertheless reduce your typing load by copying the definition and implementation of class LCL_TRAVEL_AGENCY from the template SAPBC401_INT_T2, include BC401_INT_T2_AGENCY. Of course, 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.
3. As a private class attribute, add an internal table MT_PARTNER for the buffering of references to the business partners who have implemented the interface LIF_PARTNER.
 - a) See the source code extract from the model solution.
4. Define and implement a public method ADD_PARTNER so that the business partner references can be added to the list MT_PARTNERS. Define a suitably typed import parameter IO_PARTNER.
 - a) See the source code extract from the model solution.
5. Extend the implementation of the method DISPLAY_ATTRIBUTES in a way that it not only displays information about the travel agency itself but also about all its business partners. Use the previously defined DISPLAY_PARTNERS method.
 - 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 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 instances to the travel agency.
 - a) See the source code extract from the model solution.
4. Display the attributes of the travel agency (including information about its business partners) by calling its method DISPLAY_ATTRIBUTES.
 - a) See the source code extract from the model solution.

Result

Include BC401_INT_S2_AGENCY

```
*<-----*
*&   Include          BC401_INT_S2_AGENCY
*-----*>
INTERFACE lif_partner.

METHODS:
  display_partner.

ENDINTERFACE.           "lif_partner
*-----*
*      CLASS lcl_travel_agency DEFINITION
*-----*
CLASS lcl_travel_agency DEFINITION.

PUBLIC SECTION.

METHODS:
  constructor IMPORTING iv_name TYPE string,
```

Continued on next page

```

add_partner IMPORTING io_partner TYPE REF TO lif_partner,
display_agency_partners,
display_attributes.

PRIVATE SECTION.

DATA:
mv_name      TYPE string,
mt_partners  TYPE TABLE OF REF TO lif_partner.

ENDCLASS.          "lcl_travel_agency DEFINITION
*-----*
*      CLASS lcl_travel_agency IMPLEMENTATION
*-----*
CLASS lcl_travel_agency IMPLEMENTATION.

METHOD display_attributes.
WRITE: / icon_private_files AS ICON,
      'Travel Agency:'(007), mv_name.
ULINE.
display_agency_partners( ).  

ENDMETHOD.          "display_attributes

METHOD display_agency_partners.
DATA:
lo_partner TYPE REF TO lif_partner.

WRITE 'Here are the partners of the travel agency:'(008).
ULINE.
LOOP AT mt_partners INTO lo_partner.
lo_partner->display_partner( ).  

ENDLOOP.

ENDMETHOD.          "display_agency_partners

METHOD constructor.
mv_name = iv_name.
ENDMETHOD.          "constructor

METHOD add_partner.
APPEND io_partner TO mt_partners.
ENDMETHOD.          "add_partner

```

Continued on next page

```
ENDCLASS.           "lcl_travel_agency IMPLEMENTATION
```

Main Program SAPBC401_INT_S2

```
REPORT  sapbc401_int_s2.
```

```
TYPE-POOLS icon.
```

```
INCLUDE bc401_int_s2_agency.  
INCLUDE bc401_int_s2_carrier.
```

```
DATA:
```

```
go_agency      TYPE REF TO lcl_travel_agency,  
go_carrier     TYPE REF TO lcl_carrier,  
go_airplane    TYPE REF TO lcl_airplane,  
go_cargo       TYPE REF TO lcl_cargo_plane,  
go_passenger   TYPE REF TO lcl_passenger_plane,  
gv_count       TYPE i.
```

```
START-OF-SELECTION.
```

```
*****
```

```
***** create travel_agency *****
```

```
CREATE OBJECT go_agency  
EXPORTING  
iv_name = 'Travel&Smile Travel'.
```

```
***** Create Carrier *****
```

```
CREATE OBJECT go_carrier  
EXPORTING  
iv_name = 'Smile&Fly Travel'.
```

```
***** Insert carrier into business partner list of travel agency
```

```
go_agency->add_partner( go_carrier ).
```

```
***** Passenger Plane *****
```

```
CREATE OBJECT go_passenger  
EXPORTING  
iv_name      = 'LH BERLIN'  
iv_planetype = '747-400'  
iv_seats     = 345  
EXCEPTIONS  
wrong_planetype = 1.
```

Continued on next page

```
IF sy-subrc = 0.  
    go_carrier->add_airplane( go_passenger ).  
ENDIF.  
  
***** cargo Plane *****  
CREATE OBJECT go_cargo  
EXPORTING  
    iv_name      = 'US Hercules'  
    iv_planetype = '747-200F'  
    iv_cargo     = 533  
EXCEPTIONS  
    wrong_planetype = 1.  
IF sy-subrc = 0.  
    go_carrier->add_airplane( go_cargo ).  
ENDIF.  
  
***** show attributes of all partners of travel_agency *****  
go_agency->display_attributes( ).
```


Exercise 12: Integrate Trainer's Object Model

Exercise Objectives

After completing this exercise, you will be able to:

- Define interfaces
- Implement interfaces
- Work with interfaces

Business Example

You now do the final step and add the car rental company with its vehicles to your own object model. By implementing the already defined interface, you very easily ensure that the car rental company can be added to the list of business partners of the travel agency.

Template:

SAPBC401_INT_T3

Solution:

SAPBC401_INT_S3

Task 1:

Copy the definitions and implementations of classes LCL_RENTAL and the vehicle classes (LCL_VEHICLE, LCL_TRUCK and LCL_BUS) to your program and create instances of these classes.

1. Complete your ZBC401##_MAIN program (## is your two-digit group number) or copy the template program with all its includes.
2. If you have copied the template program analyze it and then continue with the next task. If you continue working with your own program first copy include program BC401_INT_T3_RENTAL (recommended name: ZBC401##_RENTAL, where ## is your two-digit group number) and make the new include program part of your own program.
3. In your main program, create an instance of LCL_RENTAL and a number of instances of vehicle classes. Add all the vehicles to the rental company's list of vehicles by calling method ADD_VEHICLE of LCL_RENTAL.



Hint: To reduce the typing effort you can copy and paste the relevant coding from the main program of SAPBC401_INT_T3.

Continued on next page

Task 2:

Implement Interface LIF_PARTNER in class LCL_RENTAL

1. Call method ADD_PARTNER to add the instance of LCL_RENTAL to the travel agency's list of business partners.
2. Perform a syntax check. Why does this method call cause a syntax error?

3. Go to the definition part of class LCL_RENTAL and declare the implementation of interface LIF_PARTNER.
4. Provide an implementation of the method defined in the interface.
5. Activate the include with the definition and implementation of LCL_RENTAL. Then go to the main program and perform another syntax check. Why does the call of ADD_PARTNER now work for the car rental company?

Solution 12: Integrate Trainer's Object Model

Task 1:

Copy the definitions and implementations of classes LCL_RENTAL and the vehicle classes (LCL_VEHICLE, LCL_TRUCK and LCL_BUS) to your program and create instances of these classes.

1. Complete your ZBC401_##_MAIN program (## is your two-digit group number) or copy the template program with all its includes.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
2. If you have copied the template program analyze it and then continue with the next task. If you continue working with your own program first copy include program BC401_INT_T3_RENTAL (recommended name: ZBC401_##_RENTAL, where ## is your two-digit group number) and make the new include program part of your own program.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
3. In your main program, create an instance of LCL_RENTAL and a number of instances of vehicle classes. Add all the vehicles to the rental company's list of vehicles by calling method ADD_VEHICLE of LCL_RENTAL.



Hint: To reduce the typing effort you can copy and paste the relevant coding from the main program of SAPBC401_INT_T3.

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

Task 2:

Implement Interface LIF_PARTNER in class LCL_RENTAL

1. Call method ADD_PARTNER to add the instance of LCL_RENTAL to the travel agency's list of business partners.
 - a) See the source code extract from the model solution.
2. Perform a syntax check. Why does this method call cause a syntax error?

Answer: The up cast in the parameter passing of the method does not work because the types LCL_RENTAL and LIF_PARTNER are not compatible. They will become compatible, though, as soon as LCL_RENTAL implements the interface.

Continued on next page

3. Go to the definition part of class LCL_RENTAL and declare the implementation of interface LIF_PARTNER.
 - a) See the source code extract from the model solution.
4. Provide an implementation of the method defined in the interface.
 - a) See the source code extract from the model solution.
5. Activate the include with the definition and implementation of LCL_RENTAL. Then go to the main program and perform another syntax check. Why does the call of ADD_PARTNER now work for the car rental company?

Answer: Because now the types LCL_RENTAL and LIF_PARTNER are compatible. LCL_RENTAL implements interface LIF_PARTNER. Any instance of LCL_RENTAL is accepted as a business partner.

Result

Include BC401_INT_S3_RENTAL

```
*-----*
*      CLASS lcl_vehicle DEFINITION
*-----*
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION.

METHODS:
  constructor IMPORTING iv_make TYPE string,
  display_attributes.

PRIVATE SECTION.

DATA:
  mv_make TYPE string.

ENDCLASS.          "lcl_vehicle DEFINITION

*-----*
*      CLASS lcl_vehicle IMPLEMENTATION
*-----*
CLASS lcl_vehicle IMPLEMENTATION.

METHOD constructor.
  mv_make = iv_make.
ENDMETHOD.          "constructor
```

Continued on next page

```

METHOD display_attributes.
  WRITE mv_make.
ENDMETHOD.           "display_attributes

ENDCLASS.          "lcl_vehicle IMPLEMENTATION

*-----*
*      CLASS lcl_truck DEFINITION
*-----*
CLASS lcl_truck DEFINITION INHERITING FROM lcl_vehicle.

PUBLIC SECTION.

METHODS:
  constructor
  IMPORTING
    iv_make  TYPE string
    iv_cargo TYPE s_plan_car,
    display_attributes REDEFINITION,

PRIVATE SECTION.

DATA:
  mv_cargo TYPE s_plan_car.

ENDCLASS.          "lcl_truck DEFINITION

*-----*
*      CLASS lcl_truck IMPLEMENTATION
*-----*
CLASS lcl_truck IMPLEMENTATION.

METHOD constructor.
  super->constructor( iv_make ).
  mv_cargo = iv_cargo.
ENDMETHOD.           "constructor

METHOD display_attributes.
  WRITE / icon_ws_truck AS ICON.
  super->display_attributes( ).
  WRITE:
    20 'Max. Cargo:'(005),
    mv_cargo.

```

Continued on next page

```

        ULINE.
ENDMETHOD.           "display_attributes

ENDCLASS.           "lcl_truck DEFINITION

*-----*
*      CLASS lcl_bus DEFINITION
*-----*
CLASS lcl_bus DEFINITION INHERITING FROM lcl_vehicle.

PUBLIC SECTION.

METHODS:
constructor
IMPORTING iv_make      TYPE string
          iv_passengers TYPE i,
display_attributes REDEFINITION.

PRIVATE SECTION.

DATA:
mv_passengers TYPE i.

ENDCLASS.           "lcl_bus DEFINITION

*-----*
*      CLASS lcl_bus IMPLEMENTATION
*-----*
CLASS lcl_bus IMPLEMENTATION.

METHOD constructor.
super->constructor( iv_make ).
mv_passengers = iv_passengers.
ENDMETHOD.           "constructor

METHOD display_attributes.
WRITE: / icon_transportation_mode AS ICON.
super->display_attributes( ).
WRITE:
20 'Max. Passengers: '(006),
mv_passengers.
ULINE.
ENDMETHOD.           "display_attributes

```

Continued on next page

```

ENDCLASS.                               "lcl_bus DEFINITION

*-----*
*      CLASS lcl_rental DEFINITION
*-----*
CLASS lcl_rental DEFINITION.

PUBLIC SECTION.

INTERFACES:
  lif_partner.

METHODS:
  constructor IMPORTING iv_name TYPE string,
    add_vehicle IMPORTING io_vehicle TYPE REF TO lcl_vehicle,
    display_attributes.

PRIVATE SECTION.

DATA:
  mv_name TYPE string,
  mt_vehicles TYPE TABLE OF REF TO lcl_vehicle.

ENDCLASS.                               "lcl_rental DEFINITION
*-----*
*      CLASS lcl_rental IMPLEMENTATION
*-----*
CLASS lcl_rental IMPLEMENTATION.

METHOD constructor.
  mv_name = iv_name.
ENDMETHOD.                            "constructor

METHOD add_vehicle.
  APPEND io_vehicle TO mt_vehicles.
ENDMETHOD.                            "add_vehicle

METHOD display_attributes.
DATA:
  lo_vehicle TYPE REF TO lcl_vehicle.

WRITE:

```

Continued on next page

```

    / icon_transport_proposal AS ICON,
      mv_name.
    ULINE.
    ULINE.

    LOOP AT mt_vehicles INTO lo_vehicle.
      lo_vehicle->display_attributes( ).
    ENDLOOP.
ENDMETHOD.           "display_attributes

METHOD lif_partner~display_partner.
  me->display_attributes( ).
ENDMETHOD.           "lif_partner~display_partner

ENDCLASS.           "lcl_rental IMPLEMENTATION

```

Main Program SAPBC401_INT_S3

```

REPORT  sapbc401_int_s3.

TYPE-POOLS icon.

INCLUDE bc401_int_s3_agency.
INCLUDE bc401_int_s3_carrier.
INCLUDE bc401_int_s3_rental.

DATA:
  go_vehicle  TYPE REF TO lcl_vehicle,
  go_truck    TYPE REF TO lcl_truck,
  go_bus      TYPE REF TO lcl_bus,
  go_rental   TYPE REF TO lcl_rental.

DATA:
  go_agency   TYPE REF TO lcl_travel_agency,
  go_carrier  TYPE REF TO lcl_carrier,
  go_airplane TYPE REF TO lcl_airplane,
  go_cargo    TYPE REF TO lcl_cargo_plane,
  go_passenger TYPE REF TO lcl_passenger_plane,
  gv_count    TYPE i.

START-OF-SELECTION.
*****
```

```
***** create travel_agency *****
```

Continued on next page

```

CREATE OBJECT go_agency
EXPORTING
    iv_name = 'Travel&Smile Travel'.

***** create rental *****
CREATE OBJECT go_rental
EXPORTING
    iv_name = 'Happy Car Rental'.

***** Insert rental company into partner list of travel agency
go_agency->add_partner( go_rental ).

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
    iv_make = 'MAN'
    iv_cargo = 45.

go_rental->add_vehicle( go_truck ).

***** create truck *****
CREATE OBJECT go_bus
EXPORTING
    iv_make      = 'Mercedes'
    iv_passengers = 80.

go_rental->add_vehicle( go_bus ).

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
    iv_make = 'VOLVO'
    iv_cargo = 48.

go_rental->add_vehicle( go_truck ).

***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
    iv_name = 'Smile&Fly-Travel'.

***** Insert carrier into business partner list of travel agency
go_agency->add_partner( go_carrier ).

***** Passenger Plane *****

```

Continued on next page

```
CREATE OBJECT go_passenger
EXPORTING
  iv_name      = 'LH BERLIN'
  iv_planetype = '747-400'
  iv_seats     = 345
EXCEPTIONS
  wrong_planetype = 1.

IF sy-subrc = 0.
  go_carrier->add_airplane( go_passenger ).
ENDIF.

***** cargo Plane *****
CREATE OBJECT go_cargo
EXPORTING
  iv_name      = 'US Hercules'
  iv_planetype = '747-200F'
  iv_cargo     = 533
EXCEPTIONS
  wrong_planetype = 1.

IF sy-subrc = 0.
  go_carrier->add_airplane( go_cargo ).
ENDIF.

***** show attributes of all partners of travel_agency *****
go_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.

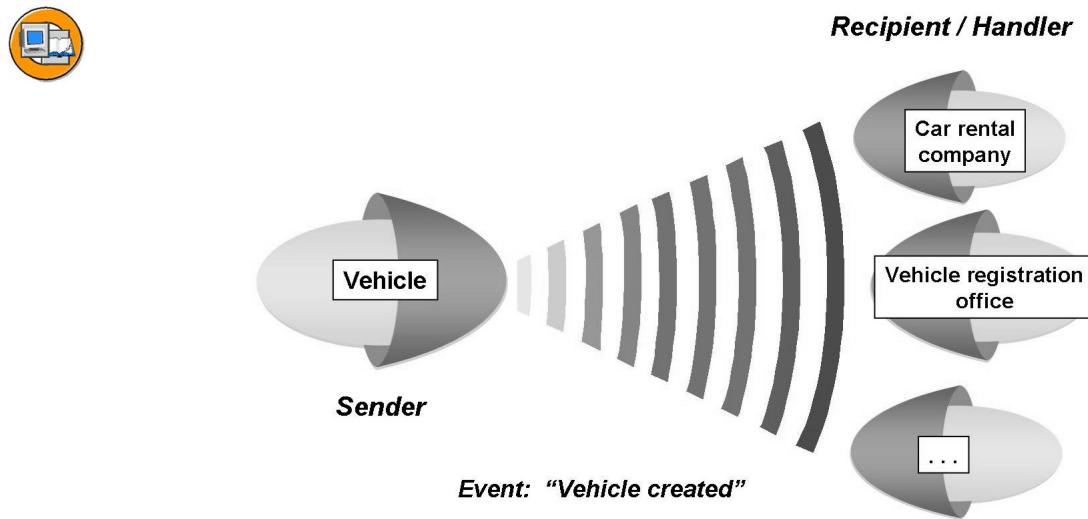


Figure 98: Event-Controlled Method Calls

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.

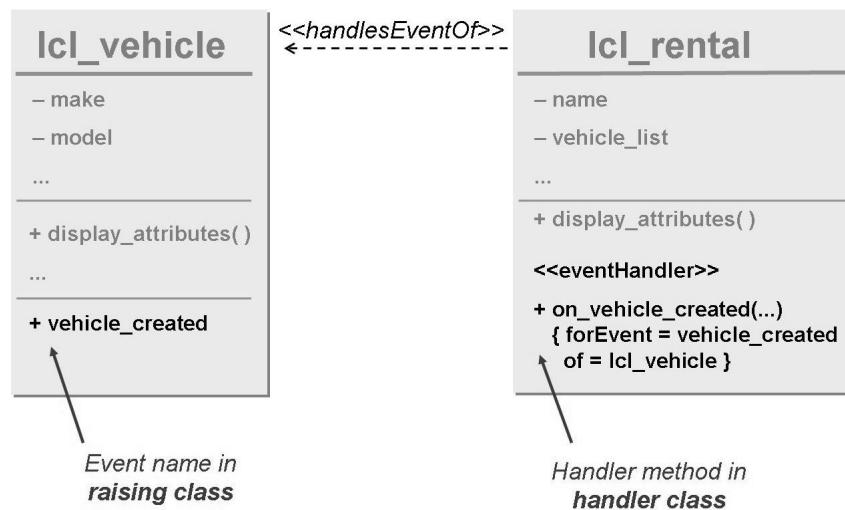


Figure 99: Event Handling in a UML Class Diagram

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.

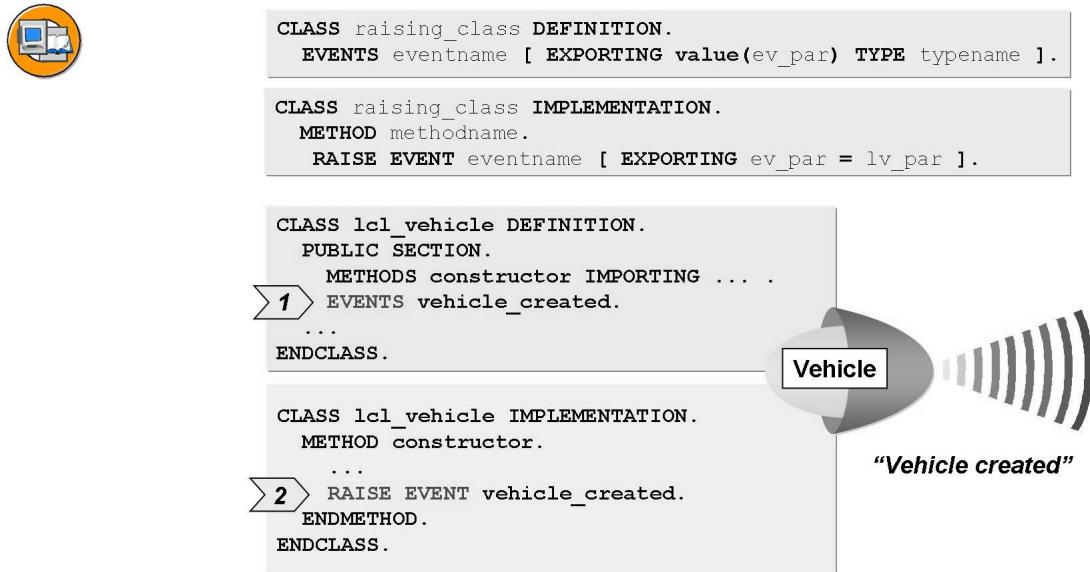


Figure 101: Defining and Triggering Events – Syntax

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**.

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 ev_par_1 .. ev_par_n [sender] ].
```

```
CLASS lcl_rental DEFINITION.
...
>3 > METHODS on_vehicle_created FOR EVENT vehicle_created
      OF lcl_vehicle
      IMPORTING sender.

ENDCLASS.
```

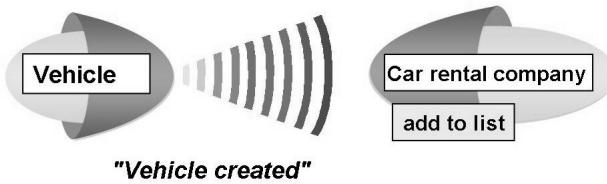


Figure 102: Handling Events – Syntax

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 nothing 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.

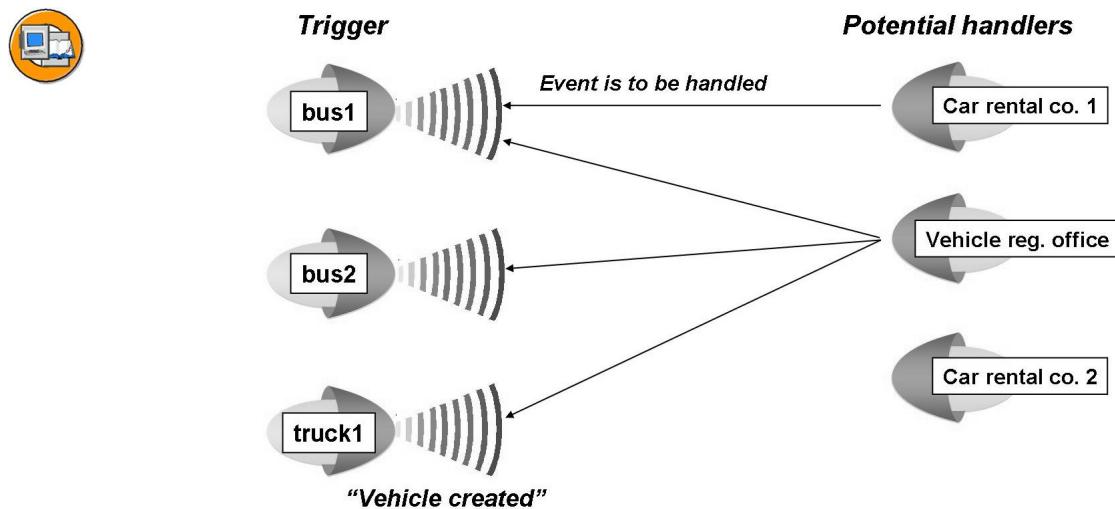


Figure 103: Registering Event Handling

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 on_vehicle_created FOR EVENT vehicle_created
OF ....
ENDCLASS.
```

```
CLASS lcl_rental IMPLEMENTATION.
METHOD constructor.
...
4 SET HANDLER me->on_vehicle_created FOR ALL INSTANCES.
ENDMETHOD.
ENDCLASS.
```

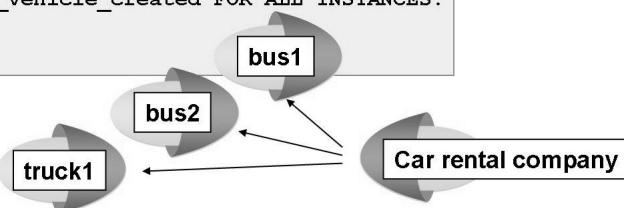


Figure 104: Registering Event Handling – Syntax

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.

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 ... .
```



Caution: 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.

With the addition ALL INSTANCES an event handler can be registered for all instances of the class that defines the instance event. This is the only way to register to objects that have not yet been created.

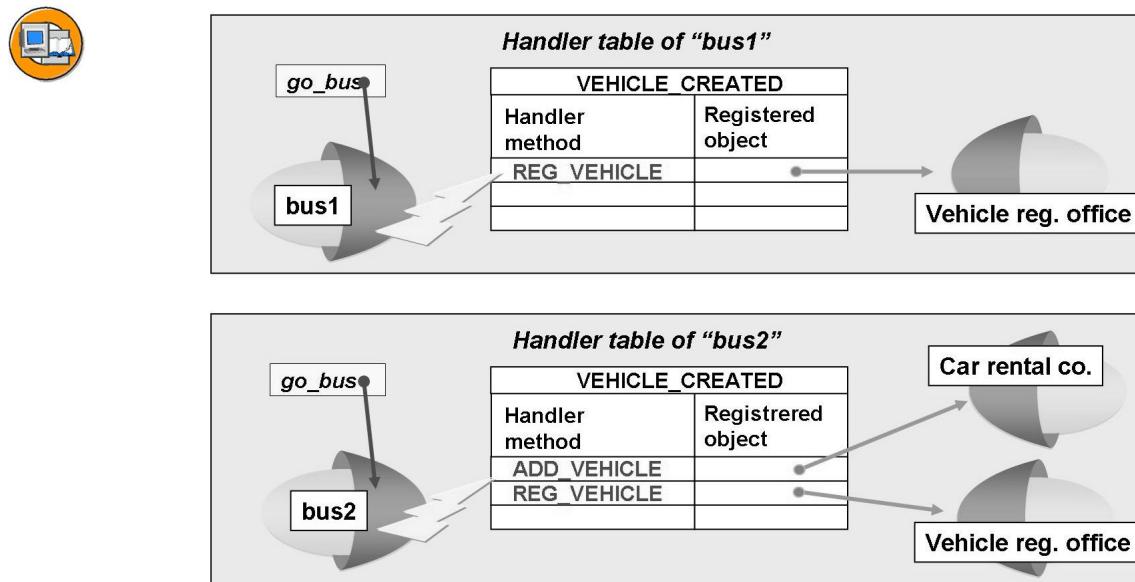


Figure 105: Registration/Deregistration: Handler Tables

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.



Hint: Objects that are registered for event handling are **not** deleted by the Garbage Collector, even if there are no remaining references to them.



Note: It is even possible to add/remove event handlers for a given event while an event handler for exactly this event is executed. If a new event handler is registered, then this event handler is added to the end of the sequence and executed when its time comes. If an existing event handler is deregistered, then this handler is deleted immediately from the event handler method sequence.



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 13: 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.

Template:

SAPBC401_INT_S3

Solution:

SAPBC401_EVE_S1

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 template program.
Identify the class that is suitable for triggering the event and the class that should be used for handling the event. Use your UML diagram if necessary.
If applicable, illustrate the relationships in your UML diagram.
2. Define the public event AIRPLANE_CREATED in class LCL_AIRPLANE. Choose a suitable method of LCL_AIRPLANE and fire the event in its implementation.
3. In class LCL_CARRIER, define a handler method for the event that you just defined (recommended name: ON_AIRPLANE_CREATED).
4. Implement the handler method so that the new airplane is added to the airplane list of the carrier.



Hint: You can do that either directly or by calling the existing method ADD_AIRPLANE. In both cases, you should make sure that the same airplane cannot be added to the list again. Therefore, either delete method ADD_AIRPLANE or make it private.

Continued on next page

5. Remove all calls of ADD_AIRPLANE from your main program.
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 could be created later.
7. Where could the SET HANDLER statement be executed in the example given? Could this also be done from the main program?

8. Observe the execution of the program in the *ABAP Debugger*.
9. 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 all calls of ADD_VEHICLE from your main program.
2. Define the public event VEHICLE_CREATED in class LCL_VEHICLE and fire the event in the implementation of the constructor..
3. In class LCL_RENTAL, define and implement a handler method for the event that you just defined (recommended name: ON_VEHICLE_CREATED).
4. Register the new handler method, so that the car rental company enters **every vehicle 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 vehicles could be created later.

5. Observe the execution of the program in the *ABAP Debugger*.

Solution 13: 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 template program.
Identify the class that is suitable for triggering the event and the class that should be used for handling the event. Use your UML diagram if necessary.
If applicable, illustrate the relationships in your UML diagram.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
2. Define the public event AIRPLANE_CREATED in class LCL_AIRPLANE. Choose a suitable method of LCL_AIRPLANE and fire the event in its implementation.
 - a) See the source code extract from the model solution.
3. In class LCL_CARRIER, define a handler method for the event that you just defined (recommended name: ON_AIRPLANE_CREATED).
 - a) See the source code extract from the model solution.
4. Implement the handler method so that the new airplane is added to the airplane list of the carrier.



Hint: You can do that either directly or by calling the existing method ADD_AIRPLANE. In both cases, you should make sure that the same airplane cannot be added to the list again. Therefore, either delete method ADD_AIRPLANE or make it private.

- a) See the source code extract from the model solution.
5. Remove all calls of ADD_AIRPLANE from your main program.
 - 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 could be created later.

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

Continued on next page

7. Where could the SET HANDLER statement be executed in the example given? Could this also be done from the main program?

Answer: Either in the constructor of LCL_CARRIER or in the main program **after** the creation of the LCL_CARRIER
but **before** the creation of the first airplane.

8. 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.
9. What would the syntax have to be in this case?

Answer: SET HANDLER go_carrier->on_airplane_created 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 all calls of ADD_VEHICLE from your main program.
 - a) See the source code extract from the model solution.
2. Define the public event VEHICLE_CREATED in class LCL_VEHICLE and fire the event in the implementation of the constructor..
 - a) See the source code extract from the model solution.
3. In class LCL_RENTAL, define and implement a handler method for the event that you just defined (recommended name: ON_VEHICLE_CREATED).
 - a) See the source code extract from the model solution.
4. Register the new handler method, so that the car rental company enters **every vehicle 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 vehicles could be created later.

- a) See the source code extract from the model solution.
5. 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.

Continued on next page

Result

Include Program BC401_EVE_S1_CARRIER

```

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

PUBLIC SECTION.

METHODS:
constructor
IMPORTING
    iv_name      TYPE string
    iv_planetype TYPE saplane-planetype
EXCEPTIONS
    wrong_planetype,
display_attributes.

CLASS-METHODS:
display_n_o_airplanes,
class_constructor.

EVENTS:
airplane_created.

PROTECTED SECTION.

CONSTANTS:
c_pos_1 TYPE i VALUE 30.

PRIVATE SECTION.

TYPES:
ty_planetypes TYPE STANDARD TABLE OF saplane
    WITH NON-UNIQUE KEY planetype.

DATA:
mv_name      TYPE string,
mv_planetype TYPE saplane-planetype,
mv_weight    TYPE saplane-weight,
mv_tankcap   TYPE saplane-tankcap.

```

Continued on next page

```

CLASS-DATA:
  gv_n_o_airplanes TYPE i,
  gt_planetypes    TYPE ty_planetypes.

CLASS-METHODS:
  get_technical_attributes
  IMPORTING
    iv_type      TYPE saplane-planetype
  EXPORTING
    ev_weight   TYPE saplane-weight
    ev_tankcap  TYPE saplane-tankcap
  EXCEPTIONS
    wrong_planetype.

ENDCLASS.                      "lcl_airplane DEFINITION

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

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

METHOD constructor.
  mv_name        = iv_name.
  mv_planetype   = iv_planetype.

  get_technical_attributes(
  EXPORTING
    iv_type = iv_planetype
  IMPORTING
    ev_weight  = mv_weight
    ev_tankcap = mv_tankcap
  EXCEPTIONS
    wrong_planetype = 1 ).

IF sy-subrc <> 0.
  RAISE wrong_planetype.
ELSE.
  gv_n_o_airplanes = gv_n_o_airplanes + 1.
  RAISE EVENT airplane_created.
ENDIF.

```

Continued on next page

```
ENDMETHOD.           "constructor

METHOD display_attributes.

...
ENDMETHOD.           "display_attributes

METHOD display_n_o_airplanes.

...
ENDMETHOD.           "display_n_o_airplanes

METHOD get_technical_attributes.

...
ENDMETHOD.           "get_technical_attributes

ENDCLASS.           "lcl_airplane IMPLEMENTATION

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

...
ENDCLASS.           "lcl_cargo_plane DEFINITION

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

...
ENDCLASS.           "lcl_cargo_plane IMPLEMENTATION

*-----*
```

Continued on next page

```

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

...
ENDCLASS.          "lcl_passenger_plane DEFINITION

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

...
ENDCLASS.          "lcl_passenger_plane IMPLEMENTATION

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

PUBLIC SECTION.

INTERFACES lif_partner.
METHODS:
constructor IMPORTING iv_name TYPE string,
display_attributes.

PRIVATE SECTION.

DATA:
mv_name      TYPE string,
mt_airplanes TYPE TABLE OF REF TO lcl_airplane.

METHODS:
on_airplane_created FOR EVENT airplane_created
                      OF lcl_airplane
                      IMPORTING sender,
display_airplanes.

ENDCLASS.          "lcl_carrier DEFINITION

```

Continued on next page

```

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

METHOD constructor.
    mv_name = iv_name.
    SET HANDLER on_airplane_created FOR ALL INSTANCES.
ENDMETHOD.                                "constructor

METHOD lif_partner~display_partner.
    display_attributes( ).
ENDMETHOD.                                "lif_partners~display_partner

METHOD on_airplane_created.
    APPEND sender TO mt_airplanes.
ENDMETHOD.                                "on_airplane_created

METHOD display_attributes.
    SKIP 2.
    WRITE: icon_flight AS ICON,
           mv_name.
    ULINE.
    ULINE.
    display_airplanes( ).
ENDMETHOD.                                "display_attributes

METHOD display_airplanes.
    DATA: lo_plane TYPE REF TO lcl_airplane.
    LOOP AT mt_airplanes INTO lo_plane.
        lo_plane->display_attributes( ).
    ENDLOOP.
ENDMETHOD.                                "display_airplanes

ENDCLASS.                                  "lcl_carrier IMPLEMENTATION

```

Include Program BC401_EVE_S1_RENTAL

```

*-----*
*      CLASS lcl_vehicle DEFINITION
*-----*
*
```

Continued on next page

```

*-----*
CLASS lcl_vehicle DEFINITION.

PUBLIC SECTION.

METHODS:

    constructor IMPORTING iv_make TYPE string,
    display_attributes.

CLASS-METHODS:
    get_count EXPORTING ev_count TYPE i.

EVENTS:
    vehicle_created.

PRIVATE SECTION.

DATA:
    mv_make TYPE string.

CLASS-DATA:
    gv_n_o_vehicles TYPE i.

ENDCLASS.          "lcl_vehicle DEFINITION

*-----*
*      CLASS lcl_vehicle IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_vehicle IMPLEMENTATION.

METHOD constructor.
    mv_make = iv_make.
    ADD 1 TO gv_n_o_vehicles.
    RAISE EVENT vehicle_created.
ENDMETHOD.          "constructor

METHOD display_attributes.
    WRITE mv_make.
ENDMETHOD.          "display_attributes

METHOD get_count.
    ev_count = gv_n_o_vehicles.

```

Continued on next page

```
ENDMETHOD.           "get_count

ENDCLASS.          "lcl_vehicle IMPLEMENTATION

*-----*
*      CLASS lcl_truck DEFINITION
*-----*
*
*-----*
CLASS lcl_truck DEFINITION INHERITING FROM lcl_vehicle.

...
ENDCLASS.          "lcl_truck DEFINITION

*-----*
*      CLASS lcl_truck IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_truck IMPLEMENTATION.

...
ENDCLASS.          "lcl_truck DEFINITION

*-----*
*      CLASS lcl_bus DEFINITION
*-----*
*
*-----*
CLASS lcl_bus DEFINITION INHERITING FROM lcl_vehicle.

...
ENDCLASS.          "lcl_bus DEFINITION

*-----*
*      CLASS lcl_bus IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_bus IMPLEMENTATION.
```

Continued on next page

```

    ...

ENDCLASS.          "lcl_bus DEFINITION

*-----*
*      CLASS lcl_rental DEFINITION
*-----*
*
*-----*
CLASS lcl_rental DEFINITION.

PUBLIC SECTION.

INTERFACES:
    lif_partner.

METHODS:
    constructor IMPORTING iv_name TYPE string,
        display_attributes.

PRIVATE SECTION.

DATA:
    mv_name TYPE string,
    mt_vehicles TYPE TABLE OF REF TO lcl_vehicle.

METHODS:
    on_vehicle_created FOR EVENT vehicle_created
        OF lcl_vehicle
        IMPORTING sender.

ENDCLASS.          "lcl_rental DEFINITION
*-----*
*      CLASS lcl_rental IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_rental IMPLEMENTATION.

METHOD lif_partner~display_partner.
    display_attributes( ).
ENDMETHOD.          "lif_partners~display_partner

METHOD constructor.

```

Continued on next page

```

mv_name = iv_name.
SET HANDLER on_vehicle_created FOR ALL INSTANCES.
ENDMETHOD.                                     "constructor

METHOD on_vehicle_created.
APPEND sender TO mt_vehicles.
ENDMETHOD.                                     "on_vehicle_created

METHOD display_attributes.
DATA:
lo_vehicle TYPE REF TO lcl_vehicle.

WRITE: / icon_transport_proposal AS ICON,
mv_name.
WRITE: / 'Here comes the vehicle list: '.
ULINE.
ULINE.

LOOP AT mt_vehicles INTO lo_vehicle.
lo_vehicle->display_attributes( ).
ENDLOOP.
ENDMETHOD.                                     "display_attributes

ENDCLASS.                                       "lcl_rental IMPLEMENTATION

```

Main Program SAPBC401_EVE_S1

```

REPORT sapbc401_eve_s1.

TYPE-POOLS icon.

INCLUDE bc401_eve_s1_agency.
INCLUDE bc401_eve_s1_carrier.
INCLUDE bc401_eve_s1_rental.

DATA:
go_vehicle  TYPE REF TO lcl_vehicle,
go_truck    TYPE REF TO lcl_truck,
go_bus      TYPE REF TO lcl_bus,
go_rental   TYPE REF TO lcl_rental,
go_agency   TYPE REF TO lcl_travel_agency,
go_carrier  TYPE REF TO lcl_carrier,
go_airplane TYPE REF TO lcl_airplane,

```

Continued on next page

```

go_cargo      TYPE REF TO lcl_cargo_plane,
go_passenger TYPE REF TO lcl_passenger_plane,
gv_count      TYPE i.

START-OF-SELECTION.
***** create travel_agency *****
CREATE OBJECT go_agency
EXPORTING
  iv_name = 'Travel&smile Travel'.

***** create rental *****
CREATE OBJECT go_rental
EXPORTING
  iv_name = 'Happy Car Rental'.

***** Insert rental company into partner list of travel agency
go_agency->add_partner( go_rental ).

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make   = 'MAN'
  iv_cargo = 45.

*   go_rental->add_vehicle( go_truck ).

***** create truck *****
CREATE OBJECT go_bus
EXPORTING
  iv_make     = 'Mercedes'
  iv_passengers = 80.

*   go_rental->add_vehicle( go_bus ).

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make   = 'VOLVO'
  iv_cargo = 48.

*   go_rental->add_vehicle( go_truck ).

***** Create Carrier *****

```

Continued on next page

```

CREATE OBJECT go_carrier
EXPORTING
    iv_name = 'Smile&Fly-Travel'.

***** Insert carrier into business partner list of travel agency
go_agency->add_partner( go_carrier ).


***** Passenger Plane ****
CREATE OBJECT go_passenger
EXPORTING
    iv_name      = 'LH BERLIN'
    iv_planetype = '747-400'
    iv_seats     = 345
EXCEPTIONS
    wrong_planetype = 1.
IF sy-subrc = 0.
*   go_carrier->add_airplane( go_passenger ).
ELSE.
    WRITE:
        / icon_failure AS ICON,
        'Wrong plane type'.
ENDIF.

***** cargo Plane ****
CREATE OBJECT go_cargo
EXPORTING
    iv_name      = 'US Hercules'
    iv_planetype = '747-200F'
    iv_cargo     = 533
EXCEPTIONS
    wrong_planetype = 1.
IF sy-subrc = 0.
*   go_carrier->add_airplane( go_cargo ).
ELSE.
    WRITE:
        / icon_failure AS ICON,
        'Wrong plane type'.
ENDIF.

***** show attributes of all partners of travel_agency ****
go_agency->display_attributes( ).
```


Exercise 14: (Optional) Events in Interfaces

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.

Template:

SAPBC401_EVE_S1

Solution:

SAPBC401_EVE_S2

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 **all** classes that implement the interface.

Continued on next page

5. Remove the method ADD_PARTNER in the class LCL_TRAVEL_AGENCY. Replace it by a handler method for the event that you just defined (suggested name: ON_PARTNER_CREATED).
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 14: (Optional) Events in Interfaces

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.
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 **all** classes that implement the interface.
 - a) See the source code extract from the model solution.
5. Remove the method ADD_PARTNER in the class LCL_TRAVEL_AGENCY. Replace it by a handler method for the event that you just defined (suggested name: ON_PARTNER_CREATED).
 - a) See the source code extract from the model solution.
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*.

Continued on next page

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

Result

Include Program BC401_EVE_S2_AGENCY

```

*-----*
*      INTERFACE lif_partners
*-----*
INTERFACE lif_partner.

METHODS:
    display_partner.

EVENTS:
    partner_created.

ENDINTERFACE.                      "lif_partners
*-----*
*      CLASS lcl_travel_agency DEFINITION
*-----*
CLASS lcl_travel_agency DEFINITION.

PUBLIC SECTION.

METHODS:
    constructor IMPORTING iv_name TYPE string,
    display_agency_partners,
    display_attributes.

PRIVATE SECTION.

DATA:
    mv_name      TYPE string,
    mt_partners TYPE TABLE OF REF TO lif_partner.

METHODS:
    on_partner_created FOR EVENT partner_created
        OF lif_partner
        IMPORTING sender.

ENDCLASS.                          "lcl_travel_agency DEFINITION
*-----*

```

Continued on next page

```

*      CLASS lcl_travel_agency IMPLEMENTATION
*-----
CLASS lcl_travel_agency IMPLEMENTATION.

METHOD display_attributes.
  WRITE: / icon_private_files AS ICON,
          'Travel Agency:'(007), mv_name.
  SKIP.
  display_agency_partners( ).
ENDMETHOD.           "display_attributes

METHOD display_agency_partners.
  DATA:
    lo_partner TYPE REF TO lif_partner.

  WRITE 'Here are the partners of the travel agency:'(008).
  ULINE.
  LOOP AT mt_partners INTO lo_partner.
    lo_partner->display_partner( ).
  ENDLOOP.

ENDMETHOD.           "display_agency_partners

METHOD constructor.
  mv_name = iv_name.
  SET HANDLER on_partner_created FOR ALL INSTANCES.
ENDMETHOD.           "constructor

METHOD on_partner_created.
  APPEND sender TO mt_partners.
ENDMETHOD.           "on_partner_created

ENDCLASS.           "lcl_travel_agency IMPLEMENTATION

```

Include Program BC401_EVE_S2_RENTAL

```

*-----
*      CLASS lcl_vehicle DEFINITION
*-----
CLASS lcl_vehicle DEFINITION.

...
ENDCLASS.           "lcl_vehicle DEFINITION

```

Continued on next page

```
*-----*
*      CLASS lcl_vehicle IMPLEMENTATION
*-----*
CLASS lcl_vehicle IMPLEMENTATION.

...
ENDCLASS.          "lcl_vehicle IMPLEMENTATION

*-----*
*      CLASS lcl_truck DEFINITION
*-----*
CLASS lcl_truck DEFINITION INHERITING FROM lcl_vehicle.

...
ENDCLASS.          "lcl_truck DEFINITION

*-----*
*      CLASS lcl_truck IMPLEMENTATION
*-----*
CLASS lcl_truck IMPLEMENTATION.

...
ENDCLASS.          "lcl_truck DEFINITION

*-----*
*      CLASS lcl_bus DEFINITION
*-----*
CLASS lcl_bus DEFINITION INHERITING FROM lcl_vehicle.

...
ENDCLASS.          "lcl_bus DEFINITION

*-----*
*      CLASS lcl_bus IMPLEMENTATION
*-----*
CLASS lcl_bus IMPLEMENTATION.

...
ENDCLASS.          "lcl_bus DEFINITION
```

Continued on next page

```

*-----*
*      CLASS lcl_rental DEFINITION
*-----*
CLASS lcl_rental DEFINITION.

...
ENDCLASS.                      "lcl_rental DEFINITION
*-----*
*      CLASS lcl_rental IMPLEMENTATION
*-----*
CLASS lcl_rental IMPLEMENTATION.

METHOD lif_partner~display_partner.
    display_attributes( ).
ENDMETHOD.                      "lif_partners~display_partner

METHOD constructor.
    mv_name = iv_name.
    SET HANDLER on_vehicle_created FOR ALL INSTANCES.
    RAISE EVENT lif_partner~partner_created.
ENDMETHOD.                      "constructor

METHOD on_vehicle_created.
    APPEND sender TO mt_vehicles.
ENDMETHOD.                      "on_vehicle_created

METHOD display_attributes.

...
ENDMETHOD.                      "display_attributes

ENDCLASS.                      "lcl_rental IMPLEMENTATION

```

Include Program BC401_EVE_S2_CARRIER

```

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

...

```

Continued on next page

```
ENDCLASS.          "lcl_airplane DEFINITION

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

...

ENDCLASS.          "lcl_airplane IMPLEMENTATION

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

...

ENDCLASS.          "lcl_cargo_plane DEFINITION

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

...

ENDCLASS.          "lcl_cargo_plane IMPLEMENTATION

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

...

ENDCLASS.          "lcl_passenger_plane DEFINITION

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

Continued on next page

```
...  
  
ENDCLASS.          "lcl_passenger_plane IMPLEMENTATION  
  
*-----*  
*      CLASS lcl_carrier DEFINITION  
*-----*  
CLASS lcl_carrier DEFINITION.  
  
...  
  
ENDCLASS.          "lcl_carrier DEFINITION  
  
*-----*  
*      CLASS lcl_carrier IMPLEMENTATION  
*-----*  
CLASS lcl_carrier IMPLEMENTATION.  
  
METHOD constructor.  
    mv_name = iv_name.  
    SET HANDLER on_airplane_created FOR ALL INSTANCES.  
    RAISE EVENT lif_partner~partner_created.  
ENDMETHOD.          "constructor  
  
METHOD lif_partner~display_partner.  
    display_attributes( ).  
ENDMETHOD.          "lif_partners~display_partner  
  
METHOD on_airplane_created.  
    APPEND sender TO mt_airplanes.  
ENDMETHOD.          "on_airplane_created  
  
METHOD display_attributes.  
  
...  
  
ENDMETHOD.          "display_attributes  
  
METHOD display_airplanes.  
  
...  
  
ENDMETHOD.          "display_airplanes
```

Continued on next page

```

ENDCLASS.                                     "lcl_carrier IMPLEMENTATION

Main Program SAPBC401_EVE_S2

REPORT  sapbc401_eve_s2.

TYPE-POOLS icon.

INCLUDE bc401_eve_s2_agency.
INCLUDE bc401_eve_s2_carrier.
INCLUDE bc401_eve_s2_rental.

DATA:
go_vehicle      TYPE REF TO lcl_vehicle,
go_truck        TYPE REF TO lcl_truck,
go_bus          TYPE REF TO lcl_bus,
go_rental        TYPE REF TO lcl_rental,
go_passenger    TYPE REF TO lcl_passenger_plane,
go_cargo         TYPE REF TO lcl_cargo_plane,
go_carrier       TYPE REF TO lcl_carrier,
go_agency        TYPE REF TO lcl_travel_agency.

START-OF-SELECTION.
*****create travel_agency *****
CREATE OBJECT go_agency
EXPORTING
iv_name = 'Travel&Smile Travel'.

*****create rental *****
CREATE OBJECT go_rental
EXPORTING
iv_name = 'Happy Car Rental'.
* go_agency->add_partner( go_rental ).

*****create truck *****
CREATE OBJECT go_truck
EXPORTING
iv_make  = 'MAN'
iv_cargo = 45.

*****create truck *****

```

Continued on next page

```

CREATE OBJECT go_bus
EXPORTING
  iv_make      = 'Mercedes'
  iv_passengers = 80.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make    = 'VOLVO'
  iv_cargo = 48.

***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly-Travel'.
* go_agency->add_partner( go_rental ).

***** Passenger Plane *****
CREATE OBJECT go_passenger
EXPORTING
  iv_name      = 'LH BERLIN'
  iv_planetype = '747-400'
  iv_seats     = 345
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc <> 0.
  WRITE:
    / icon_failure AS ICON,
    'Wrong plane type'.
ENDIF.

***** cargo Plane *****
CREATE OBJECT go_cargo
EXPORTING
  iv_name      = 'US Hercules'
  iv_planetype = '747-200F'
  iv_cargo     = 533
EXCEPTIONS
  wrong_planetype = 1.

IF sy-subrc <> 0.
  WRITE:
    / icon_failure AS ICON,
    'Wrong plane type'.
ENDIF.

```

Continued on next page

```
***** show attributes of all partners of travel_agency *****
go_agency->display_attributes( ).
```



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 have a class X that inherits from class Y. After an up-cast a reference variable that is statically typed TYPE REF TO Y points to an instance of class X. What components of class X can you access with this reference variable?

Choose the correct answer(s).

- A Components defined in class X
- B Components inherited from class Y
- C Components redefined in class X
- D Components defined in class X and redefined in its subclasses

4. Suppose you have the same class X that inherits from class Y. After a down-cast a reference variable that statically typed TYPE REF TO X points to an instance of class X. Which of the following components of class X can you access with this reference variable?

Choose the correct answer(s).

- A Components defined in class X
- B Components inherited from class Y
- C Components redefined in class X
- D Components defined in class X and redefined in its subclasses

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 have a class X that inherits from class Y. After an up-cast a reference variable that is statically typed TYPE REF TO Y points to an instance of class X. What components of class X can you access with this reference variable?

Answer: B, C

4. Suppose you have the same class X that inherits from class Y. After a down-cast a reference variable that statically typed TYPE REF TO X points to an instance of class X. Which of the following components of class X can you access with this reference variable?

Answer: A, B, C, D

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.

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Internal Use SAP Partner Only

International Use SAP Partner Only

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 probably 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 the 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 ways 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
- Create a simple ALV Grid Control
- React to a double-click on an ALV Grid Control with a handler method
- Explain the fundamentals of the BAdI technology and implement a BAdI
- Define abstract classes
- Define abstract methods
- Define final classes
- Define final methods
- Understand the special use of public read-only attributes
- Understand the term “navigation method”
- Chain functional methods
- Limit the visibility of the constructor

- Understand the term “factory method”
- Explain the “singleton pattern”
- Define friendship relationships between classes
- Create persistent objects using the standard technique
- Load persistent objects using the standard technique
- Access components of persistent objects
- Create an object-oriented transaction (transaction code type)

Unit Contents

Lesson: Global Classes and Interfaces	245
Procedure: Importing Local Classes and Interfaces	258
Procedure: Moving the Method Definition of a Global Class to an Implemented Interface	263
Exercise 15: Global Classes	275
Exercise 16: Global Interfaces	283
Exercise 17: (Optional) Refactoring Assistant	293
Exercise 18: ALV Grid Control	297
Exercise 19: (Optional) Second ALV Grid Control in a Popup	305
Lesson: Special Object-Oriented Programming Techniques	316
Exercise 20: (Optional) Abstract Classes, Final Classes and Singleton Classes	331
Exercise 21: (Optional) Factory Classes and Friendship Relationships	343
Lesson: Persistent Objects and OO-Transactions	355
Procedure: Creating Persistent Classes	359
Procedure: Creating an OO Transaction	361

Lesson: Global Classes and Interfaces

Lesson Overview

This lesson introduces you to a new tool in the *ABAP Workbench*: The *Class Builder*, which was integrated 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.

We will demonstrate the use of global standard classes using the powerful example of the SAP Grid Control.



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
- Create a simple ALV Grid Control
- React to a double-click on an ALV Grid Control with a handler method
- Explain the fundamentals of the BAdI technology and implement a BAdI

Business Example

You want to develop your classes and interfaces as actively integrated Repository objects.

Creating and Testing Global Classes

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 Quickinfo (explanatory text that appears when you place the cursor over an icon and leave it there for a short time).

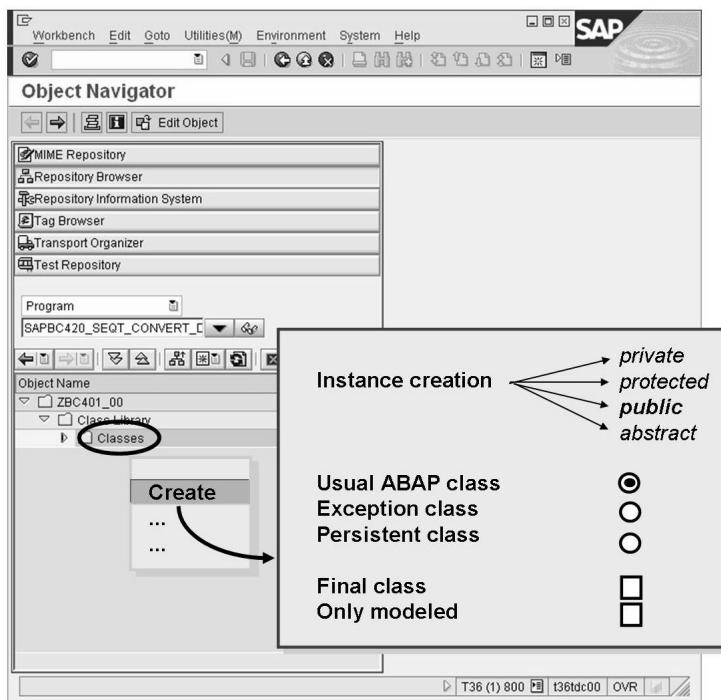


Figure 106: Creating Global Classes in the Object Navigator

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, use of the Quickinfo is therefore recommended.

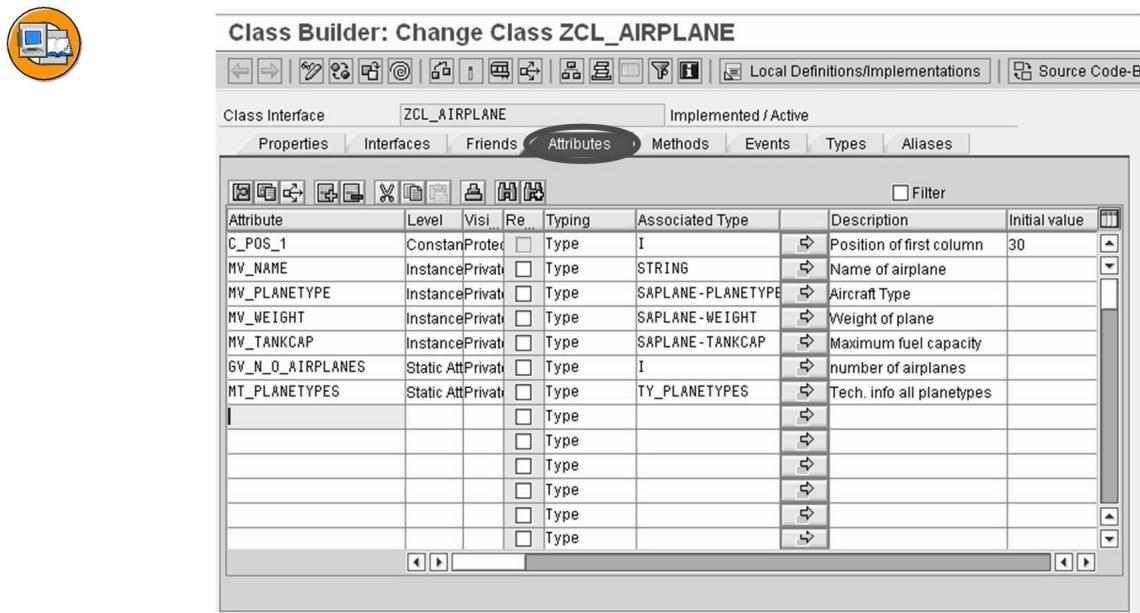


Figure 107: Definition of Attributes

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.

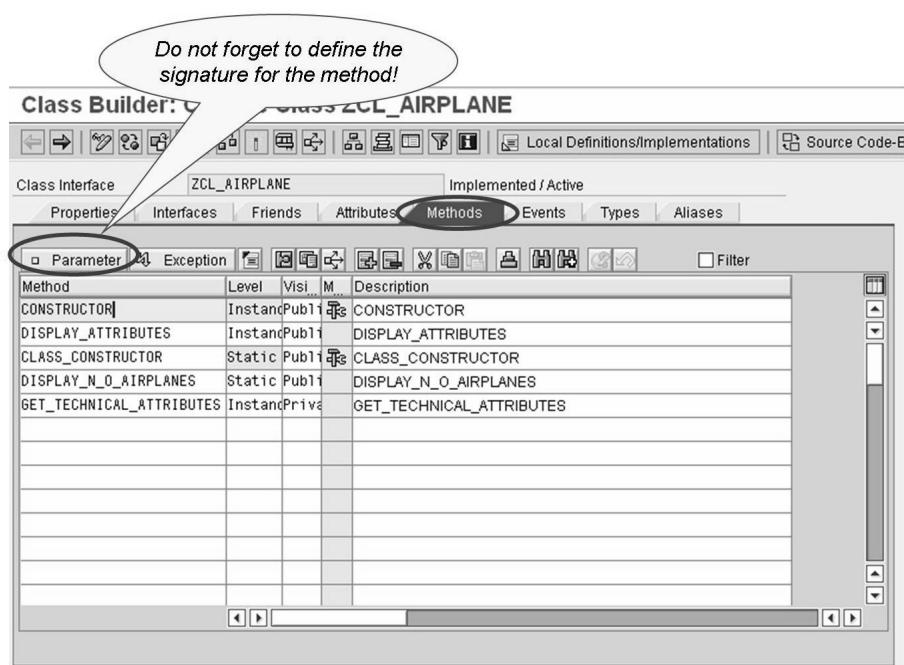


Figure 108: Definition of Methods

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 appropriately.



Hint: Methods can be transported separately.

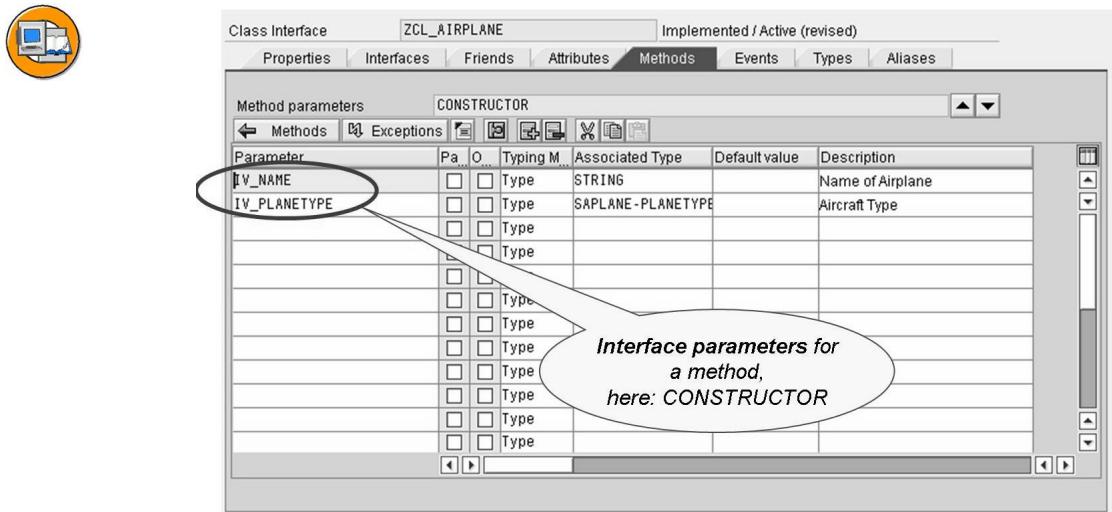


Figure 109: Definition of Method Signatures

In the method list, select a method and choose the *Parameter* button to go to the 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.

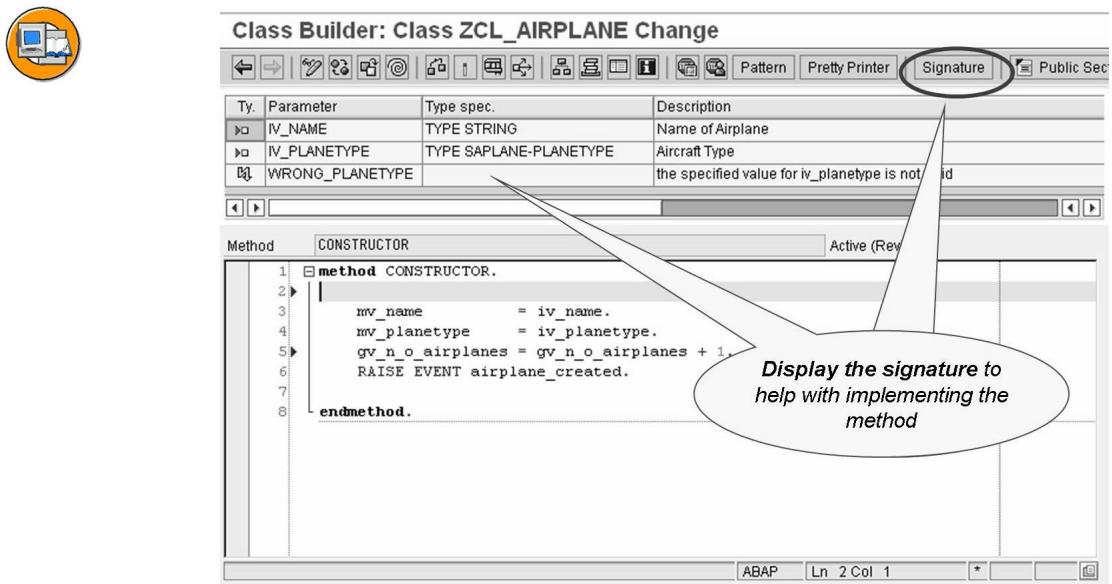


Figure 110: Implementation of Methods

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.

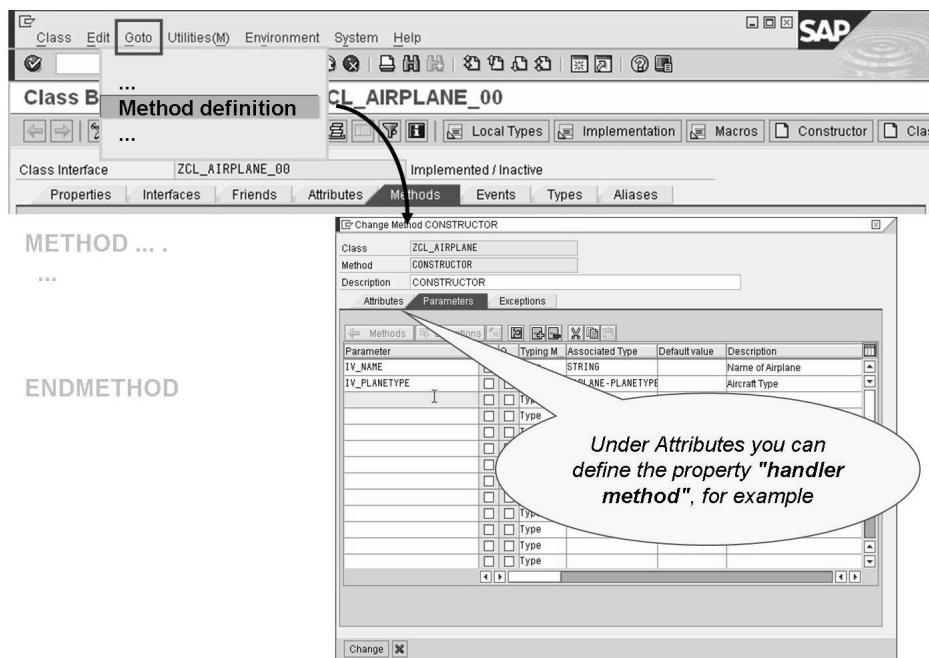


Figure 111: Displaying the Method Definition

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 by way of the *Properties* tab header in the dialog box.

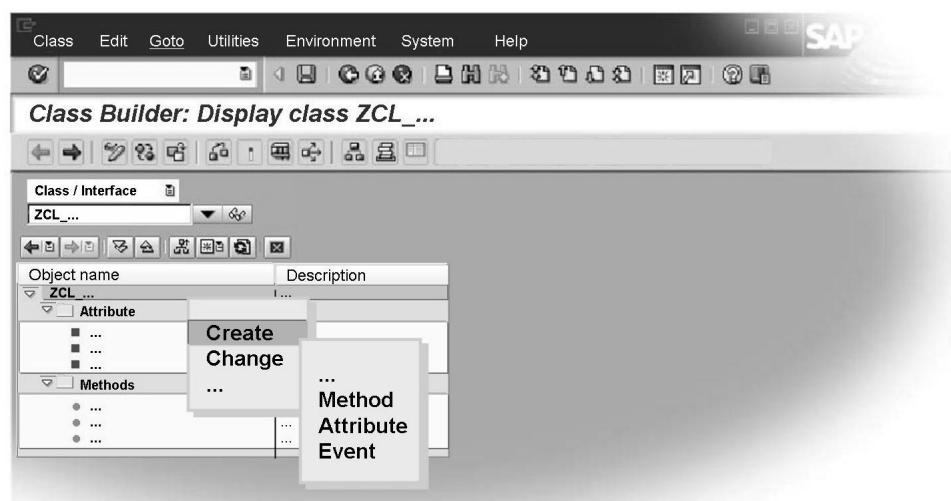


Figure 112: Definition of Components Using the Navigation Area

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 before.



Hint: Print selected portions of the source text using *Class → Print* or *Method → Print*.

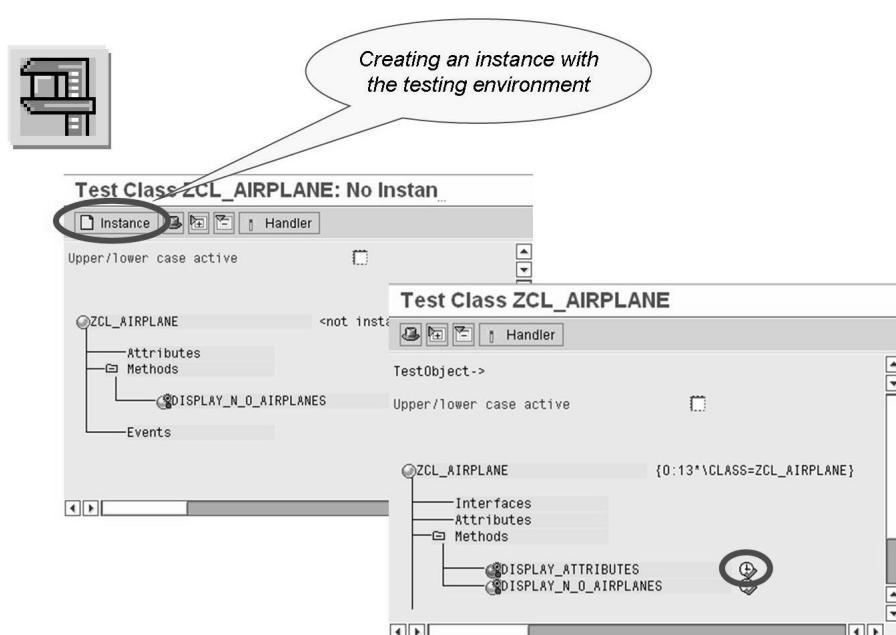


Figure 113: The Class Builder Testing Environment

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 *Call 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.

Using Global Classes in the ABAP Editor

Like other repository objects, global classes and interfaces are added in the navigation area of the OBJECT-NAVIGATOR.

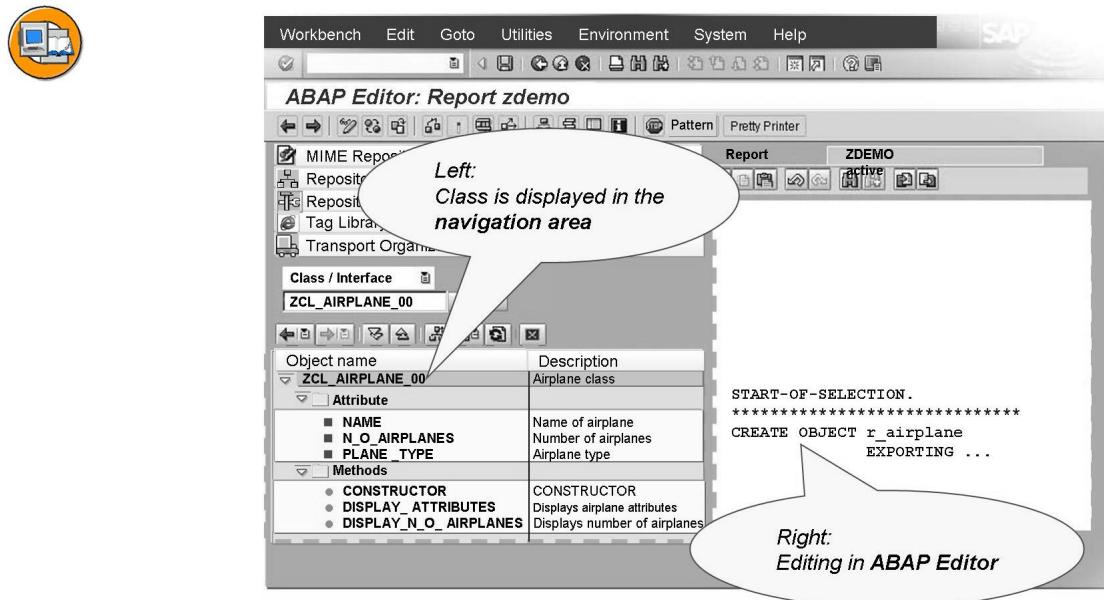


Figure 114: Separation of Navigation and Editing Areas in the Object Navigator.

This way, the previously discussed advantages also apply to working with global classes and interfaces.

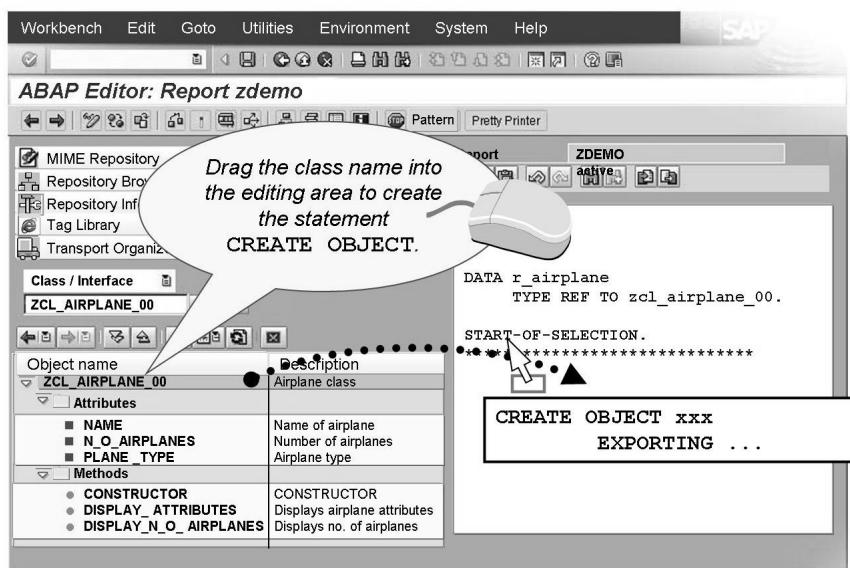


Figure 115: Object Instantiation Using Drag&Drop

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 click the *Pattern* button. The CREATE-OBJECT statement is under *ABAP Objects Pattern*. You can generate the statement using the input help.

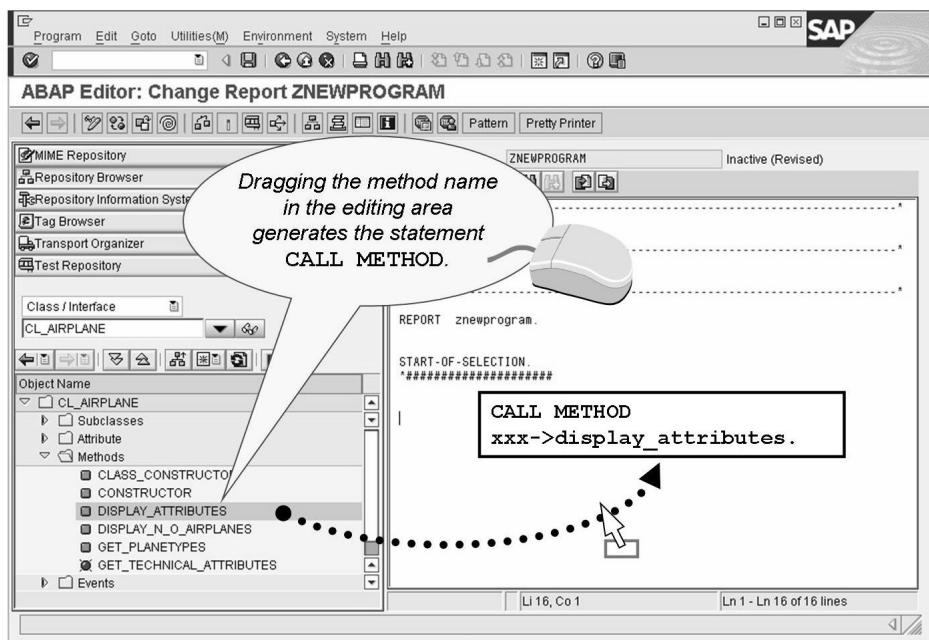


Figure 116: Method Calls Using Drag&Drop

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 click 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. Choose *Utilities → Settings → Pattern* and select the checkbox *Functional Writing Style for Call Method*.

Global Classes and Inheritance

You set out the inheritance relationships between global classes on the *Properties* tab.

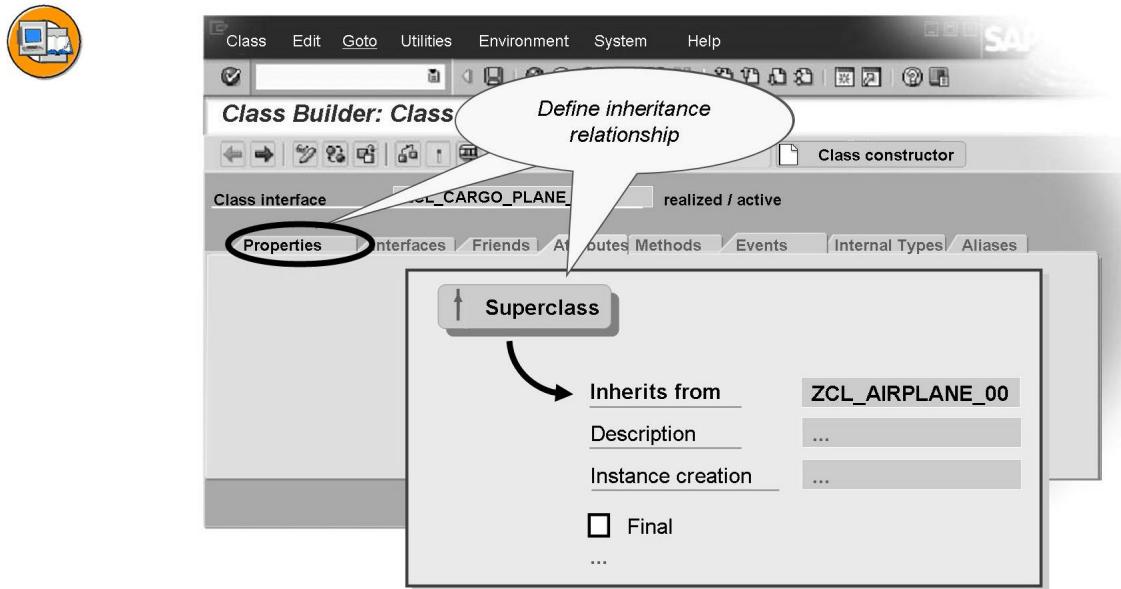


Figure 117: Defining an Inheritance Relationship

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.

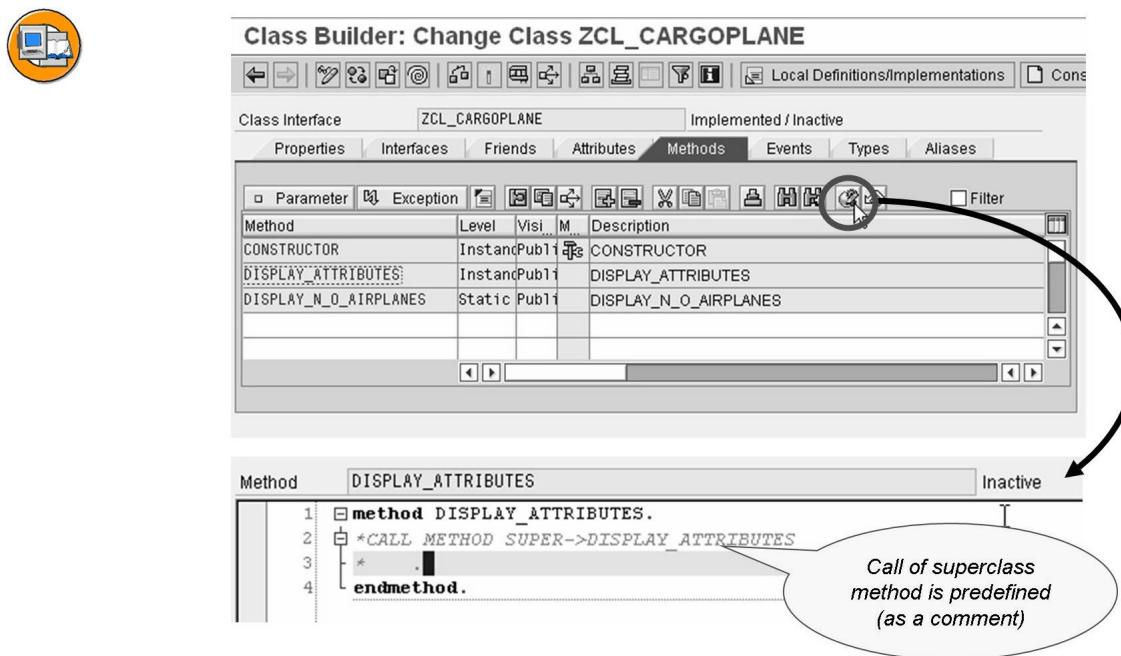


Figure 118: Redefining an Inherited Method

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.

Creating and Implementing Global Interfaces

You create global interfaces in a similar way to how you create 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.

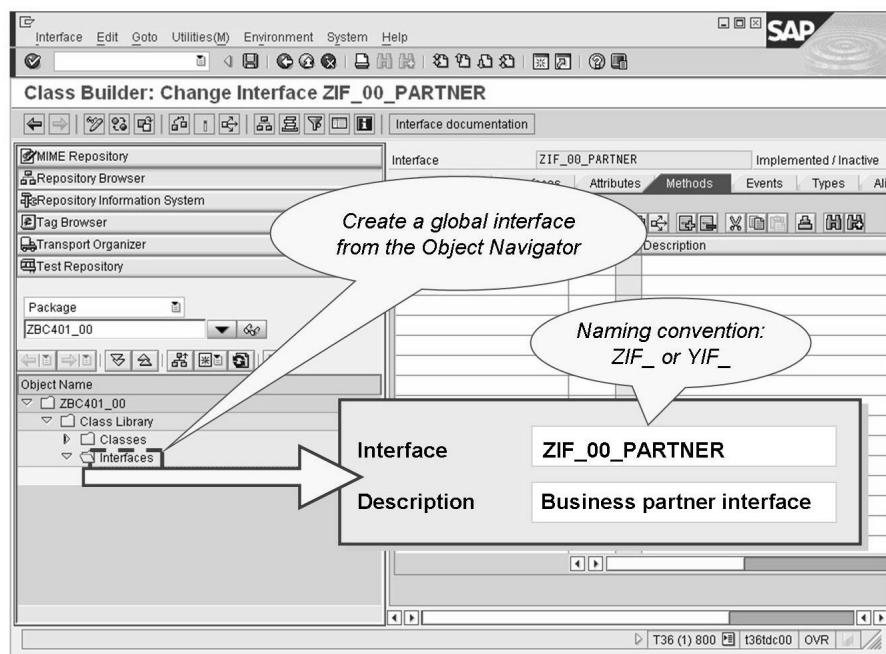


Figure 119: Defining Global Interfaces

If you want to include a global interface in your global class, you must enter the name of the interface on the *Interfaces* tab.

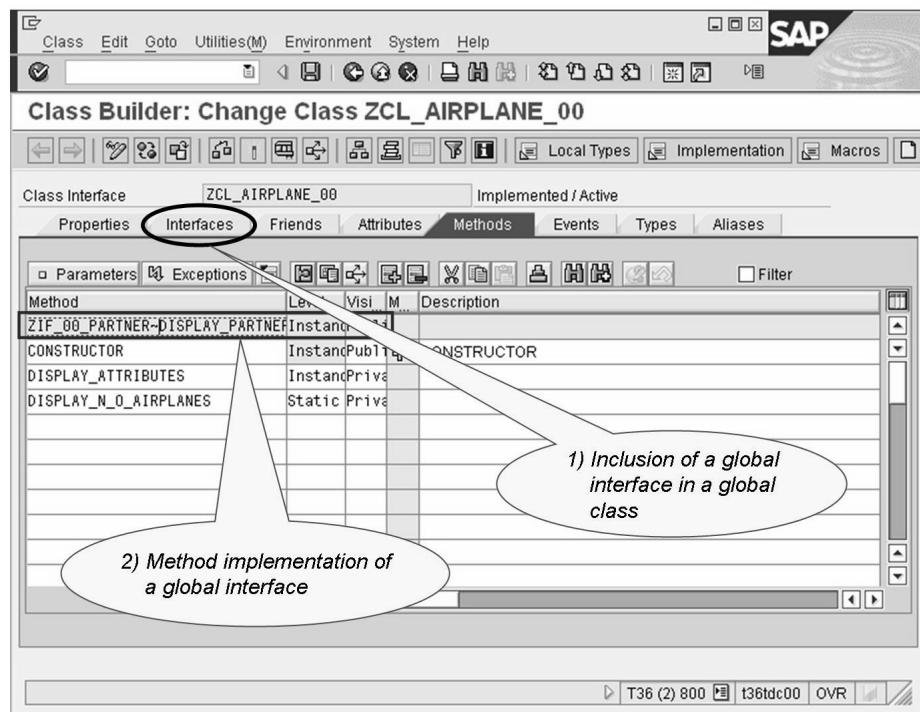


Figure 120: Including Global Interfaces

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 in a method DISPLAY_PARTNER. You can implement the method simply by double-clicking on the method name.



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 click the *Import* button.

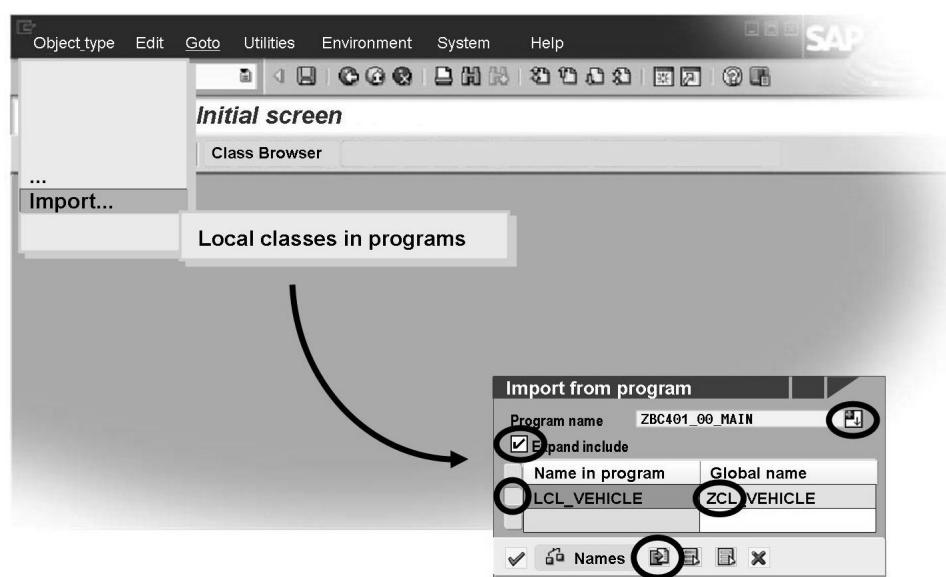


Figure 121: Importing a Local Program Class

Other Functions of the Class Builder

Besides the functions already discussed, the *Class Builder* provides many more. A few of them should at least be mentioned here.

Generate UML Diagrams for Global Classes (New in SAP NW 7.0 EhP2)

As of *SAP NetWeaver 7.0 EhP2*, the *Class Builder* can generate UML Diagrams for existing coding. This function is restricted to global classes (no local classes) but can include objects from *ABAP Dictionary*.

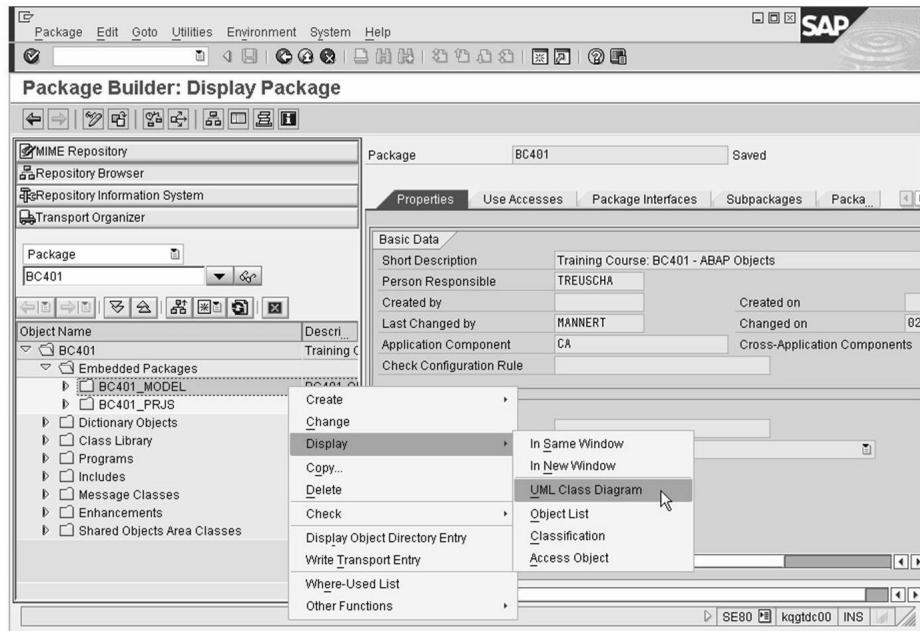


Figure 122: Generate a UML Diagram for Package

The figure explains how to start the functionality for a complete package. On a pop-up window you can add a selection of objects and specify which details you want to see on the diagram.

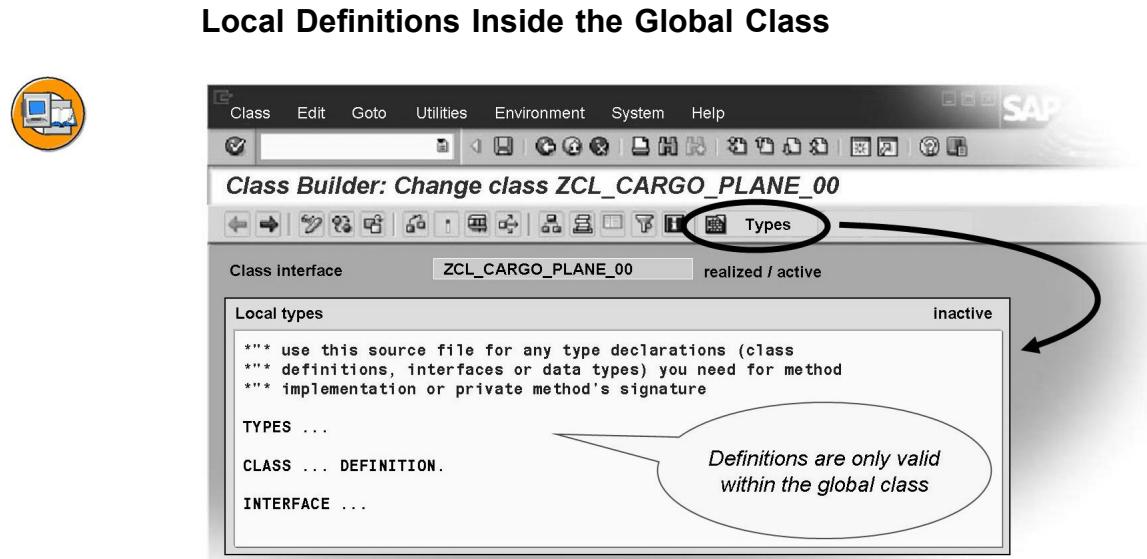


Figure 123: Defining a Local Type

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*).

Sort and Filter Display of Class Components

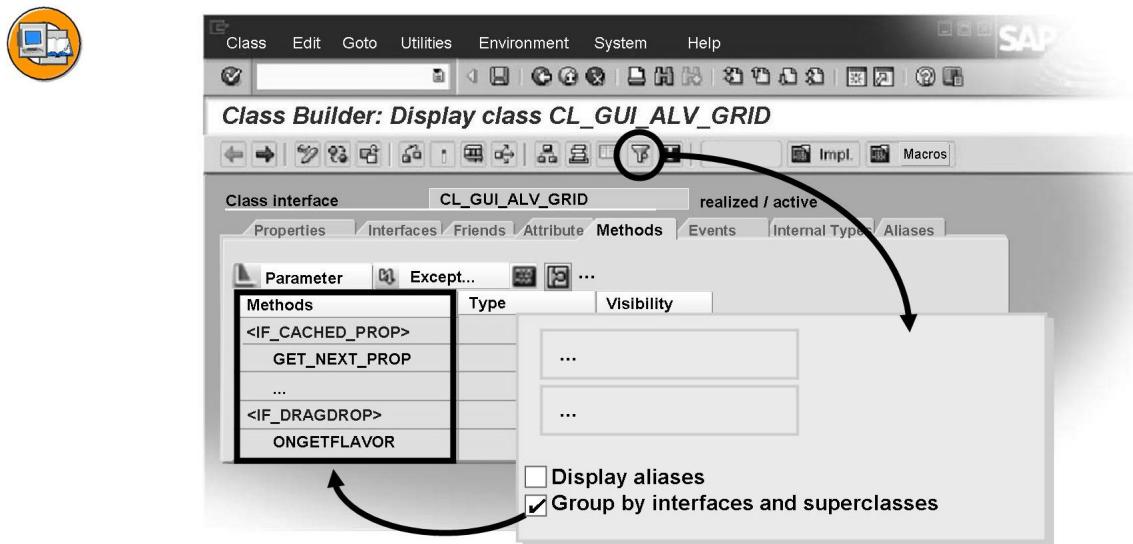


Figure 124: Structured Display of Inherited Components

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.

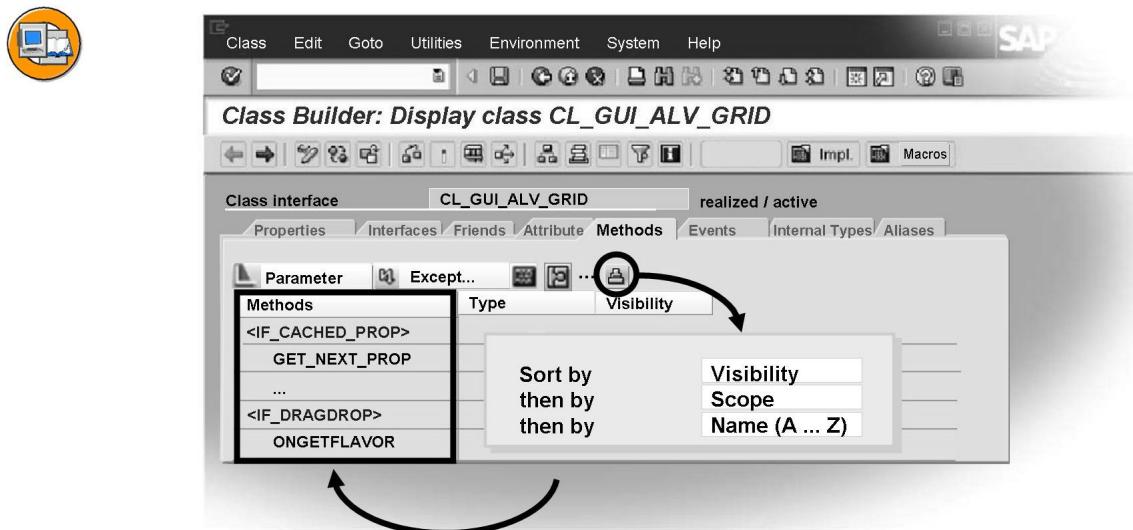


Figure 125: Sorting the Component Display of Global Classes

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.

The Refactoring Assistant

In an ideal world, all classes, interfaces, and the associations between them would be modeled completely using UML diagrams before developers began 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&Drop dialogs.

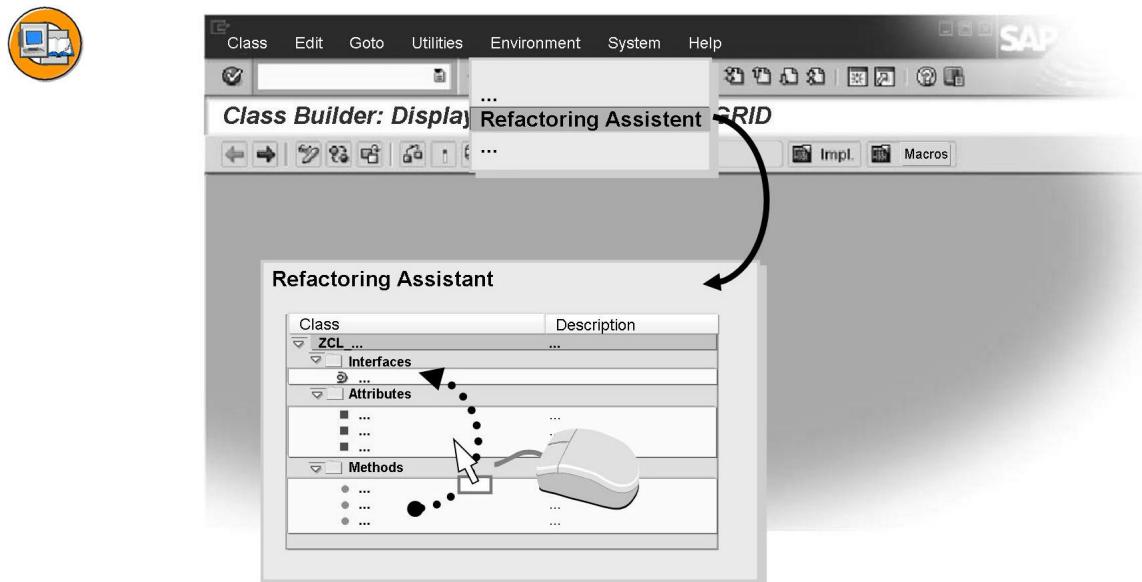


Figure 126: Working with the Refactoring Assistant



Hint: The *Tool Info* button opens the *SAP Library* article about the Refactoring Assistant. There you can also find the descriptions of the tool's other abilities.

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. Choose *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 calling the method in the source code.
7. Activate both the class **and** the interface.

Example of Object-Oriented ABAP Programming: ALV GRID CONTROL

The following topics will illustrate possible application areas of object-oriented ABAP programming and the use of SAP standard classes. As examples, there will be brief discussions of the ALV Grid Control (ALV stands for Advanced List Viewer) in the context of the Control Framework and the Business Add-Ins.

The SAP Control Framework is a collection of global classes and interfaces that you can use to add SAP GUI controls in your ABAP Objects programs, **regardless of the platform**. We will show you how to use SAP standard classes effectively with reference to the widely used ALV Grid Control.



Tool for displaying non-hierarchical lists

ID	No.	Capacity	Fre	Fre	Booking Total
AA	64	280	e	271	133.691,45
AA	64	280		267	132.346,44
AA	64	280		271	135.383,19
AA	64	280		268	132.862,67
AA	64	280		7	3.616,14
AA	64	280		11	4.990,70
AA	64	280		5	1.966,67
AA	64	280		0	0,00
AA	64	280		0	0,00
AA	64	280		1	401,79
AZ	555	220		213	48.788,20
AZ	555	220		202	46.595,95
AZ	555	220		205	47.483,95
AZ	555	220		208	48.333,10
AZ	555	220		214	48.175,85

Figure 127: The ALV Grid Control

The ALV Grid Control is a tool that you can use to display non-hierarchical lists in a standardized form. The list data is displayed in tables. It is very easy to work with as only very few programming steps have to be carried out.

The ALV Grid Control contains a number of interactive standard functions that users of lists often need, for example, print, export, and so on). As a developer, you have the option of hiding these standard functions. When required, you can also adapt the implementations to fit the needs of your application. You can also add your own functions to the application toolbar.

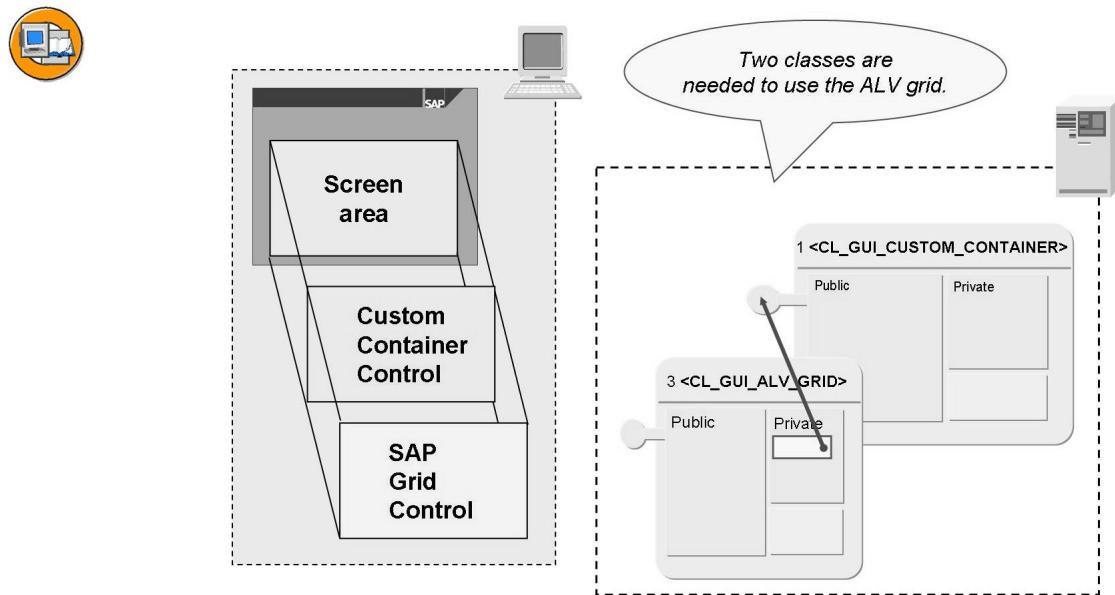


Figure 128: Including an ALV Grid Control Instance in a Dialog Program

The technical connection between the screen and application control is provided by Container Controls. Generally, application controls (ALV Grid Control, Tree Control, Picture Control and so on) are always embedded in this Container Control, which in turn is connected to the screen.

There are different types of container controls. However, they all incorporate fundamental control functions, such as scroll bars.

Let us take a common case as an example: Placing an ALV Grid Control in a screen area with a fixed size.

You need to create an instance of each of the global classes `CL_GUI_CUSTOM_CONTAINER` and `CL_GUI_ALV_GRID`.

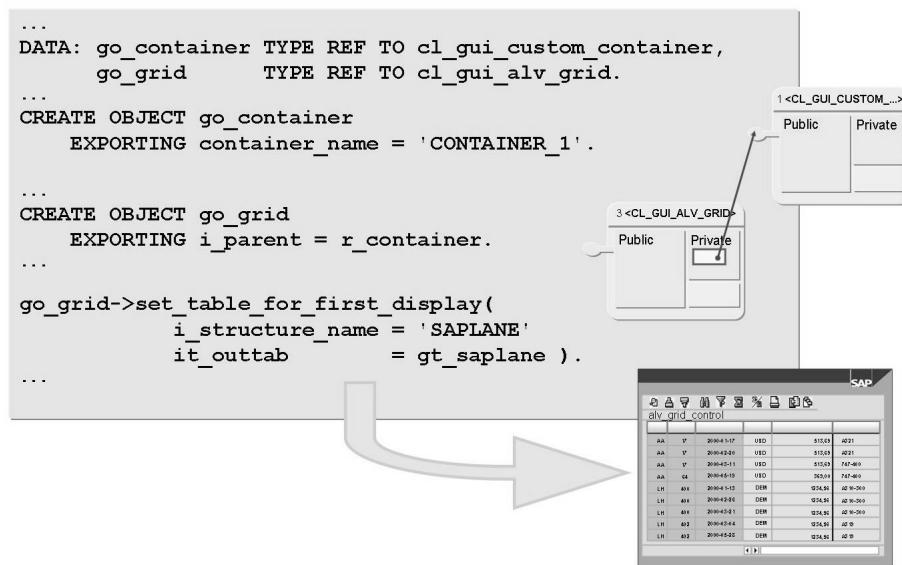


Figure 129: Displaying Application Data Using an ALV Grid Instance

For normal display, the following programming steps are sufficient:

1. Use the fullscreen editor of the Screen Painter to define a custom control area on your screen.
2. Create an instance of the class CL_GUI_CUSTOM_CONTAINER and transfer the name of the custom control area to the constructor.
3. Create an instance of the class CL_GUI_ALV_GRID and transfer the reference to the custom control instance to the constructor.
4. Call the method SET_TABLE_FOR_FIRST_DISPLAY of the grid control instance and transfer the internal standard table to it.

If this table has a global row type, you can transfer the name of this global structure to the same method. Then the Grid Control instance automatically creates the field catalog.

If you change the contents of the internal tables while the program is running, you only need to call the REFRESH_TABLE_DISPLAY method in the relevant dialog step to refresh the display.

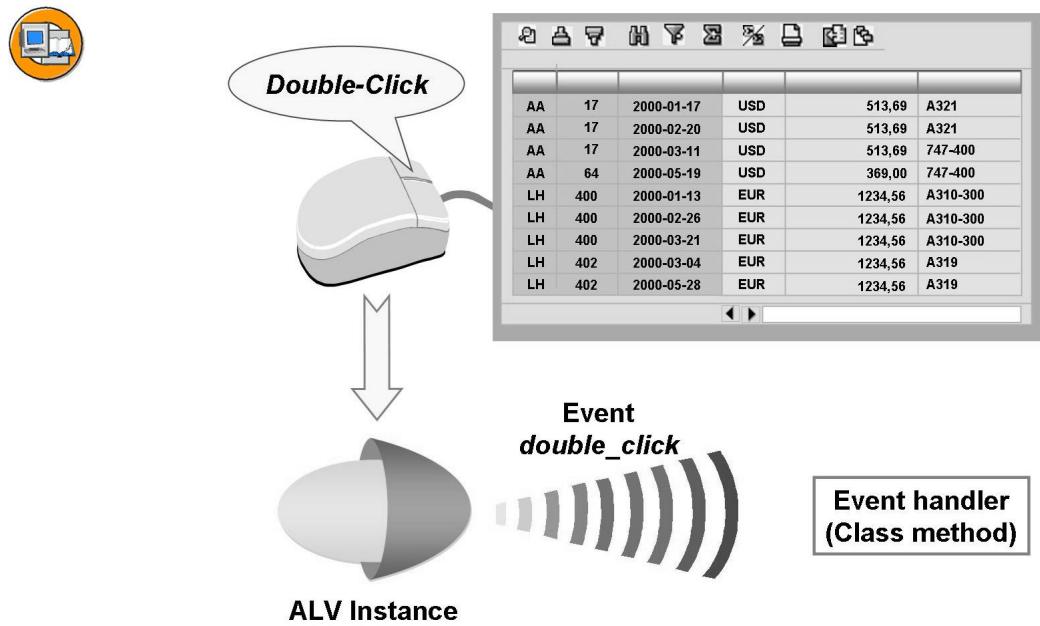


Figure 130: ALV Grid Control – Double-Click

An ALV Grid Control can react to the user's double-click. As a possible reaction, you can trigger a follow-on processing step in which additional information is displayed, for instance. In the example displayed, the booking data for the individual flight customers, or data on the aircraft, could also be displayed for the flight data. Technically, this is executed by catching the DOUBLE_CLICK event with a handler method.

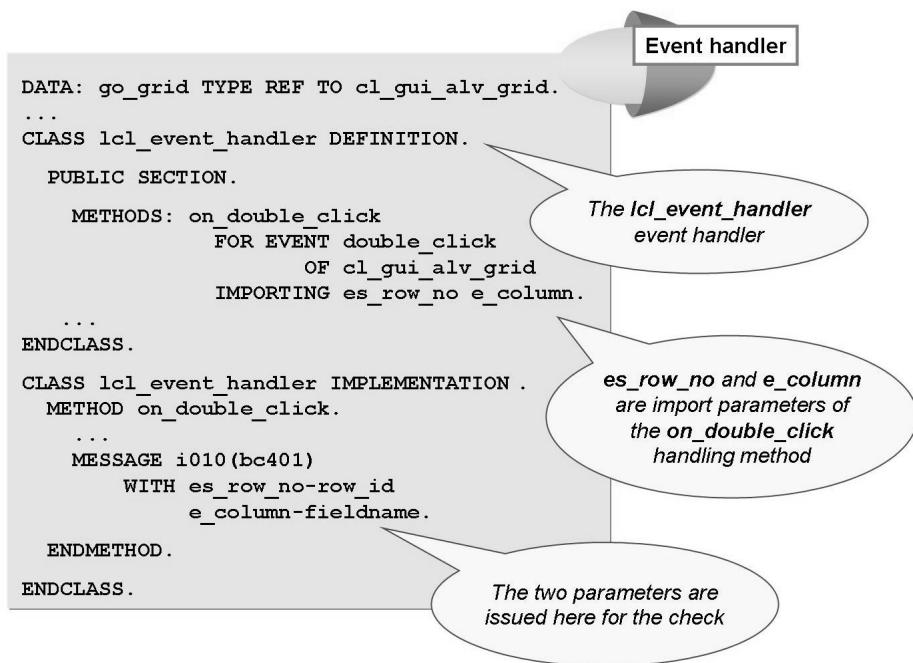


Figure 131: ALV Grid Control - Reacting to a Double-Click

A handler method can either be a class method (static method) or an instance method of an object. If a class method is defined as a handler method, no object of the handled class needs to be instantiated to use the method.

To create a handler object for an event, first define a class. This class has a public method (in the PUBLIC SECTION) that can react to an event. During the implementation of the handler method, define the source text that should be run when the event is triggered. The method receives the information delivered by the event from the mouse position when you double-click, and creates an information message in the control example that displays the row and field of the mouse-click.

→ **Note:** Extensive documentation about the SAP Control Framework is available. The *SAP Library* contains comprehensive descriptions of all classes and a complete tutorial.

The *Object Navigator* also contains the **Enjoy Demo Center**, which makes it very easy to use standard template programs. You can find it under *Environment → Examples → Control Examples*.

Individual elements of the SAP Control Framework are also covered in other courses.

SAP also offers a separate comprehensive course about dialog programming with the EnjoySAP controls. It covers all aspects, including complex programming techniques like Drag&Drop functions.

Example of object-oriented ABAP programming: Business Add-Ins

Unlike Customer Exits, the Business Add-Ins of the changed software delivery process are taken into account here. Generally, the typical delivery process does not only consist of providers and users, rather several intermediate providers may be involved in the delivery chain for software. So for example, an SAP application program can be extended with an industry solution (IBU = Industrial Business Unit); the same program could then again be extended by the customer.

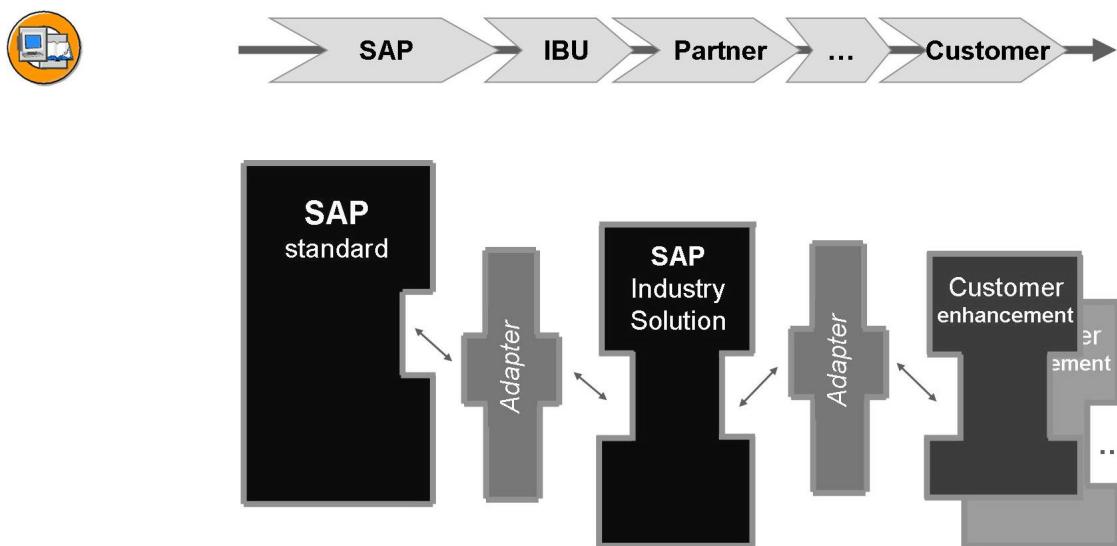


Figure 132: BAdI – Basics

With a Business Add-In, or BAdI for short, an SAP application program provides the enhancement option through an interface and an adapter class that implements the interface.

The interface can then be implemented by several possible users in the delivery chain. Multiple implementations are therefore also possible: Once a BAdI has been implemented, it can also be implemented by users who appear further to the right in the chain. Furthermore, an implementation itself could again offer a BAdI.

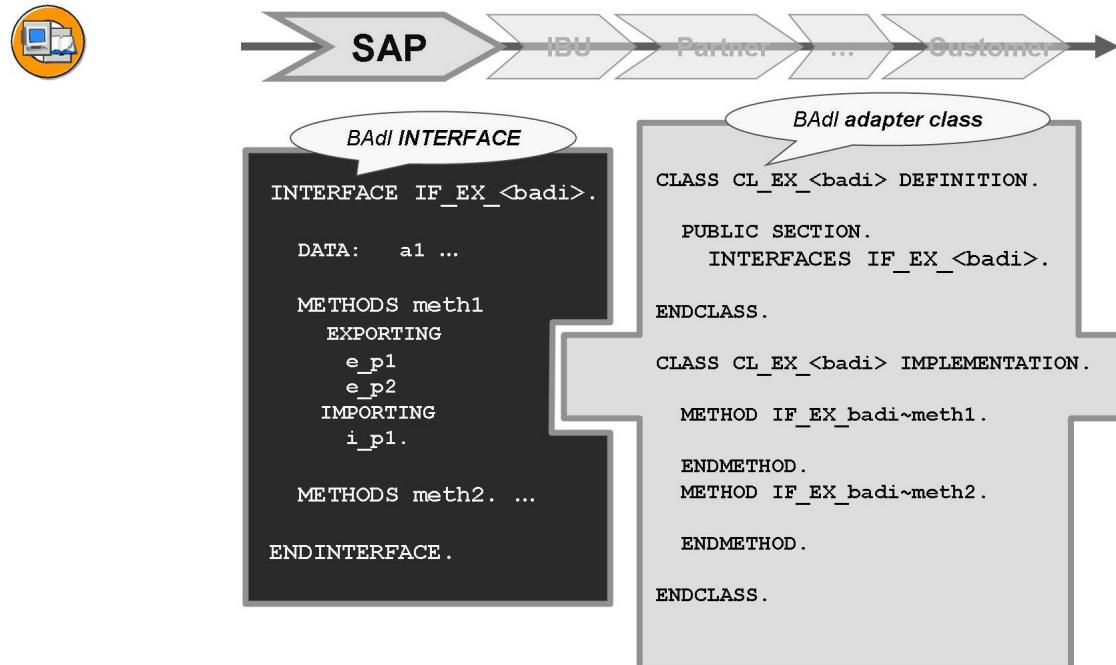


Figure 133: Definition of a BAdI – Basics

When you define a BAdI, you must specify an interface (IF_EX_<badi>) with corresponding, formally defined methods. The adapter class that is automatically generated during the interface definition (CL_EX_<badi>) has, among other things, the function of calling all active implementations of the BAdI. This does not need to be coded, but rather happens in the adapter class itself in an undefined sequence. When you have several active implementations, there is therefore no predefined processing sequence.

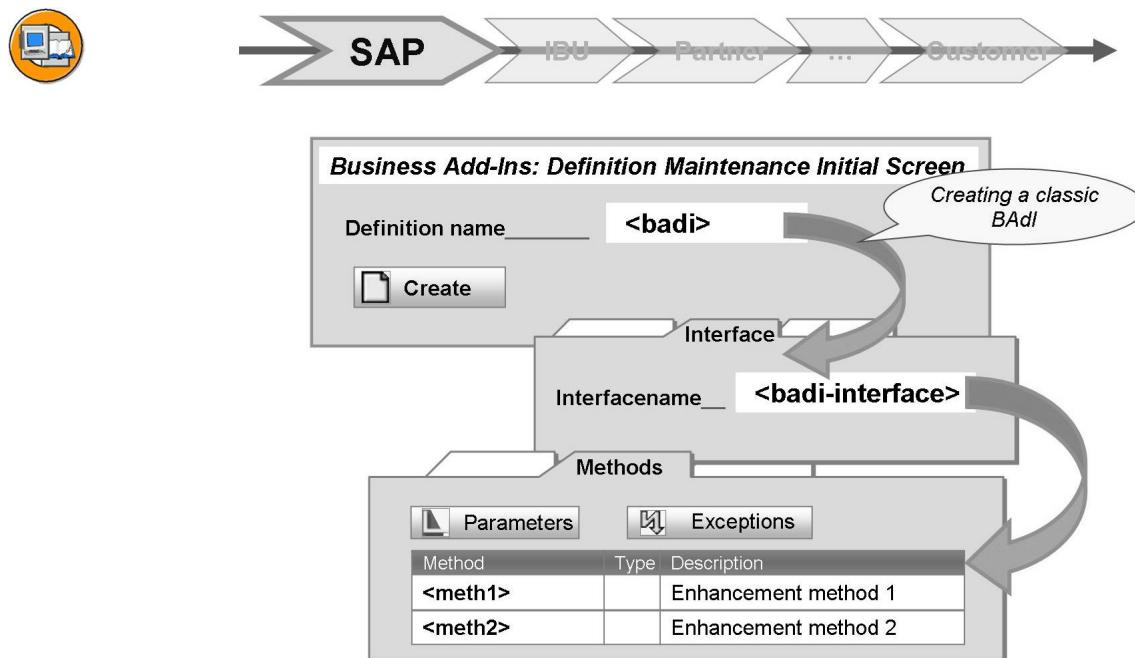


Figure 134: Definition of a BAdI – SE18

To create a BAdI, the BAdI Builder tool (*Tools → ABAP Workbench → Utilities → Business Add-Ins → Definition*) (SE18) is used.

Once the name has been issued for the BAdI, the system suggests names for the BAdI interface and the adapter class. To ensure you keep a clear overview, it is recommended that you retain these names.

→ **Note:** In addition to the way shown here, there is another option for defining BAdIs. This course focuses on the use of OO techniques, however, in particular here the use of interfaces and their reference variables.

Detailed information on the BAdI interface definition is provided in the specific training course on the enhancement techniques and is not part of this introduction.

If you double-click on the interface name, the system switches to the Class Builder, where the interface methods can be defined. These methods are defined as usual.

After the interface methods are defined, the interface must be **activated!**



```

REPORT <program_using_badi>.

DATA: gb_adapter TYPE REF TO IF_EX_<badi>.

START-OF-SELECTION.
*-----
*** this demo shows a classical BAdI call
*** it is available since 4.6 release
CALL METHOD cl_exithandler->get_instance
      CHANGING instance = gb_adapter

CALL METHOD gb_adapter->meth1
      IMPORTING
          ev_p1 = ...
          ev_p2 = ...
      EXPORTING
          iv_p1 = ...
    
```

*Program with
classical BAdI*

*Adapter class
instance*

*This instance calls all
implementations for
meth1*



Figure 135: BAdI – The Calling Program

The call of a BAdI is displayed in the graphic.

First, a reference variable must be defined of the type “Reference to the Business Add-In interface”.

An object of the adapter class is instantiated by the call of the GET_INSTANCE static method of the CL_EXITHANDLER class. GB_ADAPTER points to this instance.

The interface methods of the BAdI can now be called through this object reference.



- BAdI search using the **Repository Information System**
- BAdI search using the **Application Hierarchy**
- BAdI search using the **IMG** (documentation)
- BAdI search directly in the **application program**

REPORT <program_using_badi>.

```

DATA: gb_adapter TYPE REF TO IF_EX<badi>.
START-OF-SELECTION.
*-----
...  

*** classical BAdI demo
CALL METHOD cl_exithandler=>get_instance
    CHANGING instance      = gb_adapter.
...

```

A callout bubble points to the text 'IF_EX<badi>' in the code. A magnifying glass icon is positioned above the callout bubble. A text box labeled 'Search in SAP programs for "IF_EX_"' is also part of the callout.

Figure 136: Finding Business Add-Ins

- The Repository Infosystem (SE84)
- The application hierarchy (SE81)
- The Customizing Guide (Implementation guide for SPRO) or
- You can navigate directly by using F1 → Technical Info in the application program and look for the instantiation of the BAdI adapter class by calling CL_EXITHANDLER=>GET_INSTANCE.
- You can search in the application program for the occurrence of the BAdI interfaces with the naming convention “IF_EX_”
- As of SAP NW AS 7.0, it is also possible to search using the command “GET BADI” (see documentation for the Enhancement Framework).

Before a BAdI can be implemented by the customer, its name must of course be defined beforehand. When searching for these BAdI names, the tried and tested search mechanisms can come into action:

By double-clicking on the reference variable (here: GB_ADAPTER) you get to their definition, in which the BAdI interface is used for typing. You can derive the BAdI name from the name of the interface.

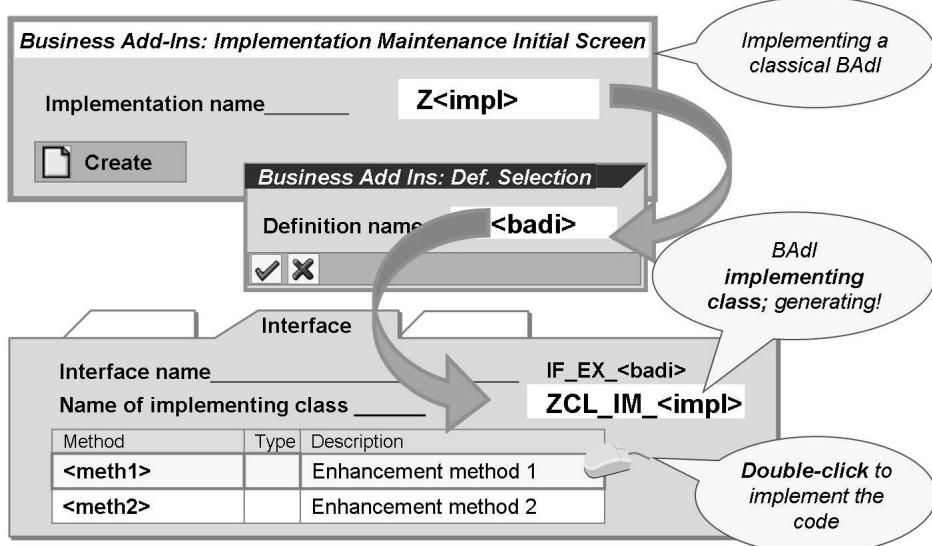


Figure 137: Implementation of a BAdI – SE19

If the name of the BAdI has been determined, the BAdI can be implemented after you study the documentation on this subject. The implementation is performed through the implementation maintenance *Tools → ABAP Workbench → Business Add-Ins → Implementation*. Alternatively, you could also go to the implementations by navigating in the menu from the BAdI definition.

To implement a BAdI, a BAdI implementation name must be issued. The naming convention here, as usual, is Z<impl>. A dialog box then appears for selecting the corresponding BAdI.

The code to be implemented is stored in a method of an automatically generated customer class. For this reason, the **name of the implementing class** must now be given in a final dialog box. The default name from the system is comprised of “Y or Z (the namespace prefix)”, “CL_ (for class)”, “IM_ (for implementation)” and “<impl> (the actual name of the implementation)”.

After you double-click on one of the BAdI methods, you can enter the code of the method. Of course, you can also create auxiliary methods in the implementing class, in order to better structure the required code of the BAdI method.

→ **Note:** Do not forget to activate the methods or the implementation when you have finished!

Exercise 15: 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

You want to create a global class to represent hotels.

Template:

SAPBC401_EVE_S2 (main program)

Solution:

SAPBC401_GCL_S1 (main program)

CL_HOTEL (global class)

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:

C_POS_1	of the type I	a private constant with value 30
MV_NAME	of the type STRING	as a private instance attribute
MV_BEDS	of the type I	as a private instance attribute
GV_N_O_HOTELS	of the type I	as a private static attribute

Maintain the short texts.

3. Define the following methods in the class:

Continued on next page

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

Use the constant defined before to align the output of the values in a column.

Maintain the short texts.

Task 2:

Check your work.

1. Activate your class.
2. Test your class in the Class Builder's testing environment.

Task 3:

Create an instance of your global class for hotels in your main program.

1. Complete your program ZBC401_##_MAIN ((where ## is your two-digit group number) or copy the template program.
2. Define a reference variable, specify your global hotel class as the type, and create an instance.
3. If you have not done the optional exercise, where you add business partners to the travel agency through event handling, try to call method ADD_PARTNER for the hotel.
4. Why can the hotel not yet be integrated into the travel agency's list of business partners?

Continued on next page

5. Could the local interface LIF_PARTNER be implemented in the global class for hotels?

Solution 15: 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.
2. Define the following attributes in the class:

C_POS_1	of the type I	a private constant with value 30
MV_NAME	of the type STRING	as a private instance attribute
MV_BEDS	of the type I	as a private instance attribute
GV_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.
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

Use the constant defined before to align the output of the values in a column.

Maintain the short texts.

- a) Follow the processes as outlined in the relevant section of this lesson.

Continued on next page

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.

Task 3:

Create an instance of your global class for hotels in your main program.

1. Complete your program ZBC401_##_MAIN ((where ## is your two-digit group number) or copy the template program.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
2. Define a reference variable, specify your global hotel class as the type, and create an instance.
 - a) See source text excerpt from the model solution.
3. If you have not done the optional exercise, where you add business partners to the travel agency through event handling, try to call method ADD_PARTNER for the hotel.
 - a) Do this exactly in the same way as you already do it for the carrier and the car rental company.
4. Why can the hotel not yet be integrated into the travel agency's list of business partners?

Answer: The global class for hotels does not implement the interface that is used by the travel agency class.

5. Could the local interface LIF_PARTNER be implemented in the global class for hotels?

Answer: No, global classes can only implement global interfaces. Therefore the local interface has to be replaced with a global one. This will be done in the next exercise.

Continued on next page

Result

Main Program SAPBC401_GCL_S1

```

REPORT  sapbc401_gcl_s1.

TYPE-POOLS icon.

INCLUDE bc401_gcl_s1_agency.
INCLUDE bc401_gcl_s1_carrier.
INCLUDE bc401_gcl_s1_rental.

DATA:
  go_hotel      TYPE REF TO cl_hotel,
  go_vehicle    TYPE REF TO lcl_vehicle,
  go_truck      TYPE REF TO lcl_truck,
  go_bus        TYPE REF TO lcl_bus,
  go_rental     TYPE REF TO lcl_rental,
  go_passenger  TYPE REF TO lcl_passenger_plane,
  go_cargo      TYPE REF TO lcl_cargo_plane,
  go_carrier    TYPE REF TO lcl_carrier,
  go_agency     TYPE REF TO lcl_travel_agency.

START-OF-SELECTION.
***** create travel_agency *****
CREATE OBJECT go_agency
  EXPORTING
    iv_name = 'Travel&Smile Travel'.

***** create hotel *****
CREATE OBJECT go_hotel
  EXPORTING
    iv_name = 'Sleep Well Hotel'
    iv_beds = 345.

* hotel not added to partner list - does not implement interface
* go_agency->add_partner( go_hotel )

***** create rental *****
CREATE OBJECT go_rental

```

Continued on next page

```

        EXPORTING
          iv_name = 'Happy Car Rental'.
*  go_agency->add_partner( go_rental ).

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make  = 'MAN'
  iv_cargo = 45.

***** create truck *****
CREATE OBJECT go_bus
EXPORTING
  iv_make      = 'Mercedes'
  iv_passengers = 80.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make  = 'VOLVO'
  iv_cargo = 48.

***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly-Travel'.
*  go_agency->add_partner( go_rental ).

***** Passenger Plane *****
CREATE OBJECT go_passenger
EXPORTING
  iv_name       = 'LH BERLIN'
  iv_planetype = '747-400'
  iv_seats     = 345
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc <> 0.
  WRITE:
    / icon_failure AS ICON,
    'Wrong plane type'.
ENDIF.

***** cargo Plane *****
CREATE OBJECT go_cargo
EXPORTING

```

Continued on next page

```
        iv_name          = 'US Hercules'  
        iv_planetype    = '747-200F'  
        iv_cargo         = 533  
EXCEPTIONS  
        wrong_planetype = 1.  
  
IF sy-subrc <> 0.  
  WRITE:  
    / icon_failure AS ICON,  
      'Wrong plane type'.  
ENDIF.  
  
***** show attributes of all partners of travel_agency *****  
go_agency->display_attributes( ).
```

Exercise 16: 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.

Template:

SAPBC401_EVE_S2 (program)
CL_HOTEL (global class)

Solution:

SAPBC401_GCL_S1 (program)
CL_HOTEL2 (global class)
IF_PARTNER (global interface)

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_##_PARTNER.

Define the instance method DISPLAY_PARTNER in your global interface. If the local interface contains the event PARTNER_CREATED define the event also in your global interface.



Hint: Instead of creating the global interface manually you can **import** the local interface from your program ZBC401_##_MAIN or from the template program.

Continued on next page

Task 2:

Replace the local interface in your main program with the global one.

1. Complete your program ZBC401_##_MAIN ((where ## is your two-digit group number) or copy the template program .
2. Completely remove the definition of the local interface
3. Adapt all places where the local interface was used so that the global interface is used and implemented instead.



Hint: To reduce the typing effort take advantage of the search/replace functionality of the *ABAP Editor*.

Task 3:

Have your hotel class implement the interface and test your program.

1. Complete your global class ZCL_##_HOTEL (where ## is your two-digit group number) or copy the template class.
2. In your main program, replace the reference to the template class with a reference to your own class.
3. 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.
4. Declare the interface as an implemented interface in your hotel class.
5. Implement the interface method in such a way that the hotel's DISPLAY_ATTRIBUTES instance method is called.
6. Activate the global class and your program. Run your program. If you have done everything correctly, the hotel attributes should now be displayed on the list too. Otherwise, debugging is the best way of analyzing any errors that may have occurred here.

Solution 16: 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_##_PARTNER.

Define the instance method DISPLAY_PARTNER in your global interface. If the local interface contains the event PARTNER_CREATED define the event also in your global interface.



Hint: Instead of creating the global interface manually you can **import** the local interface from your program ZBC401_##_MAIN or from the template program.

- 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:

Replace the local interface in your main program with the global one.

1. Complete your program ZBC401_##_MAIN ((where ## is your two-digit group number) or copy the template program .
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
2. Completely remove the definition of the local interface
 - a) See source text excerpt from the model solution.
3. Adapt all places where the local interface was used so that the global interface is used and implemented instead.



Hint: To reduce the typing effort take advantage of the search/replace functionality of the *ABAP Editor*.

- a) See source text excerpt from the model solution.

Continued on next page

Task 3:

Have your hotel class implement the interface and test your program.

1. Complete your global class ZCL_##_HOTEL (where ## is your two-digit group number) or copy the template class.
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
2. In your main program, replace the reference to the template class with a reference to your own class.
 - a) See source text excerpt from the model solution.
3. 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.
4. Declare the interface as an implemented interface in your hotel class.
 - a) Follow the processes as outlined in the relevant section of this lesson.
5. 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.
6. Activate the global class and your program. Run your program. If you have done everything correctly, the hotel attributes should now be displayed on the list too. 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

Include BC401_GCL_S2_AGENCY

```
*-----*
*      INTERFACE lif_partners
*&-----*
*INTERFACE lif_partner.
*  METHODS:
*    display_partner.
*  EVENTS:
*    partner_created.
```

Continued on next page

```

*ENDINTERFACE.                                "lif_partners
*-----*
*      CLASS lcl_travel_agency DEFINITION
*-----*
CLASS lcl_travel_agency DEFINITION.

PUBLIC SECTION.

METHODS:
    constructor IMPORTING iv_name TYPE string,
    display_agency_partners,
    display_attributes.

PRIVATE SECTION.

DATA:
    mv_name      TYPE string,
    mt_partners TYPE TABLE OF REF TO if_partner.

METHODS:
    on_partner_created FOR EVENT partner_created
        OF if_partner
        IMPORTING sender.

ENDCLASS.                                     "lcl_travel_agency DEFINITION
*-----*
*      CLASS lcl_travel_agency IMPLEMENTATION
*-----*
CLASS lcl_travel_agency IMPLEMENTATION.

METHOD display_attributes.
    WRITE: / icon_private_files AS ICON,
            'Travel Agency:'(007), mv_name.
    SKIP.
    display_agency_partners( ).
ENDMETHOD.                                     "display_attributes

METHOD display_agency_partners.
    DATA:
        lo_partner TYPE REF TO if_partner.

    WRITE 'Here are the partners of the travel agency:'(008).

```

Continued on next page

```

        ULINE.

LOOP AT mt_partners INTO lo_partner.
    lo_partner->display_partner( ).
ENDLOOP.

ENDMETHOD.           "display_agency_partners

METHOD constructor.
    mv_name = iv_name.
    SET HANDLER on_partner_created FOR ALL INSTANCES.
ENDMETHOD.           "constructor

METHOD on_partner_created.
    APPEND sender TO mt_partners.
ENDMETHOD.           "on_partner_created

ENDCLASS.           "lcl_travel_agency IMPLEMENTATION

```

Include BC401_GCL_S2_CARRIER

```

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

PUBLIC SECTION.

INTERFACES if_partner.
METHODS:
    constructor IMPORTING iv_name TYPE string,
    on_airplane_created FOR EVENT airplane_created
        OF lcl_airplane
        IMPORTING sender,
    display_attributes.

PRIVATE SECTION.

DATA:
    mv_name      TYPE string,

```

Continued on next page

```

mt_airplanes TYPE TABLE OF REF TO lcl_airplane.

METHODS:
    display_airplanes.

ENDCLASS.           "lcl_carrier DEFINITION

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

METHOD constructor.
    mv_name = iv_name.
    SET HANDLER on_airplane_created FOR ALL INSTANCES.
    RAISE EVENT if_partner~partner_created.
ENDMETHOD.          "constructor

METHOD if_partner~display_partner.
    display_attributes( ).
ENDMETHOD.          "if_partners~display_partner

METHOD on_airplane_created.
    APPEND sender TO mt_airplanes.
ENDMETHOD.          "on_airplane_created

METHOD display_attributes.
    SKIP 2.
    WRITE: icon_flight AS ICON,
           mv_name.
    ULINE.
    ULINE.
    display_airplanes( ).
ENDMETHOD.          "display_attributes

METHOD display_airplanes.
    DATA: lo_plane TYPE REF TO lcl_airplane.
    LOOP AT mt_airplanes INTO lo_plane.
        lo_plane->display_attributes( ).
    ENDLOOP.
ENDMETHOD.          "display_airplanes

ENDCLASS.           "lcl_carrier IMPLEMENTATION

```

Continued on next page

Include BC401_GCL_S2_RENTAL

```

...
*-----*
*      CLASS lcl_rental DEFINITION
*-----*
CLASS lcl_rental DEFINITION.

PUBLIC SECTION.

INTERFACES:
  if_partner.

METHODS:
  constructor IMPORTING iv_name TYPE string,
    on_vehicle_created FOR EVENT vehicle_created
      OF lcl_vehicle
      IMPORTING sender,
    display_attributes.

PRIVATE SECTION.

DATA:
  mv_name TYPE string,
  mt_vehicles TYPE TABLE OF REF TO lcl_vehicle.

ENDCLASS.          "lcl_rental DEFINITION
*-----*
*      CLASS lcl_rental IMPLEMENTATION
*-----*
CLASS lcl_rental IMPLEMENTATION.

METHOD if_partner~display_partner.
  display_attributes( ).
ENDMETHOD.          "if_partners~display_partner

METHOD constructor.
  mv_name = iv_name.
  SET HANDLER on_vehicle_created FOR ALL INSTANCES.
  RAISE EVENT if_partner~partner_created.

```

Continued on next page

```
ENDMETHOD.           "constructor

METHOD  on_vehicle_created.
    APPEND sender TO mt_vehicles.
ENDMETHOD.           "on_vehicle_created

METHOD  display_attributes.
DATA:
    lo_vehicle TYPE REF TO lcl_vehicle.

WRITE: / icon_transport_proposal AS ICON,
       mv_name.
WRITE: / 'Here comes the vehicle list: '.
ULINE.
ULINE.

LOOP AT mt_vehicles INTO lo_vehicle.
    lo_vehicle->display_attributes( ).
ENDLOOP.
ENDMETHOD.           "display_attributes

ENDCLASS.           "lcl_rental IMPLEMENTATION
```


Exercise 17: (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.

Template:

CL_HOTEL2 (global class)

Solution:

CL_HOTEL3 (global class)

CL_HOUSE (global class)

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 MV_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_##_HOUSE and leave it **empty**.



Caution: Make sure you un-check the checkbox *final*. In the next lesson you'll learn the exact meaning of this property.

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 c_POS_1 constant, the MV_NAME attribute, the instance constructor, and the DISPLAY_ATTRIBUTES method into the class ZCL_##_HOUSE.

Continued on next page

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 and implement the instance constructor for the subclass.
5. Redefine the DISPLAY_ATTRIBUTES method in the subclass.



Hint: To make constant C_POS_1 available in the implementation of DISPLAY_ATTRIBUTES of the subclass, you have to change its visibility to at least protected.

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 17: (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 MV_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_##_HOUSE and leave it **empty**.



Caution: Make sure you un-check the checkbox *final*. In the next lesson you'll learn the exact meaning of this property.

- 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 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 c_POS_1 constant, the MV_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 and implement 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.



Hint: To make constant C_POS_1 available in the implementation of DISPLAY_ATTRIBUTES of the subclass, you have to change its visibility to at least protected.

- 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.

Exercise 18: ALV Grid Control

Exercise Objectives

After completing this exercise, you will be able to:

- Create an ALV Grid Control
- React to double-clicking on the ALV Grid Control

Business Example

The user wants to see a list of flight connections in an ALV Grid Control.

When double-clicking on a connection, the selected row number and column name should be displayed.

Template:

SAPBC401_ALV_T1

Solution:

SAPBC401_ALV_S1

Task 1:

Programming an ALV Grid Control and displaying the list of flight connections already stored in internal table GT_SPFLI.

1. Copy the template program SAPBC401_ALV_T1 to ZBC401_##_ALV.
2. In the main program, define the reference variables GO_CONTAINER and GO_ALV_GRID required for the ALV Grid Control.
3. Analyze the already existing screen 100 – in particular the name of the Custom Container Control Area, which is used as the basis for the ALV Grid Control.

Now implement the PBO module “ALV_GRID”. Use the reference variables to instantiate the two classes in this PBO module. Pay particular attention to the fact that the CREATE OBJECT is NOT called repeatedly when the PBO is executed repeatedly.

4. To display data in the ALV Grid Control, call the method SET_TABLE_FOR_FIRST_DISPLAY and transfer the internal table with the flight connection data and the name of the line type of that internal table.
5. Activate and test your program.

Continued on next page



Caution: Note that you are working with a screen in this exercise and that you should always activate all parts of the program. Therefore, do **not** forget to activate the screen.

Task 2:

Handle a **Double-click** on the ALV Grid Control.

If the user double-clicks on a row in the ALV Grid Control, the row number and column name where the user double-clicked should be displayed in a message of type 'T'.

1. Define a local class LCL_EVENT_HANDLER and in this class define an instance method to handle the event DOUBLE_CLICK triggered by the ALV Grid Control.



Note: Note that this class only acts as a handler and otherwise does not have any other functions.

Import parameters ES_ROW_NO and E_COLUMN into the handler method.

2. Implement the handler method. Send a message of type 'T' to output the row number and column name the user clicked on.



Hint: You can use message 010 of message class BC401.

3. Instantiate the handler class and register the event handler method.
4. Activate and test your program.

Solution 18: ALV Grid Control

Task 1:

Programming an ALV Grid Control and displaying the list of flight connections already stored in internal table GT_SPFLI.

1. Copy the template program SAPBC401_ALV_T1 to ZBC401_##_ALV.
 - a) Carry out this step in the usual manner.
2. In the main program, define the reference variables GO_CONTAINER and GO_ALV_GRID required for the ALV Grid Control.
 - a) See the source code extract from the model solution or refer to the corresponding slides about the ALV Grid Control.
3. Analyze the already existing screen 100 – in particular the name of the Custom Container Control Area, which is used as the basis for the ALV Grid Control.

Now implement the PBO module “ALV_GRID”. Use the reference variables to instantiate the two classes in this PBO module. Pay particular attention to the fact that the CREATE OBJECT is NOT called repeatedly when the PBO is executed repeatedly.

- a) See the source code extract from the model solution or refer to the corresponding slides about the ALV Grid Control. Speak to your instructor if you have any questions.
4. To display data in the ALV Grid Control, call the method SET_TABLE_FOR_FIRST_DISPLAY and transfer the internal table with the flight connection data and the name of the line type of that internal table.
 - a) See the source code extract from the model solution or refer to the corresponding slides about the ALV Grid Control. Speak to your instructor if you have any questions.
5. Activate and test your program.



Caution: Note that you are working with a screen in this exercise and that you should always activate all parts of the program. Therefore, do **not** forget to activate the screen.

- a) Carry out this step in the usual manner.

Continued on next page

Task 2:

Handle a **Double-click** on the ALV Grid Control.

If the user double-clicks on a row in the ALV Grid Control, the row number and column name where the user double-clicked should be displayed in a message of type 'T'.

1. Define a local class LCL_EVENT_HANDLER and in this class define an instance method to handle the event DOUBLE_CLICK triggered by the ALV Grid Control.

 **Note:** Note that this class only acts as a handler and otherwise does not have any other functions.

Import parameters ES_ROW_NO and E_COLUMN into the handler method.

- a) See the source code extract from the model solution or refer to the corresponding slides about double-clicking on the ALV Grid Control.
2. Implement the handler method. Send a message of type 'T' to output the row number and column name the user clicked on.



Hint: You can use message 010 of message class BC401.

- a)
3. Instantiate the handler class and register the event handler method.
 - a) See the source code extract from the model solution or refer to the corresponding slides about double-clicking on the ALV Grid Control.
4. Activate and test your program.
 - a) Carry out this step in the usual manner.

Result

SAPBC401_ALV_S1

```
REPORT  sapbc401_alv_s1.

*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
```

Continued on next page

```

METHODS on_double_click FOR EVENT double_click
        OF cl_gui_alv_grid
        IMPORTING es_row_no e_column.
ENDCLASS.                                     "lcl_event_handler DEFINITION

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
CLASS lcl_event_handler IMPLEMENTATION.

METHOD on_double_click.

MESSAGE i010(bc401) WITH es_row_no-row_id e_column-fieldname.

ENDMETHOD.                                     "handler_method
ENDCLASS.                                     "lcl_event_handler IMPLEMENTATION

***** Types and Data Definitions *****
TYPES: ty_spfli TYPE STANDARD TABLE OF spfli
       WITH NON-UNIQUE KEY carrid connid.

DATA: ok_code TYPE sy-ucomm.

DATA: go_handler TYPE REF TO lcl_event_handler,
      go_container TYPE REF TO cl_gui_custom_container,
      go_alv_grid  TYPE REF TO cl_gui_alv_grid.

DATA: gt_spfli TYPE ty_spfli.

START-OF-SELECTION.
***** 

SELECT * FROM spfli INTO TABLE gt_spfli.

CALL SCREEN '0100'.

*&-----*
*&     Module STATUS_0100  OUTPUT
*&-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYNPROSTATUS'.
  SET TITLEBAR 'TITLE1'.
ENDMODULE.                                     " STATUS_0100  OUTPUT
*&-----*

```

Continued on next page

```

*&      Module ALV_GRID  OUTPUT
*-----*
MODULE alv_grid OUTPUT.

*** Create object of class CL_GUI_CUSTOM_CONTAINER to manage data !
IF go_container IS NOT BOUND.
  CREATE OBJECT go_container
    EXPORTING
      container_name = 'CONTAINER_1'
  EXCEPTIONS
    others      = 6.
  IF sy-subrc <> 0.
    MESSAGE ID sy-msgid TYPE sy-msgtyle NUMBER sy-msgno
      WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  ENDIF.

*** create object of class cl_gui_alv_grid to visualize data !
CREATE OBJECT go_alv_grid
  EXPORTING
    i_parent = go_container
  EXCEPTIONS
    others      = 5.
  IF sy-subrc <> 0.
    MESSAGE ID sy-msgid TYPE sy-msgtyle NUMBER sy-msgno
      WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  ENDIF.

*** create handler instance and register handler method *****
CREATE OBJECT go_handler.
SET HANDLER go_handler->on_double_click FOR go_alv_grid.

*** Call method to visualize data of internal table *****
CALL METHOD go_alv_grid->set_table_for_first_display
  EXPORTING
    i_structure_name = 'SPFLI'
  CHANGING
    it_outtab        = gt_spfli
  EXCEPTIONS
    OTHERS          = 4.
  IF sy-subrc <> 0.
    MESSAGE ID sy-msgid TYPE sy-msgtyle NUMBER sy-msgno
      WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  ENDIF.

ENDIF.

```

Continued on next page

```
ENDMODULE.          " ALV_GRID  OUTPUT
*&-----*
*&      Module  USER_COMMAND_0100  INPUT
*&-----*
MODULE user_command_0100 INPUT.

CASE ok_code.
  WHEN 'BACK'.
    SET SCREEN 0.
  WHEN 'EXIT'.
    LEAVE PROGRAM.
ENDCASE.

ENDMODULE.          " USER_COMMAND_0100  INPUT
```


Exercise 19: (Optional) Second ALV Grid Control in a Popup

Exercise Objectives

After completing this exercise, you will be able to:

- Advanced ALV Grid Programming

Business Example

The user wants to see a list of flight connections in an ALV Grid Control.

By double-clicking on one connection the list of flights for this connection should be displayed in a second ALV Grid Control.

Template:

SAPBC401_ALV_S1

Solution:

SAPBC401_ALV_S2

Task 1:

Further develop the solution of the previous exercise. First, read from database table SFLIGHT the flights, that belong to the flight connection the user has double-clicked.

1. Complete your program ZBC401_##_ALV (where ## is your two-digit group number) or copy the model solution of the previous exercise.
2. In the event handler method, remove the MESSAGE statement. Define a local structure (suggested name: LS_SPFLI) to serve as a work area for internal table GT_SPFLI. From GT_SPFLI, read the one entry, that corresponds to the grid row the user double-clicked on.



Hint: Use READ TABLE with an index access.

Continued on next page

3. Is it still okay to define GT_SPFLI after the definition of the event handler class?

4. In the definition of the event handler class, define a private instance attribute MT_SFLIGHT, which is an internal (standard) table with line type SFLIGHT.



Hint: We suggest that you define a private table type TY_SFLIGHT in the same class.

5. In the event handler method, implement an array fetch for database table SFLIGHT to fill MT_SFLIGHT with those flights, which have the same key fields as the flight connection the user has clicked on.
6. Activate your program and analyze it in the *ABAP Debugger*. Make sure MT_SFLIGHT is filled correctly.

Task 2:

Further extend your event handler method so that the selected flights are presented in a second ALV Grid Control which is displayed in a Dialogbox Container.

1. In your event handler class, define two reference variables as private instance attributes (suggested names: MO_CONT_POPUP and MO_ALV_POPUP). Type them with the global classes CL_GUI_DIALOGBOX_CONTAINER and CL_GUI_ALV_GRID.
2. In your event handler method, use these reference variables to create instances of the two classes.

Pay particular attention to the fact that the CREATE OBJECT is NOT called repeatedly when the event handler method is executed repeatedly.

3. To display data in the ALV Grid Control, call the method SET_TABLE_FOR_FIRST_DISPLAY and transfer the internal table MT_SFLIGHT with the flight data and the name of the line type of that internal table.

Make sure this method is only called if the dialogbox container and the second ALV Grid Control have just been created.

Continued on next page

4. If the dialogbox container and the second ALV Grid Control already exist, do not call method SET_TABLE_FOR_FIRST_DISPLAY to refresh the data displayed. Call the method REFRESH_TABLE_DISPLAY instead.



Hint: You don't have to provide values for the optional parameters of this method.

5. Activate and test your program.



Note: Note, that at the moment, clicking on the button in the top right corner of the window does not close the dialogbox container. To implement that, you would have to handle event CLOSE of class CL_GUI_DIALOGBOX_CONTAINER and call method FREE of the dialogbox container. Details are explained in other training courses or in the documentation of the Control Framework.

Solution 19: (Optional) Second ALV Grid Control in a Popup

Task 1:

Further develop the solution of the previous exercise. First, read from database table SFLIGHT the flights, that belong to the flight connection the user has double-clicked.

1. Complete your program ZBC401_##_ALV (where ## is your two-digit group number) or copy the model solution of the previous exercise.
 - a) Carry out this step in the usual manner.
2. In the event handler method, remove the MESSAGE statement. Define a local structure (suggested name: LS_SPFLI) to serve as a work area for internal table GT_SPFLI. From GT_SPFLI, read the one entry, that corresponds to the grid row the user double-clicked on.



Hint: Use READ TABLE with an index access.

- a) See the source code extract from the model solution.
3. Is it still okay to define GT_SPFLI after the definition of the event handler class?

Answer: No. To be able to access the global data object GT_SPFLI from within the local class implementation, the data statement has to be **before** the local class.

4. In the definition of the event handler class, define a private instance attribute MT_SFLIGHT, which is an internal (standard) table with line type SFLIGHT.



Hint: We suggest that you define a private table type TY_SFLIGHT in the same class.

- a) See the source code extract from the model solution.
5. In the event handler method, implement an array fetch for database table SFLIGHT to fill MT_SFLIGHT with those flights, which have the same key fields as the flight connection the user has clicked on.
 - a) See the source code extract from the model solution.
6. Activate your program and analyze it in the *ABAP Debugger*. Make sure MT_SFLIGHT is filled correctly.
 - a) Carry out this step in the usual manner.

Continued on next page

Task 2:

Further extend your event handler method so that the selected flights are presented in a second ALV Grid Control which is displayed in a Dialogbox Container.

1. In your event handler class, define two reference variables as private instance attributes (suggested names: MO_CONT_POPUP and MO_ALV_POPUP). Type them with the global classes CL_GUI_DIALOGBOX_CONTAINER and CL_GUI_ALV_GRID.

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

2. In your event handler method, use these reference variables to create instances of the two classes.

Pay particular attention to the fact that the CREATE OBJECT is NOT called repeatedly when the event handler method is executed repeatedly.

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

3. To display data in the ALV Grid Control, call the method SET_TABLE_FOR_FIRST_DISPLAY and transfer the internal table MT_SFLIGHT with the flight data and the name of the line type of that internal table.

Make sure this method is only called if the dialogbox container and the second ALV Grid Control have just been created.

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

4. If the dialogbox container and the second ALV Grid Control already exist, do not call method SET_TABLE_FOR_FIRST_DISPLAY to refresh the data displayed. Call the method REFRESH_TABLE_DISPLAY instead.



Hint: You don't have to provide values for the optional parameters of this method.

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

5. Activate and test your program.



Note: Note, that at the moment, clicking on the button in the top right corner of the window does not close the dialogbox container. To implement that, you would have to handle event CLOSE of class CL_GUI_DIALOGBOX_CONTAINER and call method FREE of the dialogbox container. Details are explained in other training courses or in the documentation of the Control Framework.

a) Carry out this step in the usual manner.

Continued on next page

Result

SAPBC401_ALV_S2

```

REPORT  sapbc401_alv_s2.

TYPES: ty_spfli TYPE STANDARD TABLE OF spfli
      WITH NON-UNIQUE KEY carrid connid.

DATA: gt_spfli TYPE ty_spfli.

*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
*
*-----*
CLASS lcl_event_handler DEFINITION.
PUBLIC SECTION.

METHODS on_double_click FOR EVENT double_click
      OF cl_gui_alv_grid
      IMPORTING es_row_no e_column.

PRIVATE SECTION.

TYPES: ty_sfflight TYPE STANDARD TABLE OF sfflight
      WITH NON-UNIQUE KEY carrid connid fldate.

DATA: mo_cont_popup TYPE REF TO cl_gui_dialogbox_container,
      mo_alv_popup  TYPE REF TO cl_gui_alv_grid.

DATA: mt_sfflight TYPE ty_sfflight.

ENDCLASS.          "lcl_event_handler DEFINITION

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
*
*-----*
CLASS lcl_event_handler IMPLEMENTATION.

METHOD on_double_click.
```

Continued on next page

```

DATA ls_spfli TYPE spfli.

READ TABLE gt_spfli INTO ls_spfli
      INDEX es_row_no-row_id.

SELECT * FROM sflight INTO TABLE mt_sflight
      WHERE carrid = ls_spfli-carrid
      AND connid = ls_spfli-connid.

IF mo_cont_popup IS NOT BOUND.

CREATE OBJECT mo_cont_popup
  EXPORTING
    width  = 600
    height = 300
  EXCEPTIONS
    others = 8.
IF sy-subrc <> 0.
  MESSAGE ID sy-msgid TYPE sy-msgtty NUMBER sy-msgno
  WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
ENDIF.

CREATE OBJECT mo_alv_popup
  EXPORTING
    i_parent = mo_cont_popup
  EXCEPTIONS
    others   = 5.
IF sy-subrc <> 0.
  MESSAGE ID sy-msgid TYPE sy-msgtty NUMBER sy-msgno
  WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
ENDIF.

mo_alv_popup->set_table_for_first_display(
  EXPORTING
    i_structure_name          = 'SFLIGHT'
  CHANGING
    it_outtab                 = mt_sflight
  EXCEPTIONS
    OTHERS                     = 4
  .
IF sy-subrc <> 0.
  MESSAGE ID sy-msgid TYPE sy-msgtty NUMBER sy-msgno
  WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.

```

Continued on next page

```

ENDIF.

ELSE.

mo_alv_popup->refresh_table_display(
  EXCEPTIONS
    OTHERS      = 2
).
IF sy-subrc <> 0.
  MESSAGE ID sy-msgid TYPE sy-msgty NUMBER sy-msgno
    WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
ENDIF.

ENDIF.

*   MESSAGE i010(bc401) WITH es_row_no-row_id e_column-fieldname.

ENDMETHOD.           "handler_method
ENDCLASS.           "lcl_event_handler IMPLEMENTATION

DATA: ok_code TYPE sy-ucomm.

DATA: go_handler TYPE REF TO lcl_event_handler,
      go_container TYPE REF TO cl_gui_custom_container,
      go_alv_grid  TYPE REF TO cl_gui_alv_grid.

START-OF-SELECTION.
*****



SELECT * FROM spfli INTO TABLE gt_spfli.

CALL SCREEN '0100'.

*-----*
*&     Module STATUS_0100 OUTPUT
*-----*
*     text
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYNPROSTATUS'.
  SET TITLEBAR 'TITLE1'.
ENDMODULE.           " STATUS_0100 OUTPUT
*-----*
*&     Module ALV_GRID OUTPUT
*-----*

```

Continued on next page

```

*      text
*-----
MODULE alv_grid OUTPUT.
*** Create object of class CL_GUI_CUSTOM_CONTAINER to manage data !
IF go_container IS NOT BOUND.
  CREATE OBJECT go_container
    EXPORTING
      container_name = 'CONTAINER_1'
    EXCEPTIONS
      others          = 6.
  IF sy-subrc <> 0.
    MESSAGE ID sy-msgid TYPE sy-msgtty NUMBER sy-msgno
      WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  ENDIF.
*** create object of class cl_gui_alv_grid to visualize data !
CREATE OBJECT go_alv_grid
  EXPORTING
    i_parent = go_container
  EXCEPTIONS
    others   = 5.
  IF sy-subrc <> 0.
    MESSAGE ID sy-msgid TYPE sy-msgtty NUMBER sy-msgno
      WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  ENDIF.

*** create handler Instance ****
CREATE OBJECT go_handler.

*** set handler to react on double-click ****
SET HANDLER go_handler->on_double_click FOR go_alv_grid.

*** Call method to visualize data of internal table ****
CALL METHOD go_alv_grid->set_table_for_first_display
  EXPORTING
    i_structure_name = 'SPFLI'
  CHANGING
    it_outtab       = gt_spfli
  EXCEPTIONS
    OTHERS          = 4.
  IF sy-subrc <> 0.
    MESSAGE ID sy-msgid TYPE sy-msgtty NUMBER sy-msgno
      WITH sy-msgv1 sy-msgv2 sy-msgv3 sy-msgv4.
  ENDIF.

ENDIF.

```

Continued on next page

```
ENDMODULE.          " ALV_GRID  OUTPUT
*&-----*
*&      Module  USER_COMMAND_0100  INPUT
*&-----*
MODULE user_command_0100 INPUT.

CASE ok_code.
  WHEN 'BACK'.
    SET SCREEN 0.
  WHEN 'EXIT'.
    LEAVE PROGRAM.
ENDCASE.

ENDMODULE.          " USER_COMMAND_0100  INPUT
```



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
- Create a simple ALV Grid Control
- React to a double-click on an ALV Grid Control with a handler method
- Explain the fundamentals of the BAdI technology and implement a BAdI

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
- Understand the special use of public read-only attributes
- Understand the term “navigation method”
- Chain functional methods
- Limit the visibility of the constructor
- Understand the term “factory method”
- Explain the “singleton pattern”
- Define friendship relationships between classes

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 (among 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.

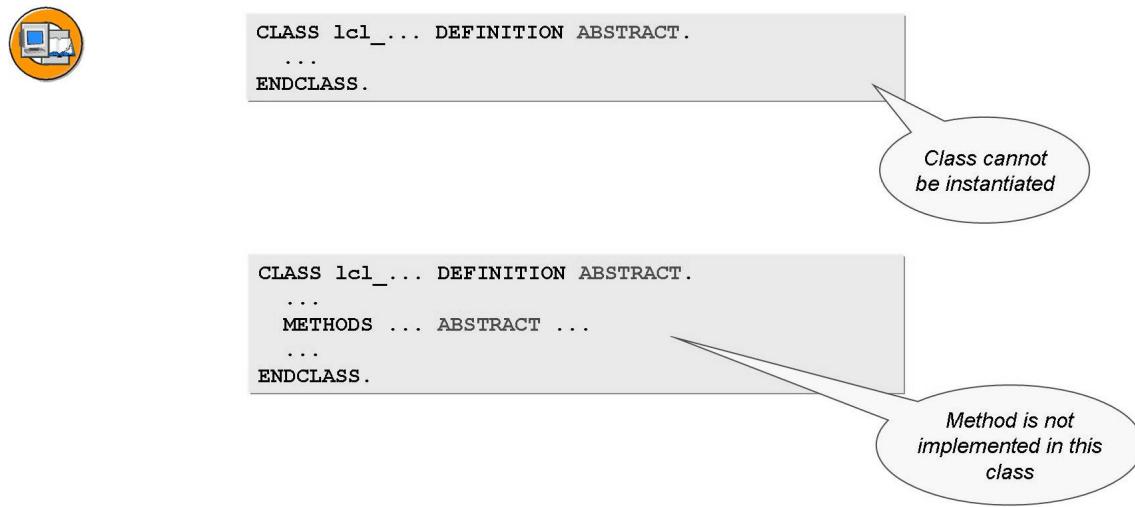


Figure 138: Abstract Classes and Abstract Methods

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.



```
CLASS lcl_... DEFINITION FINAL  
[ INHERITING FROM ... ] .  
...  
ENDCLASS.
```

*Other classes
cannot inherit
from this one*

```
CLASS lcl_... DEFINITION.  
...  
METHODS ... FINAL ...  
...  
ENDCLASS.
```

*Method cannot
be redefined*

Figure 139: 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.

Access to Internal Tables with Object References

We have learned in a previous section that object references can be stored in internal tables. This is used, in particular, to implement associations. However, we have not yet discussed how we should retrieve a specific object from such a list.

The following figure gives an example of how this could be achieved:

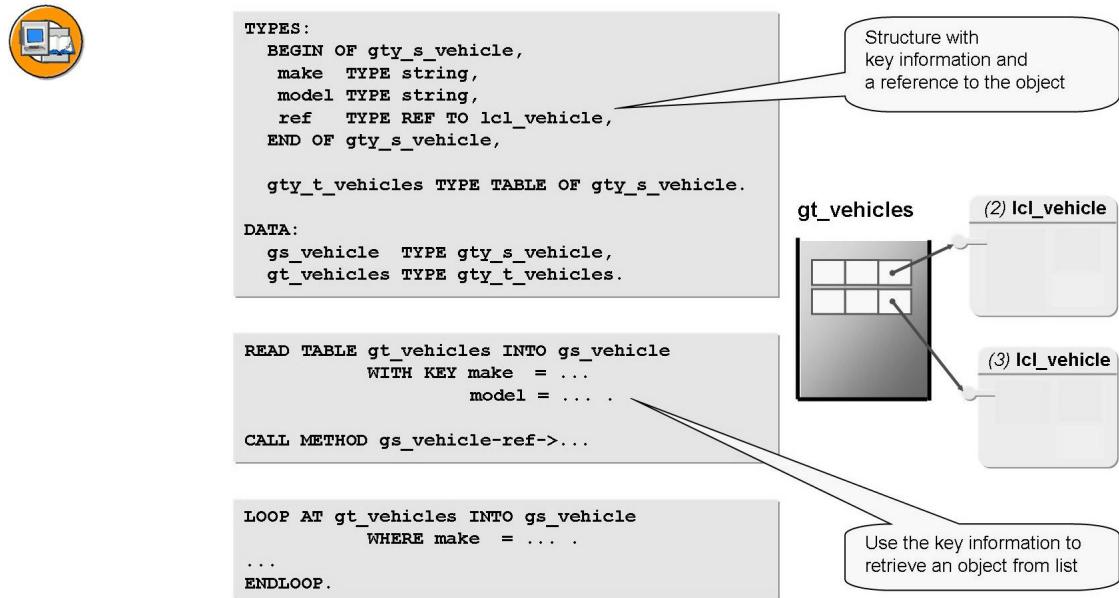


Figure 140: Read Access to a Table with Object References

The object reference is stored in the table along with some key information. In the example it is make and model of the vehicle. The object reference can easily be retrieved through this key information.

Unfortunately, this technique implies a redundant storage of information: the key values are already stored in attributes of the object. By making the key attributes public, this redundancy can be avoided, as is shown in the following figures:

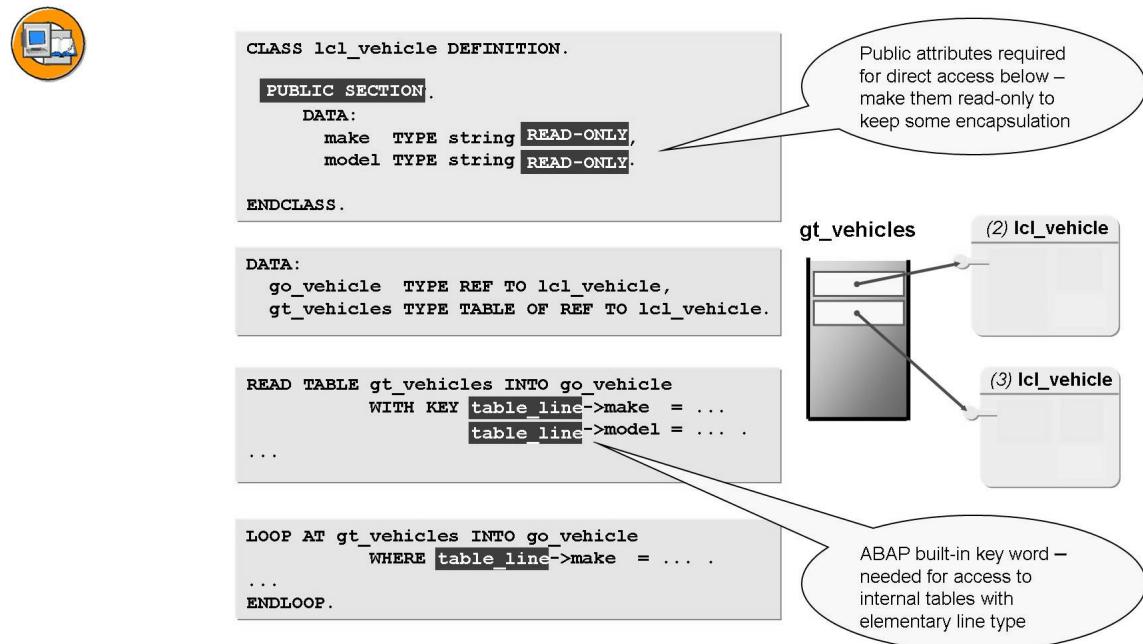


Figure 141: Read Access Using Public Attributes

As we can see, public attributes may be used directly in the statements with which we access internal tables (READ TABLE ... and LOOP AT ..., for example).

The expression TABLE_LINE, which is used here, is a built-in part of the ABAP language. We have to use TABLE_LINE whenever an internal table's line type is not a structure type but an elementary data type. Such internal tables have just one column, which does not have a column name. TABLE_LINE is used as the generic column name for this single column.



Hint: Strictly speaking, it is against the fundamental principles of object-oriented programming to make attributes public. Unfortunately, in ABAP it is not allowed to use functional methods (like GET_MAKE, GET_MODEL, etc.) on the left hand side of WHERE clauses and WITH KEY additions. Therefore, the use of public attributes should be tolerable in such special cases – but they must remain read-only!

Navigation Methods and the Chaining of Method Calls

We have learned that Associations (like Aggregations and Compositions) are an important design principle of object oriented programming. An Association means that at runtime an instance of one class stores references to objects of another class.

Classes that have other classes associated to them often come with methods which return one of the stored references to the caller. Such methods are often called **navigation methods** as they can be used to navigate from one object

to another. In the following example, the class LCL_VEHICLE is associated to class LCL_RENTAL. Class LCL_RENTAL provides the navigation method GET_VEHICLE.

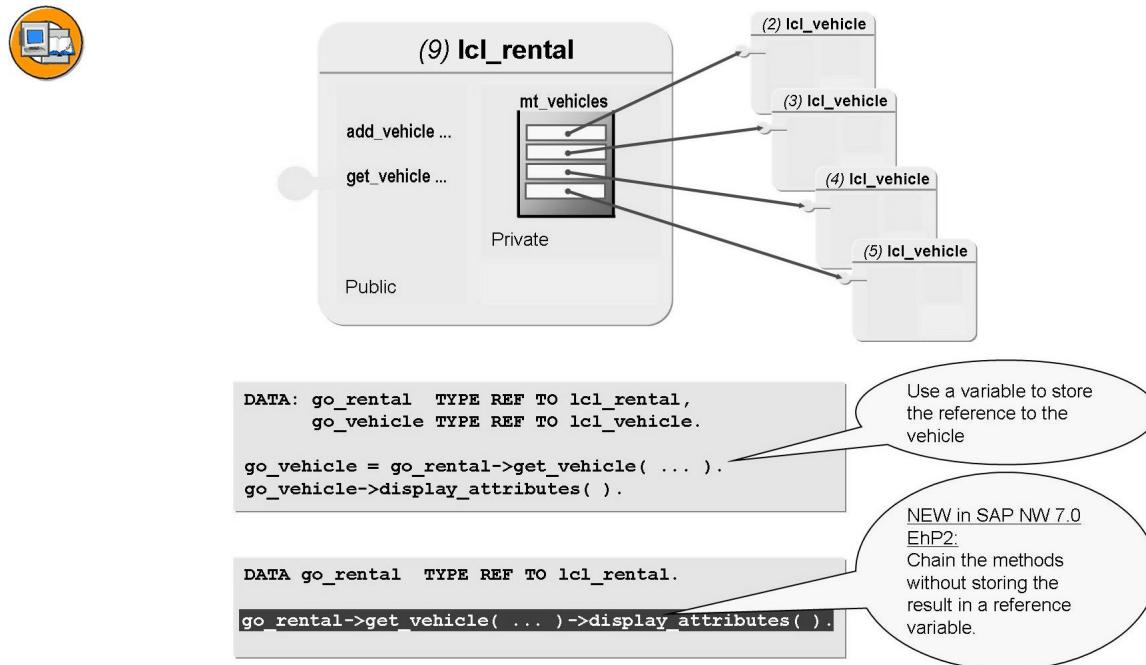


Figure 142: Navigation Methods and Chaining of Method Calls

Often, the purpose of navigating from one object to another is to call a single method of this other object. Before release *SAP NetWeaver 7.0 EhP 2* it was mandatory to store the reference to the new object in a reference variable, first.

As of *SAP NetWeaver 7.0 EhP 2* we can chain method calls, that is, we can directly call a method of the object returned by the navigation method. The above figure gives you an example for this.



Hint: This technique is not restricted to the chaining of two methods. In more complicated object models any number of methods can be chained.

Visibility of the Instance Constructor

In ABAP Objects, we can restrict the visibility of the instance constructor. If the visibility of the instance constructor has been restricted, the CREATE OBJECT statements to instantiate this class are only allowed in certain parts of the coding.



PRIVATE

If a class has a private instance constructor, we can only instantiate it from within the class itself – typically in static methods that are defined for that purpose. These methods are sometimes called “Factor-Methods” (see below).

PROTECTED

If the instance constructor is protected, the visibility is extended to all of its subclasses, that is, also the subclasses can create instances of the class.

PUBLIC

A public instance constructor is the default: Instances of the class can be created anywhere – inside the class itself, inside other classes or even in a non-object oriented part of the program (e.g. the main program).

We set the visibility 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.



```
CLASS lcl_... DEFINITION CREATE PUBLIC.  
...  
ENDCLASS.
```

*All users can
instantiate the
class*

```
CLASS lcl_... DEFINITION CREATE PROTECTED.  
...  
ENDCLASS.
```

*Only the class itself
or its subclasses
can instantiate the
class*

```
CLASS lcl_... DEFINITION CREATE PRIVATE.  
...  
ENDCLASS.
```

*The class can only
be instantiated by
itself*

Figure 143: Implicit Setting of the Instance Constructor's Visibility



Hint: The visibility of the constructor is **not** set by placing the definition of method CONSTRUCTOR in the respective section of the class definition. In fact, before SAP NetWeaver 7.0 it was **mandatory** to place the constructor syntactically in the **public section**.

Since SAP NetWeaver 7.0 it is allowed (but not required) to place the constructor in the protected or private section – if this is not more restrictive than the CREATE ... addition in the class definition. This is meant to increase the readability of the coding.

Factory Methods

Restricting the visibility of the instance constructor only makes sense if somewhere within the class (or its subclasses, in case of CREATE PROTECTED) the instantiation is done.

Often such classes provide a public static method that creates an instance of this class and returns the reference to this instance to the caller. Such methods are sometimes called “factory methods”. Like the instance constructor itself, factory methods are executed when an instance of the class is created. However, they have some advantages:

Some Advantages of Factory Methods:

- A factory method can have coding that is executed **before** the actual instantiation.

For example, the method can do checks, read data from the database, or set locks, before actually creating the instance. If an error occurs during this, no instance is created; and therefore, no memory is allocated for it.

- A class can have more than one factory method. Each of them could have a different implementation and signature.

This makes it possible to create instances of the same class based on different information. As an example, one factory method could create a new instance, offering input parameters for all attributes, while another factory method would only import key information and then retrieve the other attribute values from the database.

- Factory methods can be used to administrate the instances of a class within the class itself.

This is often used to avoid the creation of many, absolutely identical instances: A reference of every new instance is stored in a private static attribute of the class (an internal table). Whenever the factory method is called, it first checks whether an instance with this key already exists. If so, it returns the existing instance instead of creating a new one.

- A factory method can be used to instantiate one of the subclasses rather than the class itself. As an example, the factory method of an airplane class could first check the plane type, then instantiate either a cargo or a passenger plane.

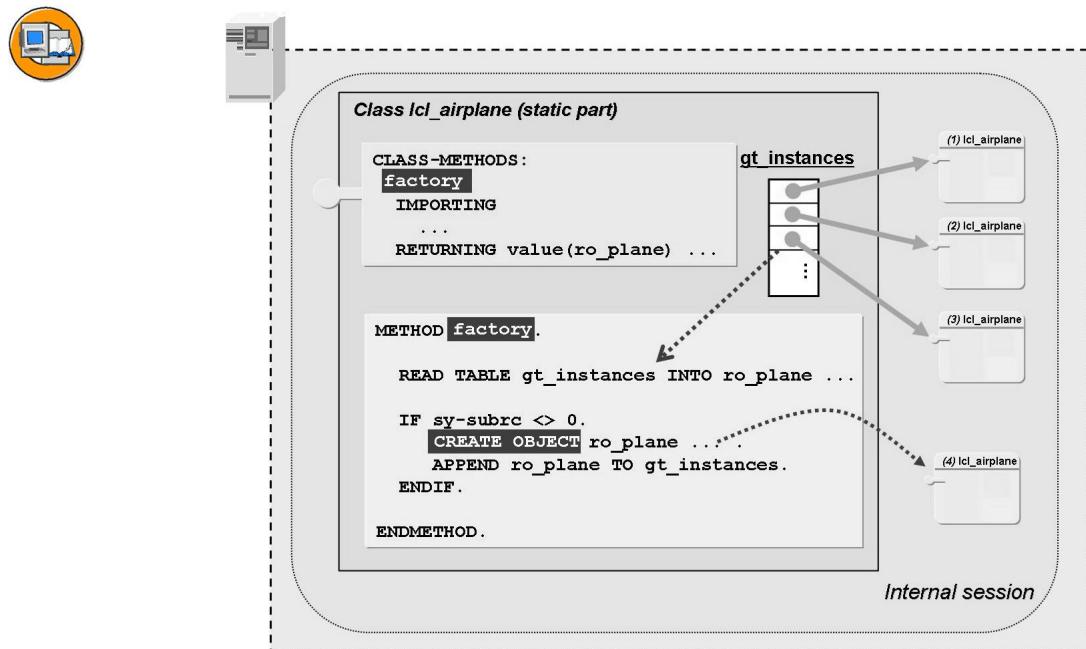


Figure 144: Example for a Class with Factory Method

The above example illustrates a class that keeps a reference to all its instances in a static attribute. The instantiation is restricted to the class itself and is only done in static method FACTORY. When the method is called, it first checks whether an identical instance has already been created. If so, it returns the existing instance instead of creating a new one.

Classes Without Multiple Instantiation (Singleton Concept)

There are many cases in which you need to prevent a class from being instantiated more than once for the same program context. You can do this as follows using the **singleton concept**.

There are various different ways to implement a singleton class. The first singleton pattern is a special case of the factory concept.

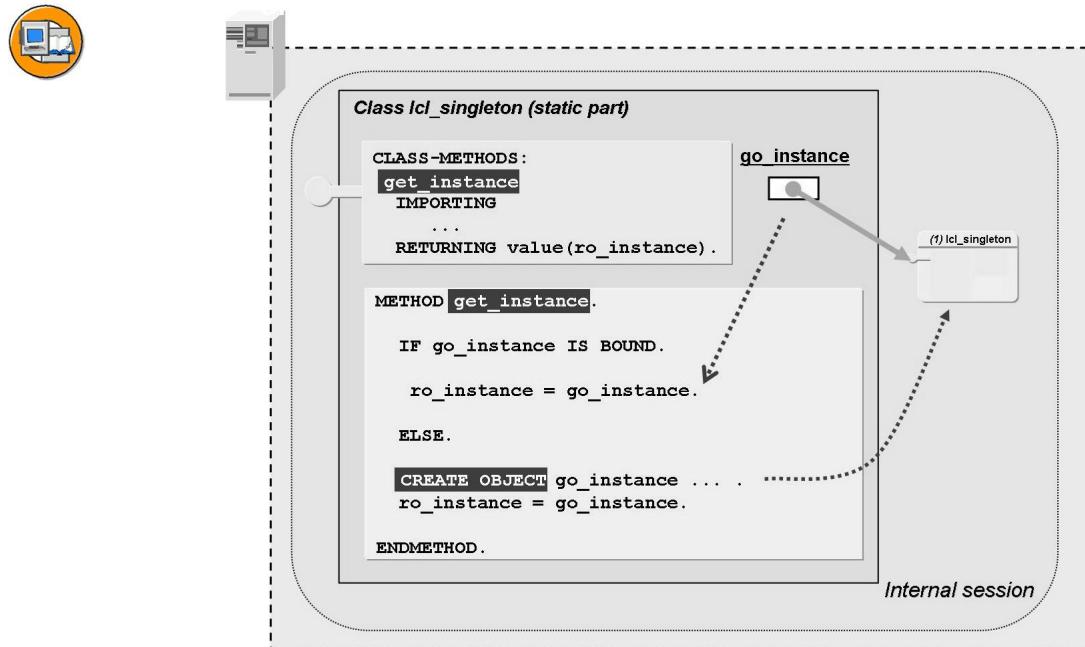


Figure 145: Singleton Class 1: Using a Factory Method

Instead of storing several instances, the class stores only one instance (**the** instance). When the factory method GET_INSTANCE is called for the first time, it instantiates the class. For every subsequent call, it simply returns the reference to the existing object.

The following example shows a more sophisticated singleton pattern:

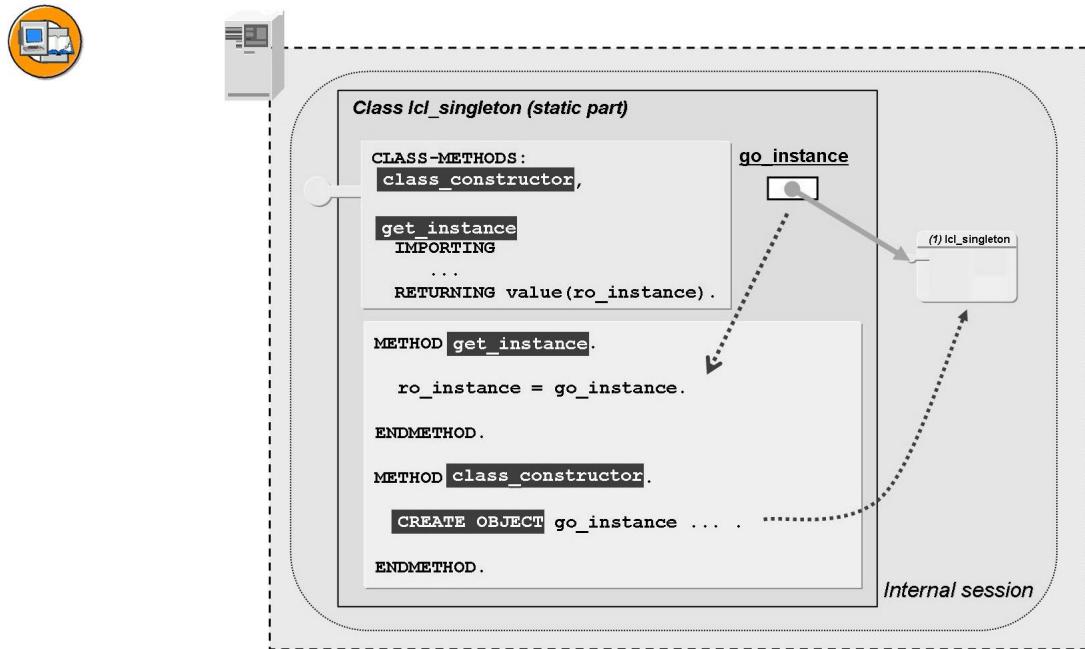


Figure 146: Singleton Class 2: Using the Static Constructor

The class uses its static constructor to create the single instance in advance. The GET_INSTANCE method no longer creates the instance but only returns a reference to the already existing instance.



Hint: In a third variant of the singleton pattern the static attribute GO_INSTANCE is made public and read-only. The GET_INSTANCE method is no longer needed, then.

Friendship Relationship Between Classes

In some cases, classes have to work together so closely that one class needs access to the other's protected and private components. Similarly, it might need to be able to create instances of the other class regardless of the visibility of the constructor. To avoid making these options available to all users of the class, you can use the concept of **class friendship**. A class can grant friendship to other classes and interfaces (and hence to all classes that implement the interface).

Definition of a Friendship Relationship Between Classes

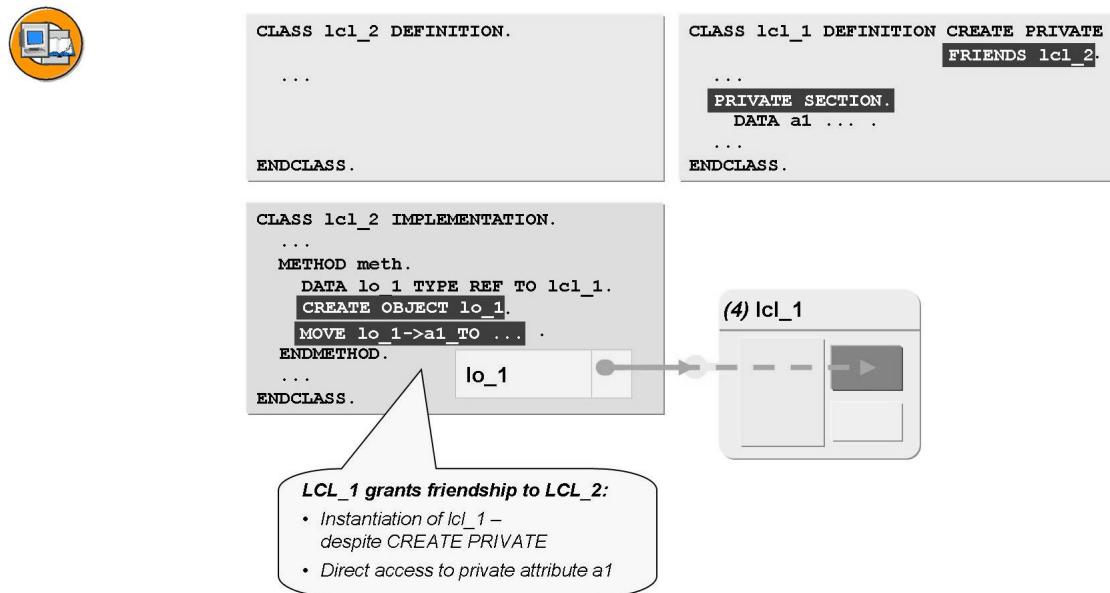


Figure 147: Definition of a Friendship Relationship Between Classes

This is the purpose of the `FRIENDS` addition of the `CLASS` statement or the `FRIENDS` tab in the *Class Builder*. All classes and interfaces to which friendship is granted are listed there.

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.

Factory Class as Application Example for Friendship

A typical application of the **friends** concept is the definition of a **Factory Class**. Like the factory method we discussed earlier, a factory class creates and administers the instances of a class. By outsourcing the administration to a dedicated class, the class itself is kept smaller and easier to understand.

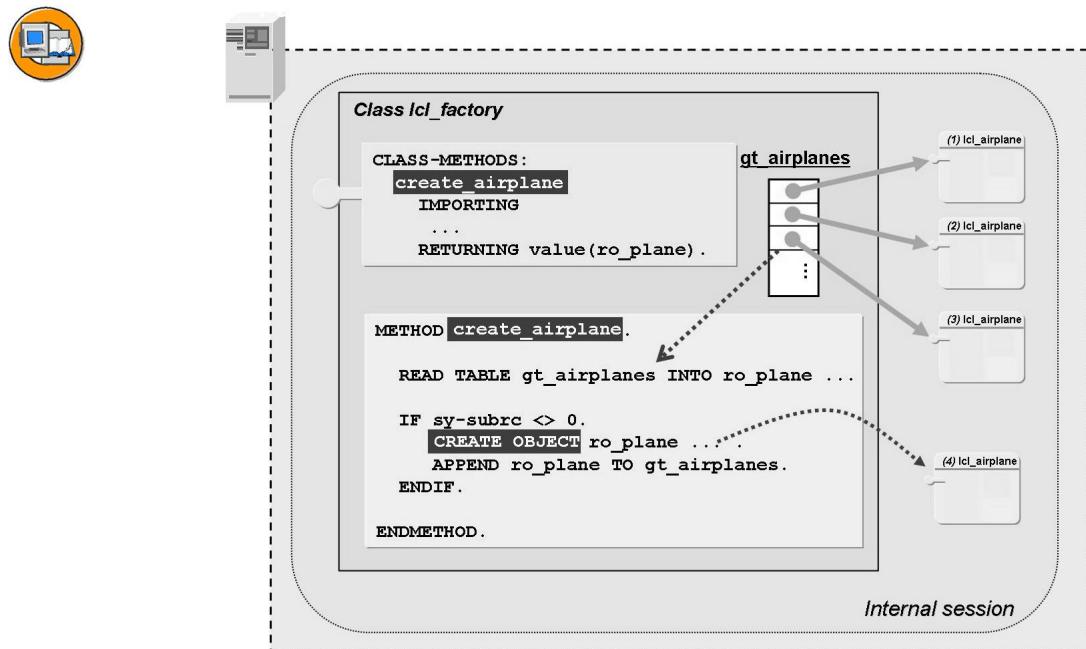


Figure 148: Example for a Factory Class

In the example above, class LCL_FACTORY serves as a factory class for airplanes. It provides a public method CREATE_AIRPLANE in which it either instantiates class LCL_AIRPLANE or returns a reference to an already existing instance.

To restrict the instantiation, class LCL_AIRPLANE is defined with addition CREATE PRIVATE. By adding FRIENDS LCL_FACTORY it allows the factory class (and only the factory class!) to create airplane instances and to access the private attributes.



Hint: Another advantage of the dedicated factory class:

If the class has subclasses, the decision which of the classes should be instantiated could be done inside the factory class rather than by the calling program. In the example above: Depending on the input (for example, the plane type) the CREATE_AIRPLANE method could create and return an instance of either LCL_CARGO_PLANE or LCL_PASSENGER_PLANE.

Friendship and Inheritance

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, **granting** friendship is **not** inherited. A friend of a superclass is therefore not automatically a friend of its subclasses.

Exercise 20: (Optional) Abstract Classes, Final Classes and Singleton Classes

Exercise Objectives

After completing this exercise, you will be able to:

- Define final classes
- Define abstract classes
- Define singleton classes
- Instantiate singleton classes

Business Example

You want to improve the robustness of your application. You want to make sure

- that the two special kinds of airplanes (passenger and cargo) are not further specialized
- that only specific airplanes are created
- that the travel agency can only be instantiated once.

Template:

SAPBC401_GCL_S2

Solution:

SAPBC401_SPC_S1

Task 1:

Use the techniques of ABAP Objects to avoid the further specialization of passenger and cargo planes.

1. Complete your ZBC401_##_MAIN program or copy the template program.
2. Declare classes LCL_PASSENGER_PLANE and LCL_CARGO_PLANE as final.

Continued on next page

3. What would happen if you declared class LCL_AIRPLANE final? (Try it out if you are not sure).

4. What would happen if you only made method DISPLAY_ATTRIBUTES in class LCL_AIRPLANE final? (Try it out if you are not sure).

Task 2:

Use the techniques of ABAP Objects to inhibit the instantiation of class LCL_AIRPLANE.

1. Declare LCL_AIRPLANE as an abstract class.
2. Proof that it is no longer allowed to instantiate the class itself. In your main program, declare a reference variable TYPE REF TO lcl_airplane and Implement a CREATE OBJECT statement for it.

After the syntax check remove the CREATE OBJECT statement.

Task 3:

Make sure the class LCL_TRAVEL_AGENCY can only be instantiated once in your program. Implement one of the singleton patterns discussed in the course.

1. First, restrict the visibility of the instance constructor and inhibit any inheritance from the class.
2. Define a static attribute to store a reference to the one instance, there. Depending on the pattern you implement, make it private or public and read-only.
3. Depending on the pattern you implement, define and implement a static constructor and/or a GET_INSTANCE method. Implement the instantiation of the class according to the singleton pattern.
4. Adjust the main program. Remove the CREATE OBJECT statement for the travel agency from the main program. Depending on the single pattern you implemented, replace it with a call of the GET_INSTANCE method or with an access to the public attribute.

Continued on next page

5. Optional: If you have implemented a pattern that includes a GET_INSTANCE method use a method chain to retrieve the singleton instance and call its DISPLAY_ATTRIBUTE method in one statement.

Solution 20: (Optional) Abstract Classes, Final Classes and Singleton Classes

Task 1:

Use the techniques of ABAP Objects to avoid the further specialization of passenger and cargo planes.

1. Complete your ZBC401_##_MAIN program or copy the template program.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
2. Declare classes LCL_PASSENGER_PLANE and LCL_CARGO_PLANE as final.
 - a) See the source code extract from the model solution.
3. What would happen if you declared class LCL_AIRPLANE final? (Try it out if you are not sure).

Answer: This would result in a syntax error as classes LCL_CARGO_PLANE and LCL_PASSENGER_PLANE already inherit from this class.

4. What would happen if you only made method DISPLAY_ATTRIBUTES in class LCL_AIRPLANE final? (Try it out if you are not sure).

Answer: This would result in a syntax error as the method is already redefined in classes LCL_CARGO_PLANE and LCL_PASSENGER_PLANE.

Task 2:

Use the techniques of ABAP Objects to inhibit the instantiation of class LCL_AIRPLANE.

1. Declare LCL_AIRPLANE as an abstract class.
 - a) See the source code extract from the model solution.
2. Proof that it is no longer allowed to instantiate the class itself. In your main program, declare a reference variable TYPE REF TO lcl_airplane and Implement a CREATE OBJECT statement for it.

After the syntax check remove the CREATE OBJECT statement.

- a) Carry out this step in the usual manner.

Continued on next page

Task 3:

Make sure the class LCL_TRAVEL_AGENCY can only be instantiated once in your program. Implement one of the singleton patterns discussed in the course.

1. First, restrict the visibility of the instance constructor and inhibit any inheritance from the class.
 - a) Use the FINAL and CREATE PRIVATE additions to the CLASS ... DEFINITION statement. Optionally, also move the definition of the instance constructor to the private section. See the source code extract from the model solution.
2. Define a static attribute to store a reference to the one instance, there. Depending on the pattern you implement, make it private or public and read-only.
 - a) See the source code extract from the model solution.
3. Depending on the pattern you implement, define and implement a static constructor and/or a GET_INSTANCE method. Implement the instantiation of the class according to the singleton pattern.
 - a) See the source code extract from the model solution.
4. Adjust the main program. Remove the CREATE OBJECT statement for the travel agency from the main program. Depending on the single pattern you implemented, replace it with a call of the GET_INSTANCE method or with an access to the public attribute.
 - a) See the source code extract from the model solution.
5. Optional: If you have implemented a pattern that includes a GET_INSTANCE method use a method chain to retrieve the singleton instance and call its DISPLAY_ATTRIBUTE method in one statement.
 - a) See the source code extract from the model solution.

Result

Source code extract:

Include BC401_SPC_S1_CARRIER

```
*-----*  
*      CLASS lcl_airplane DEFINITION          *  
*-----*  
CLASS lcl_airplane DEFINITION ABSTRACT.
```

...

Continued on next page

```
ENDCLASS.          "lcl_airplane DEFINITION

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

CLASS lcl_airplane IMPLEMENTATION.

...

ENDCLASS.          "lcl_airplane IMPLEMENTATION
*-----*
*      CLASS lcl_cargo_plane DEFINITION
*-----*

CLASS lcl_cargo_plane DEFINITION INHERITING FROM lcl_airplane FINAL.

...

ENDCLASS.          "lcl_cargo_plane DEFINITION
*-----*
*      CLASS lcl_cargo_plane IMPLEMENTATION
*-----*

CLASS lcl_cargo_plane IMPLEMENTATION.

...

ENDCLASS.          "lcl_cargo_plane IMPLEMENTATION
*-----*
*      CLASS lcl_passenger_plane DEFINITION
*-----*

CLASS lcl_passenger_plane DEFINITION INHERITING FROM lcl_airplane FINAL.

...

ENDCLASS.          "lcl_passenger_plane DEFINITION
*-----*
*      CLASS lcl_passenger_plane IMPLEMENTATION
*-----*

CLASS lcl_passenger_plane IMPLEMENTATION.

...
```

Continued on next page

```
ENDCLASS.           "lcl_passenger_plane IMPLEMENTATION
```

Include BC401_SPC_S1_AGENCY

```
-----*
*      CLASS lcl_travel_agency DEFINITION
*-----
CLASS lcl_travel_agency DEFINITION FINAL CREATE PRIVATE.

PUBLIC SECTION.

CLASS-METHODS:
    get_instance
    RETURNING
        value(ro_instance) TYPE REF TO lcl_travel_agency.

METHODS:
    display_agency_partners,
    display_attributes.

PRIVATE SECTION.

DATA:
    mv_name      TYPE string,
    mt_partners TYPE TABLE OF REF TO if_partner.

CLASS-DATA:
    go_instance TYPE REF TO lcl_travel_agency.

METHODS:
    constructor
    IMPORTING
        iv_name TYPE string.

METHODS:
    on_partner_created FOR EVENT partner_created
        OF if_partner
        IMPORTING sender.

ENDCLASS.           "lcl_travel_agency DEFINITION
```

Continued on next page

```

*      CLASS lcl_travel_agency IMPLEMENTATION
*-----
CLASS lcl_travel_agency IMPLEMENTATION.

METHOD get_instance.

IF go_instance IS NOT BOUND.
  CREATE OBJECT go_instance
    EXPORTING
      iv_name = 'Travel&Smile Travel'.
ENDIF.

ro_instance = go_instance.

ENDMETHOD.          "get_instance

METHOD display_attributes.
  WRITE: / icon_private_files AS ICON,
         'Travel Agency:'(007), mv_name.
  SKIP.
  display_agency_partners( ).
ENDMETHOD.          "display_attributes

METHOD display_agency_partners.
  DATA:
    lo_partner TYPE REF TO if_partner.

  WRITE 'Here are the partners of the travel agency:'(008).
  ULINE.
  LOOP AT mt_partners INTO lo_partner.
    lo_partner->display_partner( ).
  ENDLOOP.

ENDMETHOD.          "display_agency_partners

METHOD constructor.
  mv_name = iv_name.
  SET HANDLER on_partner_created FOR ALL INSTANCES.
ENDMETHOD.          "constructor

METHOD on_partner_created.
  APPEND sender TO mt_partners.
ENDMETHOD.          "on_partner_created

ENDCLASS.

```

Continued on next page

Main Program SAPBC401_SPC_S1

```

REPORT  sapbc401_spc_s1.

TYPE-POOLS icon.

INCLUDE bc401_spc_s1_agency.
INCLUDE bc401_spc_s1_carrier.
INCLUDE bc401_spc_s1_rental.

DATA:
go_hotel      TYPE REF TO cl_hotel2,
go_agency     TYPE REF TO lcl_travel_agency,
go_vehicle    TYPE REF TO lcl_vehicle,
go_truck      TYPE REF TO lcl_truck,
go_bus        TYPE REF TO lcl_bus,
go_rental     TYPE REF TO lcl_rental,
go_passenger  TYPE REF TO lcl_passenger_plane,
go_cargo      TYPE REF TO lcl_cargo_plane,
go_carrier    TYPE REF TO lcl_carrier.

START-OF-SELECTION.
*****create travel_agency *****
*  CREATE OBJECT go_agency
*    EXPORTING
*      iv_name = 'Travel&Smile Travel'.

go_agency = lcl_travel_agency=>get_instance( ).

*****create hotel *****
CREATE OBJECT go_hotel
EXPORTING
iv_name = 'Sleep Well Hotel'
iv_beds = 345.

*****create rental *****
CREATE OBJECT go_rental
EXPORTING
iv_name = 'Happy Car Rental'.

*****create truck *****
CREATE OBJECT go_truck

```

Continued on next page

```

EXPORTING
  iv_make  = 'MAN'
  iv_cargo = 45.

***** create truck *****
CREATE OBJECT go_bus
EXPORTING
  iv_make      = 'Mercedes'
  iv_passengers = 80.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make  = 'VOLVO'
  iv_cargo = 48.

***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly Travel'.

***** Passenger Plane *****
CREATE OBJECT go_passenger
EXPORTING
  iv_name       = 'LH BERLIN'
  iv_planetype = '747-400'
  iv_seats     = 345
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc <> 0.
  WRITE:
    / icon_failure AS ICON,
    'Wrong plane type'.
ENDIF.

***** cargo Plane *****
CREATE OBJECT go_cargo
EXPORTING
  iv_name       = 'US Hercules'
  iv_planetype = '747-200F'
  iv_cargo     = 533
EXCEPTIONS
  wrong_planetype = 1.
IF sy-subrc <> 0.

```

Continued on next page

```
WRITE:  
/ icon_failure AS ICON,  
'Wrong plane type'.  
ENDIF.  
  
***** show attributes of all partners of travel_agency *****  
go_agency->display_attributes( ).  
  
* Optional part: Use method chain:  
*-----*  
* lcl_travel_agency=>get_instance( )->display_attributes( ).
```


Exercise 21: (Optional) Factory Classes and Friendship Relationships

Exercise Objectives

After completing this exercise, you will be able to:

- Define and implement a factory method in a factory class
- Define friendship relationships
- Use friendship relationships

Business Example

Instead of instantiating airplane objects in the main program directly, airplanes should be created by a factory class. The factory class should provide one single factory method. Depending on the input, the factory class decides inside this factory method whether a cargo plane or a passenger plane is to be created.

Template:

SAPBC401_SPC_S1

Solution:

SAPBC401_SPC_S2

Task 1:

In your program, define a factory class (suggested name: **LCL_AIR-PLANE_FACTORY**) with a public static factory method for airplanes (suggested name: **CREATE_AIRPLANE**).

The factory method should have suitably typed import parameters for name, plane type, cargo and seats. The latter two you should make optional.

It should also have a single returning parameter to return a reference to an airplane instance - either a cargo plane or a passenger plane.

1. Complete your program ZBC401_##_MAIN ((where ## is your two-digit group number) or copy the template program .
2. Edit the include which contains the classes for airplanes. Add the definition of the new class, there.
3. Within the class, define a public static factory method for airplanes. Define four import parameters for name, plane type, cargo and seats. Name and type them exactly as in the constructors of your airplane classes.

Continued on next page

Make the parameters for cargo and seats optional.



Hint: Use addition OPTIONAL to do so.

4. Define a returning parameter (suggested name: RO_AIRPLANE). How do you have to type the parameter so that the method can return a reference to either a cargo plane instance or a passenger plane instance?

Task 2:

Implement the factory method. Depending on the optional parameter that is supplied, create either a cargo plane instance or a passenger plane instance. Define a suitable exception and raise it if both optional parameters are supplied or none of them.

1. In the factory method, define two local reference variables. Type one with the class for cargo planes (LCL_CARGO_PLANE) and one with the class for passenger planes (LCL_PASSENGER_PLANE).
2. Implement an IF-structure in which you analyze the content of the import parameters for cargo and seats. Create either an instance of the cargo plane class or the passenger plane class. In the case of success, move the reference to the new object to the returning parameter.
3. In the definition of the factory class, add two exceptions: One which you raise if a wrong plane type was specified (suggested name: WRONG_PLANETYPE) and one which you raise if both or none of the optional parameters were supplied (suggested name: WRONG_PARAM_COMBINATION).

Task 3:

Restrict the creation of cargo planes and passenger planes so that they can only be created from inside the factory class.

1. Use the CREATE PRIVATE addition to restrict the instantiation of the classes for cargo planes and passenger planes.
Optionally, move the constructor definitions of both classes to the private section.

Continued on next page

2. Use the FRIENDS addition to allow the instantiation of the classes for cargo planes and passenger planes from inside the factory class.



Hint: You have to use statement **CLASS DEFINITION DEFERRED**. before the definition of the airplane classes. Otherwise the factory class (or at least its name) is not known to the syntax check when it reaches the FRIENDS addition.

Task 4:

Adjust your main program. Remove the direct instantiations of airplanes and replace them by calls of the factory method.

1. Replace the instantiations of airplanes with calls of the factory method.



Hint: If you have used exactly the same parameter names as in the constructors, you can keep the parameter passing!

2. Do you have to actually receive the references to the new objects?

3. Aktivate, test and debug your program.

Solution 21: (Optional) Factory Classes and Friendship Relationships

Task 1:

In your program, define a factory class (suggested name: **LCL_AIR-PLANE_FACTORY**) with a public static factory method for airplanes (suggested name: **CREATE_AIRPLANE**).

The factory method should have suitably typed import parameters for name, plane type, cargo and seats. The latter two you should make optional.

It should also have a single returning parameter to return a reference to an airplane instance - either a cargo plane or a passenger plane.

1. Complete your program ZBC401_##_MAIN ((where ## is your two-digit group number) or copy the template program .
 - a) Carry out this step in the usual manner. For more information, refer to the SAP Library.
2. Edit the include which contains the classes for airplanes. Add the definition of the new class, there.
 - a) See source text excerpt from the model solution.
3. Within the class, define a public static factory method for airplanes. Define four import parameters for name, plane type, cargo and seats. Name and type them exactly as in the constructors of your airplane classes.

Make the parameters for cargo and seats optional.



Hint: Use addition OPTIONAL to do so.

- a) See source text excerpt from the model solution.
4. Define a returning parameter (suggested name: **RO_AIRPLANE**). How do you have to type the parameter so that the method can return a reference to either a cargo plane instance or a passenger plane instance?
 - a) The parameter has to be typed with the common superclass of cargo planes and passenger planes, that is TYPE REF TO lcl_airplane.

See source text excerpt from the model solution.

Continued on next page

Task 2:

Implement the factory method. Depending on the optional parameter that is supplied, create either a cargo plane instance or a passenger plane instance. Define a suitable exception and raise it if both optional parameters are supplied or none of them.

1. In the factory method, define two local reference variables. Type one with the class for cargo planes (LCL_CARGO_PLANE) and one with the class for passenger planes (LCL_PASSENGER_PLANE).
 - a) See source text excerpt from the model solution.
2. Implement an IF-structure in which you analyze the content of the import parameters for cargo and seats. Create either an instance of the cargo plane class or the passenger plane class. In the case of success, move the reference to the new object to the returning parameter.
 - a) See source text excerpt from the model solution.
3. In the definition of the factory class, add two exceptions: One which you raise if a wrong plane type was specified (suggested name: WRONG_PLANETYPE) and one which you raise if both or none of the optional parameters were supplied (suggested name: WRONG_PARAM_COMBINATION).
 - a) See source text excerpt from the model solution.

Task 3:

Restrict the creation of cargo planes and passenger planes so that they can only be created from inside the factory class.

1. Use the CREATE PRIVATE addition to restrict the instantiation of the classes for cargo planes and passenger planes.
Optionally, move the constructor definitions of both classes to the private section.
 - a) See source text excerpt from the model solution.
2. Use the FRIENDS addition to allow the instantiation of the classes for cargo planes and passenger planes from inside the factory class.



Hint: You have to use statement **CLASS DEFINITION DEFERRED**. before the definition of the airplane classes. Otherwise the factory class (or at least its name) is not known to the syntax check when it reaches the FRIENDS addition.

- a) See source text excerpt from the model solution.

Continued on next page

Task 4:

Adjust your main program. Remove the direct instantiations of airplanes and replace them by calls of the factory method.

1. Replace the instantiations of airplanes with calls of the factory method.



Hint: If you have used exactly the same parameter names as in the constructors, you can keep the parameter passing!

- a) See source text excerpt from the model solution.
2. Do you have to actually receive the references to the new objects?
- Answer:** No. Through the message handling implemented earlier, the carrier instance takes care of all new airplane instances and aggregates them.
3. Aktivate, test and debug your program.
- a) Carry out this step in the usual manner. For more information, refer to the SAP Library.

Result

Include BC401_SPC_S2_CARRIER

```
CLASS lcl_airplane_factory DEFINITION DEFERRED.

*-----
*      CLASS lcl_airplane DEFINITION
*-----
CLASS lcl_airplane DEFINITION ABSTRACT.

...
ENDCLASS.          "lcl_airplane DEFINITION
*-----
*      CLASS lcl_airplane IMPLEMENTATION
*-----
CLASS lcl_airplane IMPLEMENTATION.

...
ENDCLASS.          "lcl_airplane IMPLEMENTATION
*-----
*      CLASS lcl_cargo_plane DEFINITION
```

Continued on next page

```

*-----*
*
*-----*
CLASS lcl_cargo_plane DEFINITION INHERITING FROM lcl_airplane FINAL
  CREATE PRIVATE
    FRIENDS lcl_airplane_factory.

...
ENDCLASS.          "lcl_cargo_plane DEFINITION
*-----*
*      CLASS lcl_cargo_plane IMPLEMENTATION
*-----*
CLASS lcl_cargo_plane IMPLEMENTATION.

...
ENDCLASS.          "lcl_cargo_plane IMPLEMENTATION
*-----*
*      CLASS lcl_passenger_plane DEFINITION
*-----*
CLASS lcl_passenger_plane DEFINITION INHERITING FROM lcl_airplane FINAL
  CREATE PRIVATE
    FRIENDS lcl_airplane_factory.

...
ENDCLASS.          "lcl_passenger_plane DEFINITION
*-----*
*      CLASS lcl_passenger_plane IMPLEMENTATION
*-----*
CLASS lcl_passenger_plane IMPLEMENTATION.

...
ENDCLASS.          "lcl_passenger_plane IMPLEMENTATION
*-----*
*      CLASS lcl_carrier DEFINITION
*-----*
CLASS lcl_carrier DEFINITION.

...
ENDCLASS.          "lcl_carrier DEFINITION
*-----*

```

Continued on next page

```

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

...
ENDCLASS.          "lcl_carrier IMPLEMENTATION

*-----
*      CLASS lcl_airplane_factory DEFINITION
*-----
CLASS lcl_airplane_factory DEFINITION.
PUBLIC SECTION.

CLASS-METHODS:
  create_airplane
    IMPORTING
      iv_name      TYPE string
      iv_planetype TYPE saplane-planetype
      iv_cargo     TYPE s_plan_car  OPTIONAL
      iv_seats     TYPE s_seatsmax OPTIONAL
    RETURNING
      value(ro_airplane) TYPE REF TO lcl_airplane
    EXCEPTIONS
      wrong_planetype
      wrong_param_combination.

ENDCLASS.          "lcl_airplane_factory DEFINITION
*-----
*      CLASS lcl_airplane_factory IMPLEMENTATION
*-----
CLASS lcl_airplane_factory IMPLEMENTATION.

METHOD create_airplane.
  DATA:
    lo_cargo      TYPE REF TO lcl_cargo_plane,
    lo_passenger  TYPE REF TO lcl_passenger_plane.

  IF iv_cargo IS NOT INITIAL AND iv_seats IS INITIAL.

    CREATE OBJECT lo_cargo
    EXPORTING
      iv_name       = iv_name
      iv_planetype = iv_planetype
      iv_cargo     = iv_cargo

```

Continued on next page

```

EXCEPTIONS
    wrong_planetype = 1.
IF sy-subrc <> 0.
    RAISE wrong_planetype.
ELSE.
    ro_airplane = lo_cargo.
ENDIF.

ELSEIF iv_cargo IS INITIAL AND iv_seats IS NOT INITIAL.

CREATE OBJECT lo_passenger
EXPORTING
    iv_name      = iv_name
    iv_planetype = iv_planetype
    iv_seats     = iv_seats
EXCEPTIONS
    wrong_planetype = 1.
IF sy-subrc <> 0.
    RAISE wrong_planetype.
ELSE.
    ro_airplane = lo_passenger.
ENDIF.

ELSE.
    RAISE wrong_param_combination.
ENDIF.

ENDMETHOD.          "create_airplane
ENDCLASS.           "lcl_airplane_factory IMPLEMENTATION

```

SAPBC401_SPC_S2

```

REPORT  sapbc401_spc_s2.

TYPE-POOLS icon.

INCLUDE bc401_spc_s2_agency.
INCLUDE bc401_spc_s2_carrier.
INCLUDE bc401_spc_s2_rental.

DATA:
    go_agency    TYPE REF TO lcl_travel_agency,
    go_hotel     TYPE REF TO cl_hotel2,

```

Continued on next page

```

go_vehicle    TYPE REF TO lcl_vehicle,
go_truck      TYPE REF TO lcl_truck,
go_bus        TYPE REF TO lcl_bus,
go_rental     TYPE REF TO lcl_rental,
*go_passenger TYPE REF TO lcl_passenger_plane,
*go_cargo      TYPE REF TO lcl_cargo_plane,
go_carrier    TYPE REF TO lcl_carrier.

START-OF-SELECTION.
***** create travel_agency *****
*  CREATE OBJECT go_agency
*    EXPORTING
*      iv_name = 'Travel&Smile Travel'.
go_agency = lcl_travel_agency->get_instance( ).

***** create hotel *****
CREATE OBJECT go_hotel
EXPORTING
iv_name = 'Sleep Well Hotel'
iv_beds = 345.

***** create rental *****
CREATE OBJECT go_rental
EXPORTING
iv_name = 'Happy Car Rental'.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
iv_make  = 'MAN'
iv_cargo = 45.

***** create truck *****
CREATE OBJECT go_bus
EXPORTING
iv_make      = 'Mercedes'
iv_passengers = 80.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
iv_make  = 'VOLVO'

```

Continued on next page

```

iv_cargo = 48.

***** Create Carrier ****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly-Travel'.

***** Passenger Plane ****
lcl_airplane_factory=>create_airplane(
EXPORTING
  iv_name      = 'LH BERLIN'
  iv_planetype = '747-400'
  iv_seats     = 345
EXCEPTIONS
  wrong_planetype      = 1
  wrong_param_combination = 2 .
IF sy-subrc = 1.
  WRITE:
    / icon_failure AS ICON,
    'Wrong plane type'.
ENDIF.

***** cargo Plane ****
lcl_airplane_factory=>create_airplane(
EXPORTING
  iv_name      = 'US Hercules'
  iv_planetype = '747-200F'
  iv_cargo     = 533
EXCEPTIONS
  wrong_planetype      = 1
  wrong_param_combination = 2 .

IF sy-subrc = 1.
  WRITE:
    / icon_failure AS ICON,
    'Wrong plane type'.
ENDIF.

***** show attributes of all partners of travel_agency ****
go_agency->display_attributes( ).
```



Lesson Summary

You should now be able to:

- Define abstract classes
- Define abstract methods
- Define final classes
- Define final methods
- Understand the special use of public read-only attributes
- Understand the term “navigation method”
- Chain functional methods
- Limit the visibility of the constructor
- Understand the term “factory method”
- Explain the “singleton pattern”
- Define friendship relationships between classes

Related Information

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

Lesson: Persistent Objects and OO-Transactions

Lesson Overview

This lesson shows how you can use ABAP Objects to store instances of global classes in database tables, permanently and independently of a specific program. This lesson provides an introduction to the concepts involved and a demonstration of how they can be applied in practice.



Lesson Objectives

After completing this lesson, you will be able to:

- Create persistent objects using the standard technique
- Load persistent objects using the standard technique
- Access components of persistent objects
- Create an object-oriented transaction (transaction code type)

Business Example

You want to use the Object Services and OO Transactions in your ABAP Objects programs.

Persistence Services

In principle, ABAP programs work with data and objects that are valid at runtime. They are transient – that is, temporary – and disappear when the program ends. If this data is to be stored permanently and independently of the program – that is, persistently – it must be stored in a database. (Theoretically, you could also use files at operating system level.)

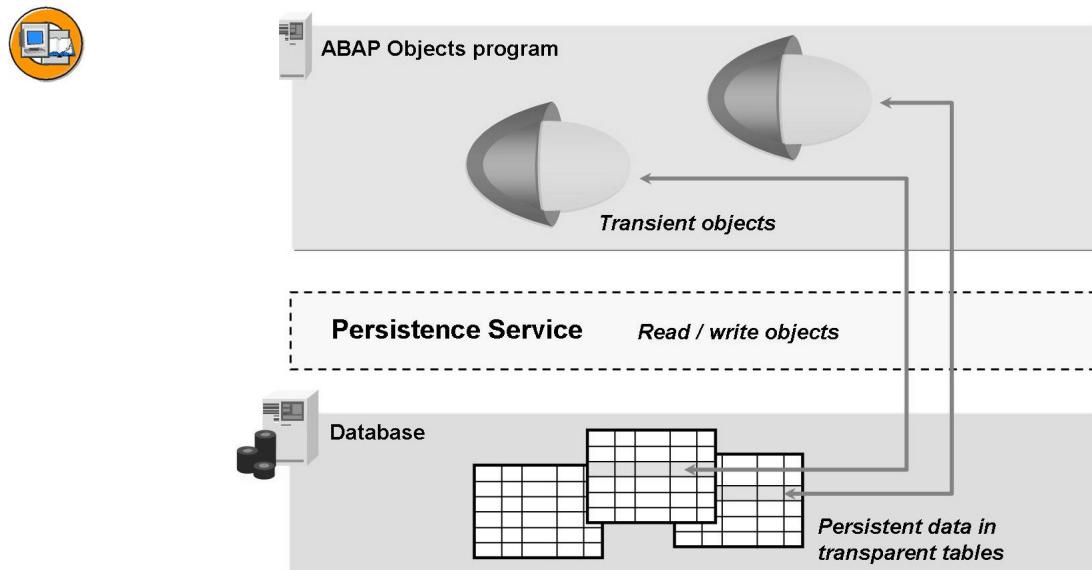


Figure 149: Simulation of an Object-Oriented Database Management System

For this purpose, the Persistence Services for ABAP Objects were introduced in the *SAP Web AS 6.10*. On request, these services write the current attribute values of objects defined as persistent to the associated transparent tables. On receiving another request, they import these values from the tables again and write the attributes to an object previously defined as persistent. When doing this, the Persistence Services use the ABAP Open SQL interface.

The Persistence Services are used with object-oriented programming techniques. In this way, you can simulate an object-oriented database management system using a relational database management system and ABAP Objects.

Persistent Objects as Instances of Special Global Classes

To use the Persistence Service for objects, their types must be created as persistent classes in the *Class Builder*. The term “persistent class” indicates that the **instances** of the class and their state are managed by the Persistence Service. (Global classes as such are generally persistent. The question is whether or not their instances are also persistent.)

To ensure that the instances of persistent classes are unique, they should contain key attributes. These can be typed either as worldwide unique identification numbers (object GUID) or as semantic keys.

When you create a persistent class ZCL_<name>, the *Class Builder* automatically generates methods for getting and setting its attributes. In addition, other Repository objects are also generated, including the class actor (also known as the class agent) ZCA_<name>. Programs must call the methods of this class to manage the instances of persistent classes, that is, persistent objects. This class

also performs the actual database access. It automatically inherits the required methods from the basis class ZCB_<name>. These methods can be redefined if necessary – for example, to extend database accesses. The class actor is a singleton instance and has a friendship relationship with the persistent class.

In the program, you need to define a reference variable with the type of the class actor and fill it with the reference from the static attribute AGENT. You can then create a new instance of the persistent class using its method CREATE_PERSISTENT. Its interface is created from the definition of the key attributes of the persistent class. If a persistent object of the same persistence class with the same key attributes already exists in the program, the class-based exception CX_OS_OBJECT_EXISTING is triggered. By default, the data is saved by asynchronous update. This process must be started using the COMMIT WORK statement. Thus, the instances of the persistent class do not become persistent objects until after that point.



```

...
DATA:
  go_carrier TYPE REF TO zcl_carrier,
  go_agent   TYPE REF TO zca_carrier.
...
go_agent = zca_carrier=>agent.

TRY.
  go_carrier = go_agent->create_persistent(
    i_carrid  = 'LH'
    i_carrname = 'Lufthansa' ).

  CATCH cx_os_object_existing.
  ...
ENDTRY.

go_carrier->...
COMMIT WORK.

```

Agent:
Singleton instance of class
ZCA_CARRIER
is friend of persistent class
ZCL_CARRIER

Factory Method:
new instance of persistent class,
data not yet on database

Figure 150: Generating and Writing to Persistent Objects

You can load a single persistent object (back) into a program using the GET_PERSISTENT method. When doing so, you must pass the key values to the interface parameters so that the object can be uniquely restored. If the program could not find the object, it raises the class-based exception CX_OS_OBJECT_NOT_FOUND.



```
...
DATA:
  go_carrier  TYPE REF TO zcl_carrier,
  go_agent    TYPE REF TO zca_carrier,
  lv_carrname TYPE s_carrname.
...
go_agent = zca_carrier=>agent.

TRY.
  go_carrier = go_agent->get_persistent( i_carrid = 'LH' ).
  lv_carrname = go_carrier->get_carrname( ).
  WRITE: 'LH: ', lv_carrname.
  CATCH cx_os_object_not_found.
  ...
ENDTRY.
```

Factory Method:
new instance of persistent class,
data are already on database

Figure 151: Reading Persistent Objects

A larger set of persistent objects can be retrieved using the *Query Service*. It is possible to provide a selection, very much like the WHERE-clause of a SELECT statement, as well as ordering criteria.



Creating Persistent Classes

1. Model the class for your persistent objects.
 2. Find or create a suitable transparent table that has a field for each attribute of the class. If necessary, adapt the attributes of the class to match the technical field attributes of the transparent table.
- Note:** There must be a primary key field in the table for each key attribute of the class or there must be a key attribute of the class for each primary key field in the table.
3. In the *Class Builder*, create a persistent class as described in the *SAP Library*. Define the persistence mapping to the table.

OO Transactions as a Transaction Type

In *transaction maintenance*, you can create a transaction code as an OO transaction. This means that you link the transaction code either to the transaction service of the ABAP Objects services for persistent objects or to a public method in a global or local class of a program. When calling this type of OO transaction, which is linked to an instance method, the system automatically creates an instance of the class in its own internal session and then executes this instance method.

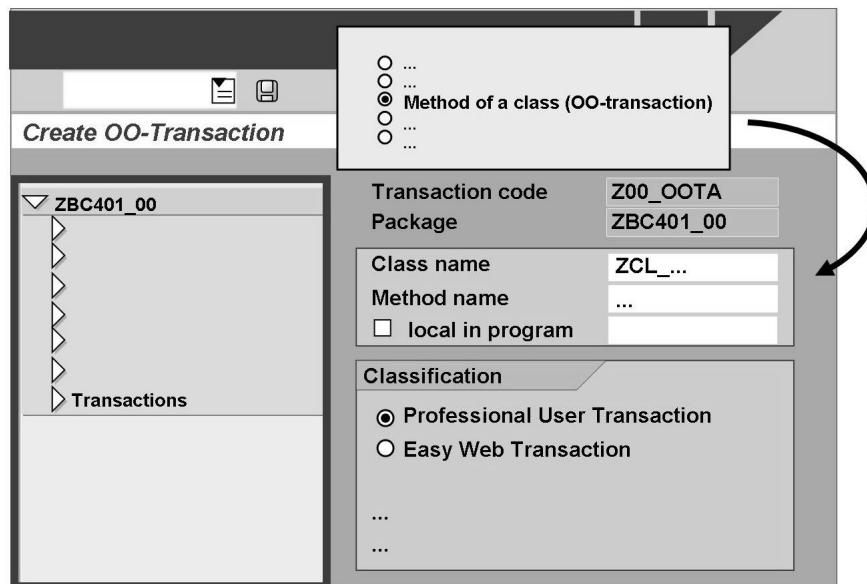


Figure 152: Creating an OO Transaction Code

If the *OO Transaction Model* flag is set, the transaction code is linked to the transaction service of the ABAP Objects services. Conversely, if it is not set, you can call any method in a global or local class.

Users therefore now have the opportunity to call methods themselves in the usual way (through menu options, command field entries, and so on).



Caution: Global classes in ABAP Objects cannot contain screen definitions. Thus, when you create a link to a global class, note the following:

- The constructor cannot contain any IMPORTING parameters.
- ABAP list displays cannot be processed.

Therefore, you may have to resort to methods in **local** program classes. Naturally, you can then use all the conventional dialog programming techniques.



Creating an OO Transaction

1. Call the usual dialog for creating a transaction code and enter a short text.
2. Choose the option *Method of a Class (OO Transaction)* as an *Initial Object*.
3. Leave the *OO Transaction Model* flag unchecked if you want to link the transaction code to a normal instance method.
4. Enter the *Class Name* and *Method*. Set the *Local in Program* flag and enter the program name, if you are using a local class.
5. Save the transaction code.



Lesson Summary

You should now be able to:

- Create persistent objects using the standard technique
- Load persistent objects using the standard technique
- Access components of persistent objects
- Create an object-oriented transaction (transaction code type)

Related Information

For more information, see the *SAP Library*.



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
- Create a simple ALV Grid Control
- React to a double-click on an ALV Grid Control with a handler method
- Explain the fundamentals of the BAdI technology and implement a BAdI
- Define abstract classes
- Define abstract methods
- Define final classes
- Define final methods
- Understand the special use of public read-only attributes
- Understand the term “navigation method”
- Chain functional methods
- Limit the visibility of the constructor
- Understand the term “factory method”
- Explain the “singleton pattern”
- Define friendship relationships between classes
- Create persistent objects using the standard technique
- Load persistent objects using the standard technique
- Access components of persistent objects
- Create an object-oriented transaction (transaction code type)



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.

4. For what purpose is the technique of persistent objects used?

Choose the correct answer(s).

- A To encapsulate database accesses in an object-oriented manner.
- B Exclusively to implement database accesses in an object-oriented context (Open SQL is not allowed in ABAP Objects).
- C To apply an object-oriented model for business objects (creating, using, writing, and reading business objects).
- D To extend SAP standard classes (which have generally been migrated in the *SAP Web AS 6.10*).
- E For authorization checks (accessing components of persistent objects is only possible using agents).



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

The creation of the first instance and the call of the first method can be realized using an OO transaction. However, certain restrictions apply.

Additional information is available in the SAP Library.

3. Which of the following statements are correct?

Answer: C, D, H, I, J

4. For what purpose is the technique of persistent objects used?

Answer: A, C

Additional information is available in the SAP Library.

Internal Use SAP Partner Only

Unit 4

Exception Handling and RTTS

Unit Overview

This last unit will look at the new exception concept and the language's features for RTTS (Run Time Type Services), thereby demonstrating how closely the procedural and object-oriented parts of ABAP Objects are connected. To fully exploit the available functions, you must therefore program in a partially object-oriented manner, even if you use the concepts in purely procedural program units.



Unit Objectives

After completing this unit, you will be able to:

- Create exception classes
- Raise class-based exceptions in ABAP Objects programs
- Propagate class-based exceptions in ABAP Objects programs
- Handle class-based exceptions in ABAP Objects programs
- Map class-based exceptions in ABAP Objects programs to each other
- Query type attributes at runtime
- Create types dynamically

Unit Contents

Lesson: Exception Handling in ABAP Objects	370
Procedure: Defining Global Exception Classes	377
Exercise 22: Class-Based Exceptions.....	391
Exercise 23: (Optional) Mapping Exceptions to Each Other.....	407
Lesson: RTTS (Run Time Type Services)	420
Exercise 24: RTTI - Querying Object Attributes	429

Lesson: Exception Handling in ABAP Objects

Lesson Overview

This lesson provides a comprehensive introduction to the new class-based exception concept in ABAP Objects. The essential attributes and activities are presented in detail and then applied in practical examples for *SAP Web Application Server 6.20*.



Lesson Objectives

After completing this lesson, you will be able to:

- Create exception classes
- Raise class-based exceptions in ABAP Objects programs
- Propagate class-based exceptions in ABAP Objects programs
- Handle class-based exceptions in ABAP Objects programs
- Map class-based exceptions in ABAP Objects programs to each other

Business Example

You should use the new exceptions concept in your ABAP Object programs.

Class-Based Exception Handling – the Concept

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 AS 6.10* introduced a new ABAP Objects exception concept, parallel to the existing concept based on sy-subrc. Exceptions and exception handling are now based on classes. This concept includes the concepts that preceded it but also enhances (and thus replaces) them.



Hint: Note that 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. In particular, all previously catchable runtime errors can now be handled as class-based exceptions.

Exception Classes and Exception Objects

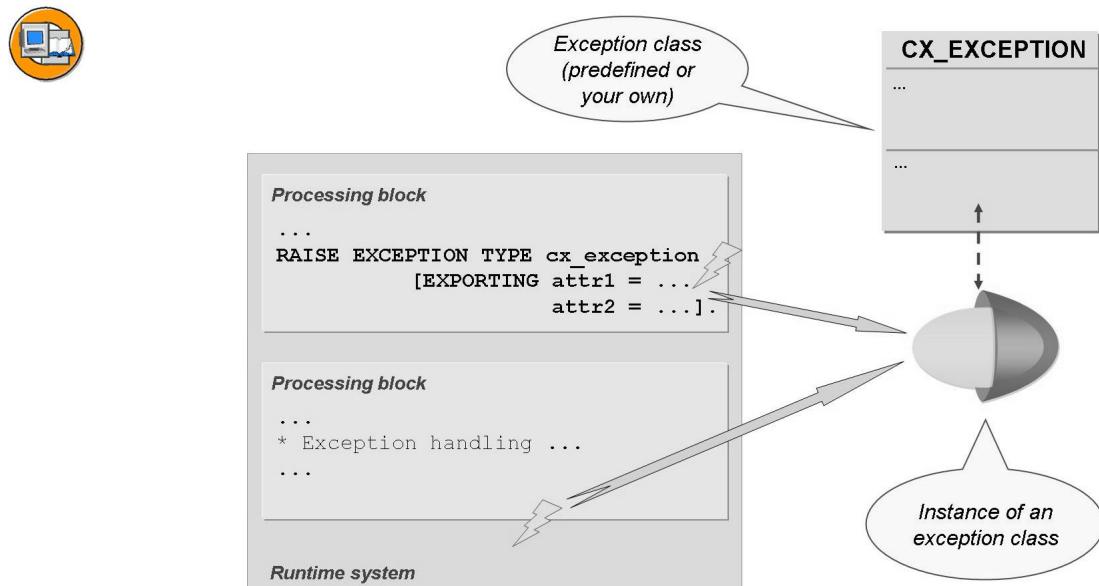


Figure 153: An Overview of the Class-Based Exception Concept

In the new exception concept, an exception is represented by an **exception object**. An exception object is an instance of an **exception class**. The attribute values of the exception object contain information about the respective error situation. **Raising** a class-based exception normally means instantiating an exception class and setting the attributes. **Handling** a class-based exception normally means getting access to the exception object and evaluating its attribute values.

Class-based exceptions are raised either by the **RAISE EXCEPTION** statement or by the runtime environment. They are caught and handled with the **TRY...CATCH...ENDTRY** structure.

Properties of Exception Classes

You can define exception classes yourself, but 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 names of global exception classes always start with <namespace>CX_. Those used by the runtime environment start with CX_SY_.

It is recommended to start the names of local exception classes with lcx_.

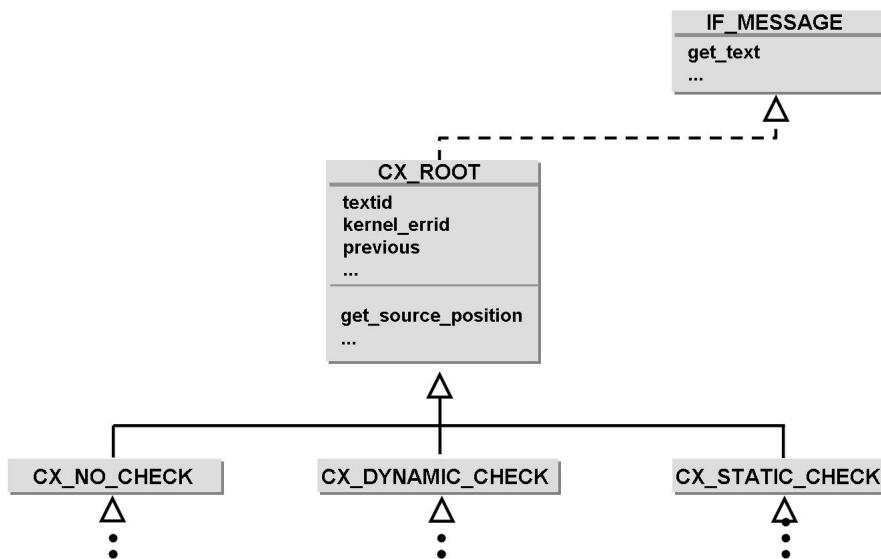


Figure 154: Exception Classes: The Inheritance Hierarchy

All exception classes are derived from the one exception class **CX_ROOT**. Therefore, it is possible to generically access any exception object through a reference variable typed REF TO CX_ROOT.

However, it is not allowed to inherit directly from CX_ROOT. Any new exception class has to be derived (directly or indirectly) from one of its subclasses **CX_NO_CHECK**, **CX_DYNAMIC_CHECK**, or **CX_STATIC_CHECK**. Through this, all exception classes are subdivided into three groups. Depending on the group a given exception belongs to, it is treated differently by syntax check and runtime environment. This will be discussed later. The default group is CX_STATIC_CHECK, which ensures maximum syntax check and program stability. The other groups should only be used in special cases.

Through their inheritance, all exception classes provide a number of common attributes and methods, among them the GET_SOURCE_POSITION method, which returns the name of the main program and (if relevant) the names of the include program and the line number in the source code where the exception occurred. The GET_TEXT method returns an exception text in the form of a string. This method is not defined in CX-ROOT directly but in Interface IF_MESSAGE, which is implemented by CX-ROOT. Other components of CX-ROOT will be discussed later.

Handling of Class-Based Exceptions

An exception can only be handled if the statement that raised it is enclosed in a TRY-ENDTRY control structure. The exception is then handled using the CATCH statement in the TRY-ENDTRY structure.

The TRY-ENDTRY Control Structure

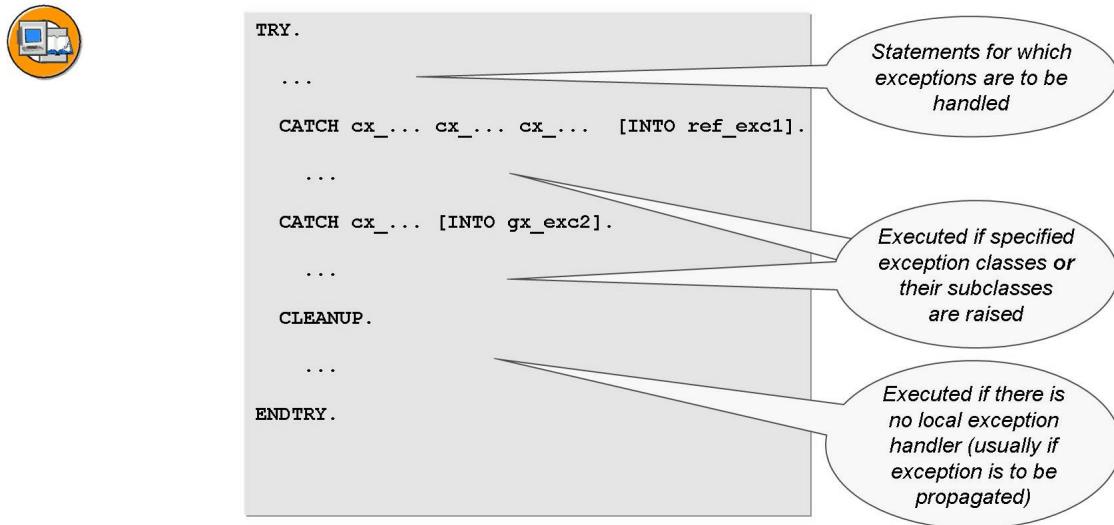


Figure 155: Structure of a TRY-ENDTRY Block

The TRY block contains the statements for which the exceptions should be handled. A CATCH block contains the exception handler that is executed if a specified exception has occurred in the associated TRY block.

Like all control structures in ABAP Objects, TRY-ENDTRY structures can be **nested** to any depth. In particular, the TRY block, CATCH blocks, and the CLEANUP block can contain complete TRY-ENDTRY structures themselves.

You can specify any number of exception classes to the CATCH statement. In this way, you define an exception handler for all these exception classes and their subclasses.

If an exception occurs, the system searches first for a matching CATCH statement in the TRY-ENDTRY structure that immediately surrounds the statement. It searches through its CATCH blocks in the order specified. A matching CATCH statement mentions either the exception class itself or one of its superclasses. If it finds one, it leaves the TRY block and navigates directly to this handler.

If it does not find one, the system gradually searches outwards in all the surrounding TRY-ENDTRY structures. If it cannot find a handler within the same procedure, the system tries to **propagate** the exception to the calling program. (This will be discussed in more detail later.)

If a TRY-ENDTRY structure contains a **CLEANUP** block, this block is executed when the TRY-ENDTRY structure is exited because the system cannot find a handler within the TRY-ENDTRY structure itself but in a surrounding TRY-ENDTRY structure or in a calling program.

Analyzing the Exception Object

To analyze the exception object in an exception handler, you have to use the CATCH statement in the form **CATCH ... INTO ...**. After the optional addition INTO, you have to specify a suitably typed object reference. This can either be a reference to the exception class itself or to any of its superclasses.

Just before the CATCH block is executed, the system will fill this reference variable so that it points to the exception object describing the present error situation.

The following picture gives an example:

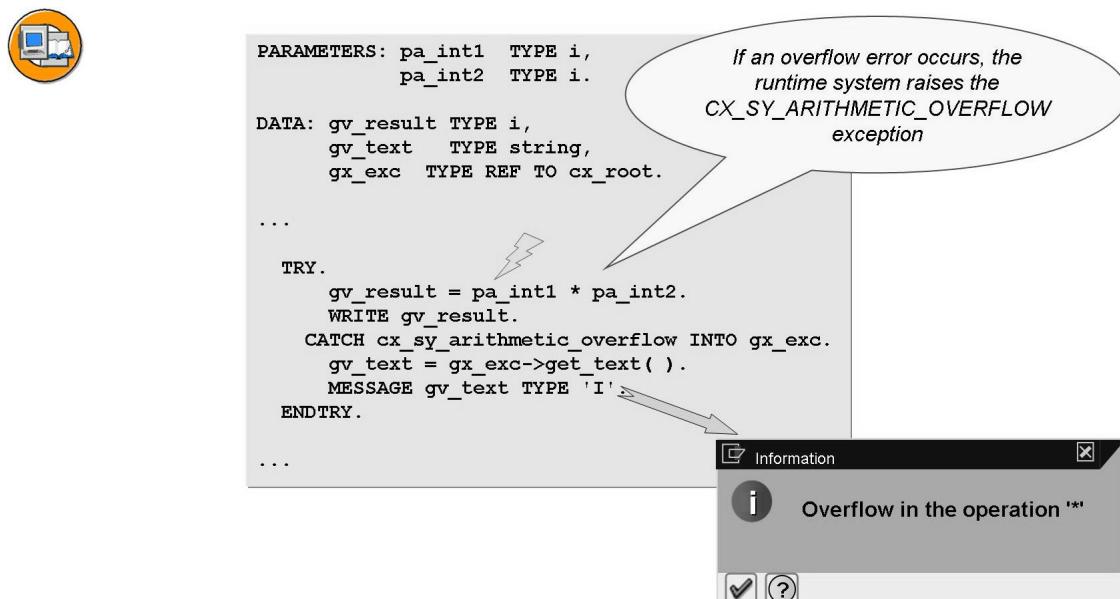


Figure 156: Example Syntax for Handling Predefined Exceptions

If the value range for data type I is exceeded in the calculation, the runtime system raises the exception CX_SY_ARITHMETIC_OVERFLOW. This exception is handled in the implemented CATCH block. The reference to the appropriate instance is stored in the data object GO_EXC. The handler can access the instance's exception text using the functional method GET_TEXT. The text is stored in the data object GV_TEXT with the type STRING and then displayed as an information message.

Of course, any other public component of the exception object could also be accessed during exception handling. But this depends, as usual, on the static type of the reference variable. In the example above, the reference variable is typed with REF TO CX_ROOT, which restricts access to the components defined in CX_ROOT. If you want to access the more specific components of the

exception object you need a more specific type for the reference variable. Other options would have been the superclass CX_SY_ARITHMETIC_ERROR and super-superclass CX_DYNAMIC_CHECK.

The *ABAP keyword documentation* for each statement lists the classes whose exceptions may occur when that statement is executed. The following section will show you how to create your own global exception classes.

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 debugging mode.

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.

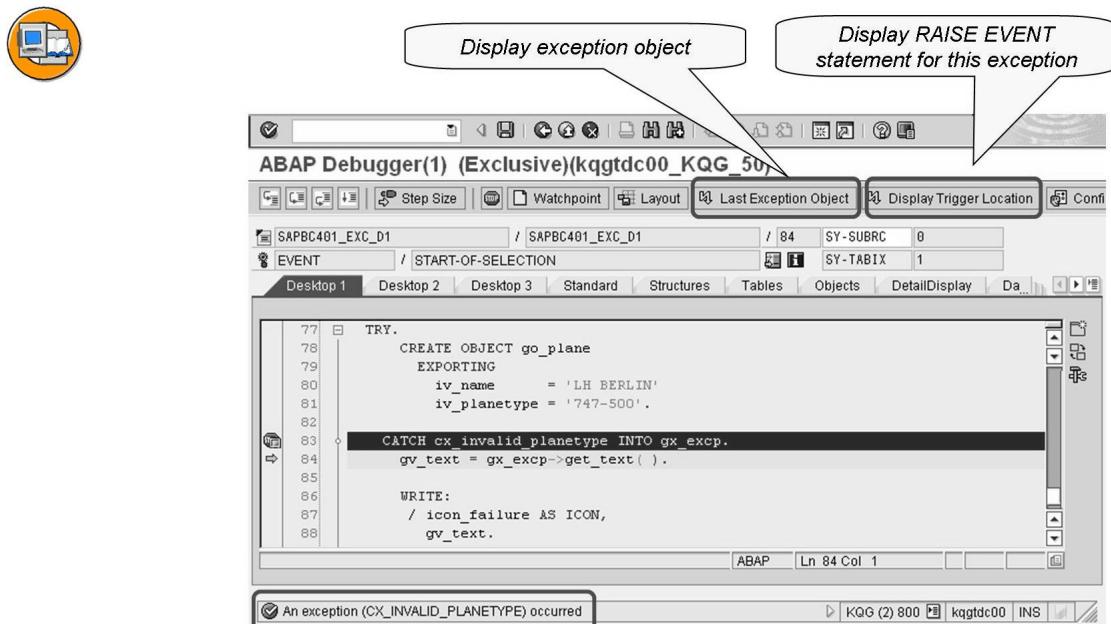


Figure 157: Class-Based Exceptions in Debugging Mode

Two pushbuttons appear if an exception occurred: They allow you to analyze the exception object and to navigate to the point in the source code where the exception occurred.



Hint: For performance reasons, the system does not actually create exception objects if the INTO addition is missing from the respective CATCH statement. In such cases, the pushbutton *Last Exception Object* will normally not appear.

You can make the *ABAP Debugger* to always create the exception object by choosing the respective option in the debugger settings.



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 the prefix **CX_**. Choose the option *Exception Class* as the *Class Type*.

 **Note:** As of SAP NW AS 6.40, you can still select the “with Message Class” checkbox when you create an exception class. Afterwards, the texts that should appear when exceptions occur can be taken from the messages classes identified in the message statement. (You will find these known messages and message classes in table T100.)

In previous releases (<6.40) the exception texts were stored solely in the OTR (Online Text Repository).

 **Note:** When you create the exception class, do not change the default entry for *Inherits from* unless you are fully aware of the consequences of doing so.

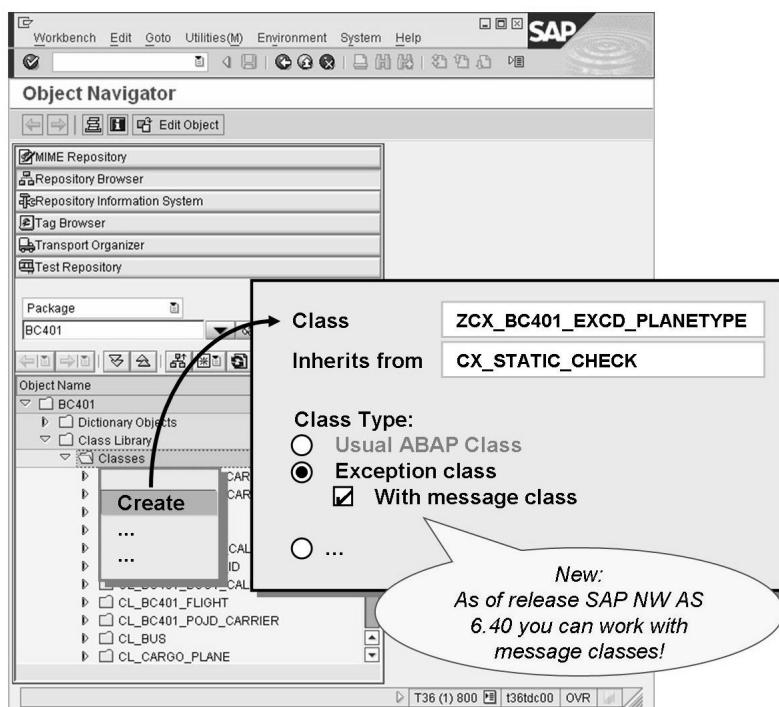


Figure 158: Creating Global Exception Classes

Continued on next page

2. Define additional attributes in your exception class if necessary (for example, for generic extensions to exception texts).



Hint: If such an attribute is **public**, the *Class Builder* automatically adjusts the **instance constructor** of the exception class:

A new input parameter is generated by which a value can be set for this attribute when raising the exception. The import parameter is generated with the same name as the attribute. By default the parameter is optional. You may declare it mandatory by editing the constructor's signature.

3. Save as many exception texts as you need. When you do so, you can insert your attributes as parameters in the static 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.

Continued on next page

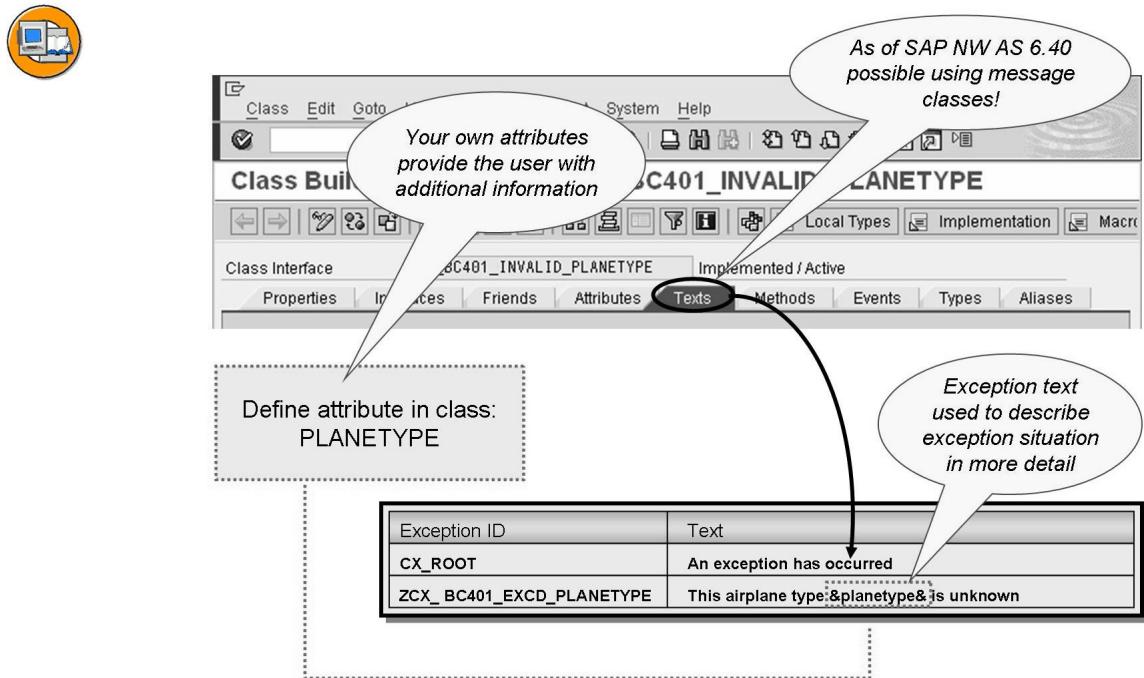


Figure 159: Define Variable Exception Texts



Note: The exception texts for global exception classes and their translations are stored in the *Online Text Repository (OTR)* in releases < SAP NW AS 6.40. 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 method GET_TEXT then exports this text, replaces any text parameters with the contents of the relevant attributes as necessary, and returns the text as a character string.

4. Activate your exception class.

Raising Class-Based Exceptions

Raising class-based exceptions is done with statement **RAISE EXCEPTION**. There are two variants of this statement:

Variants of Statement RAISE EXCEPTION ...

RAISE EXCEPTION TYPE <exception_class> [EXPORTING ...] .

This creates a **new exception object**, which is an instance of class <exception_class>.



RAISE EXCEPTION <object_ref>.

This statement uses an **existing exception object**, namely the one <object_ref> points to.

The exception object was either created directly, using a CREATE OBJECT statement or, more commonly, was caught in a previous CATCH ... INTO ... statement and now should be passed on.

The following figure gives an example for each variant:

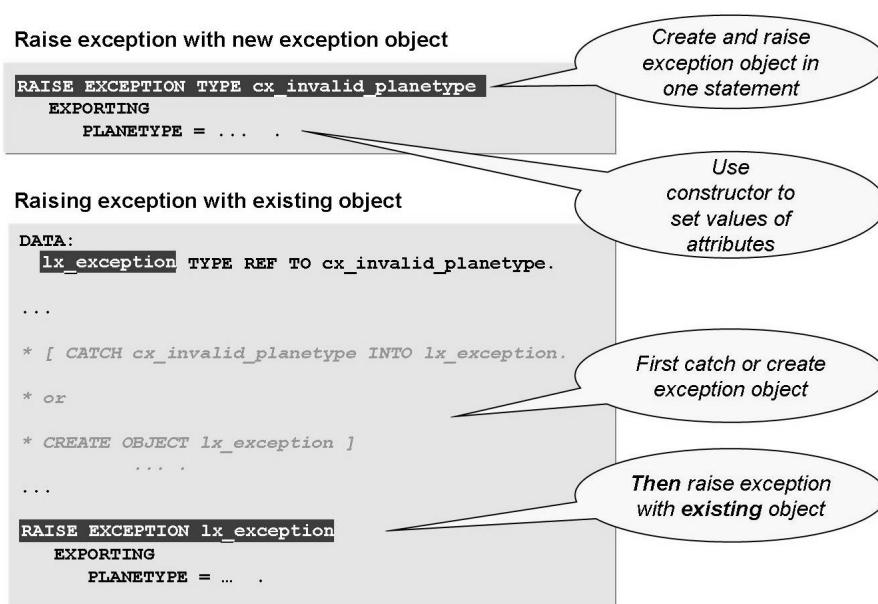


Figure 160: Variants of Statement RAISE EXCEPTION ...

When using the first variant, it is possible to provide values for the constructor parameters using the **EXPORTING** addition. One of these parameters is used to set the exception text in the new exception object.

Setting the Exception Text

All exception classes offer an optional parameter TEXTID in their constructors. This parameter is used if more than one message text is available in the exception class and you do not want to raise the exception with the default text.



Figure 161: Setting the Exception Text

For each text defined on the *Texts* tab, the *Class Builder* generates a public constant of the same name. Other such constants might also be inherited from the exception class's superclasses.

By default, the exception is raised with the text that has the same name as the exception class. To raise the exception with another text, you have to use the corresponding constant as actual parameter for constructor parameter TEXTID.

→ **Note:** The type of the constants (and of parameter TEXTID in the constructor) depends on whether the exception class was created with the *With message class* flag or not.

If the exception class still uses OTR texts, the constants are of type CHAR and contain a technical ID of the OTR text (GUID). If the exception class uses a message class, the constants are structures containing the name of the message class, the message number, and so on.

Propagating Class-Based Exceptions

Exceptions that occur in procedures do not necessarily need to be handled where they occur; they can be propagated to the caller 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.



Propagation I:

- Exception occurs in method (constructor)
- Handled in calling program (main program)

```
DATA:  
  go_plane TYPE REF TO lcl_airplane,  
  gx_exc   TYPE REF TO cx_exc.  
  
...  
  
TRY.  
  CREATE OBJECT go_plane  
    EXPORTING ...  
  ...  
  CATCH cx_exc INTO gx_exc.  
  ...  
ENDTRY.
```

```
...  
METHODS constructor  
  IMPORTING ...  
  RAISING cx_exc.  
...
```

```
METHOD constructor.  
...  
RAISE EXCEPTION TYPE cx_exc  
  EXPORTING ...  
...  
ENDMETHOD.
```

*RAISING addition
needed for most
exceptions*

-----> program flow
-----> create exception
-----> access to exception

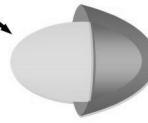


Figure 162: Propagating Class-Based Exceptions

To propagate an exception from a procedure, you generally use the RAISING addition when you are defining the procedure interface. In methods of local classes and subroutines, specify the RAISING addition directly when you are defining the procedure:

```
METHODS meth_name ... RAISING cx_... cx_... or  
FORM subr_name ... RAISING cx_... cx_...
```

After RAISING, list the exception classes whose instances are to be propagated. In methods of global classes, the exception classes whose instances are to be propagated are entered in the exception table of the method in the *Class Builder*. You also need to set the *Exception Class* flag for each exception table. The process is similar for function modules. To set the indicator in the *Function Builder*, choose the *Exceptions* tab.



Hint: Note that a single method, form, or function module can only raise one type of exceptions: class-based or conventional exceptions.

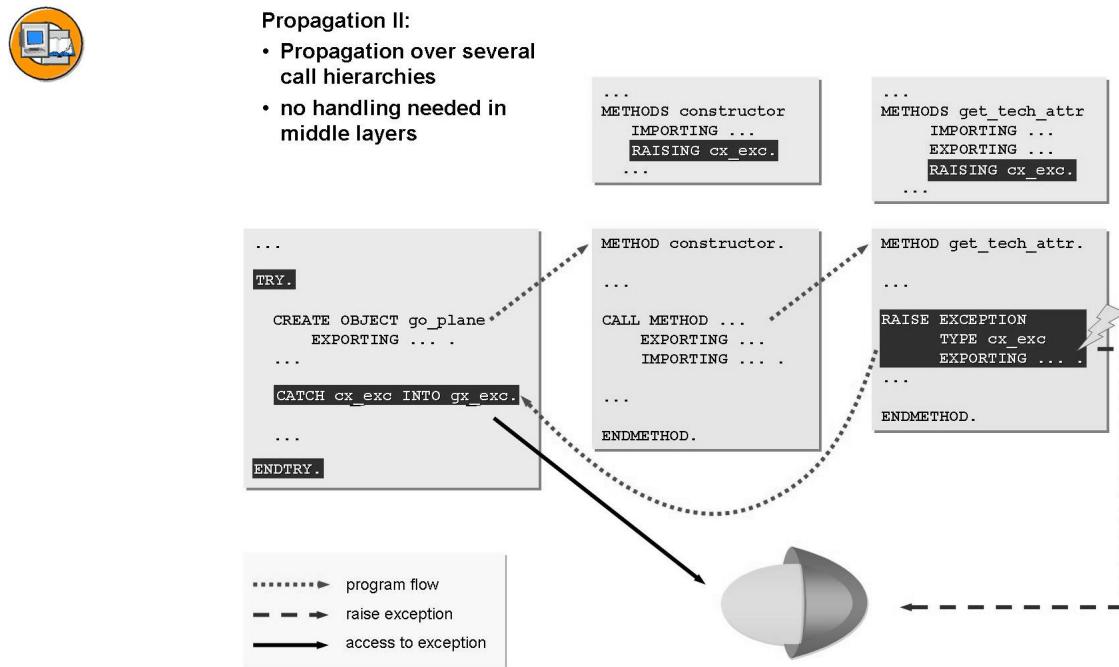


Figure 163: Propagating Exceptions Over Several Hierarchy-Levels

Propagation of class-based exception does not require that the exception actually handled by the calling method. The caller may pass on the exception to its own caller without even looking at it.

In the example above, the constructor propagates the exception `cx_exc` which is raised by method `GET_TECH_ATTR`. When the exception is raised, the program flow jumps directly to the `CATCH` block in the main program.

→ **Note:** If the constructor had implemented the (optional) `CLEANUP` block, this block would be executed before the `CATCH`-block of the main program.

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.

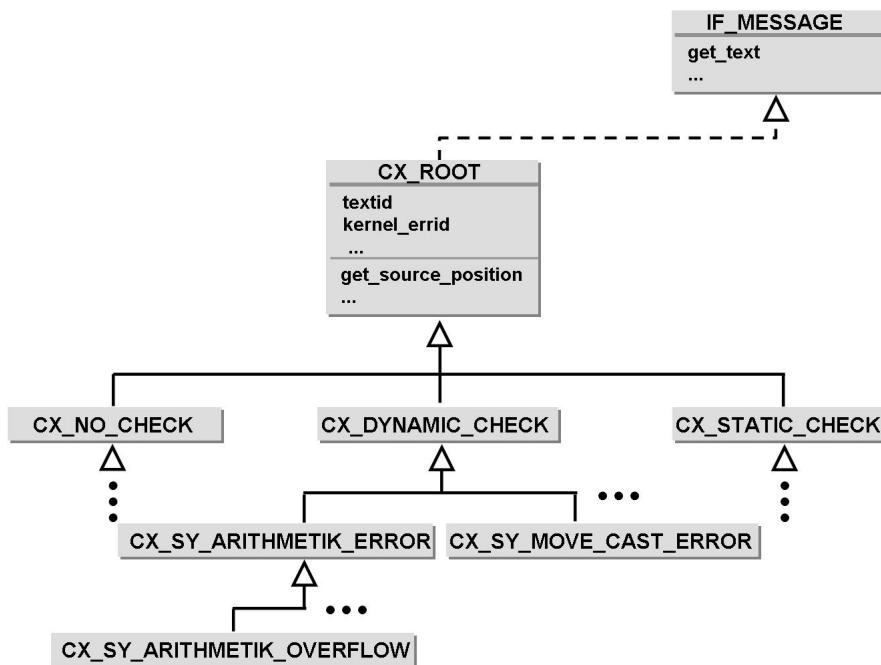


Figure 164: Integration of Standard Exceptions in the Runtime System

The choice of the superclass has an influence on the way a given exception is handled by the syntax check as well as by the runtime environment.

CX_STATIC_CHECK

If an exception class inherits (directly or indirectly) from CX_STATIC_CHECK, the relevant exception **must** either be handled, **or** propagated using the RAISING addition. If this is not the case, the **syntax check** displays a warning.

When defining new global exception classes, CX_STATIC_CHECK is defined as the superclass by default.

CX_DYNAMIC_CHECK

For subclasses of CX_DYNAMIC_CHECK, there will be no warning from the syntax check if they are neither handled nor propagated with the RAISING addition.

But if an exception is raised at runtime and it is neither handled nor propagated, the system will end the program with a **runtime error** (dump).

→ **Note:** 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.

CX_NO_CHECK

For subclasses of CX_NO_CHECK, the corresponding exceptions **cannot be propagated explicitly using the RAISING addition**. If such exceptions are not handled in the processing block where they occur they are **automatically propagated**. If they are not handled by the calling processing block, they are automatically propagated further on to the highest call hierarchy level. If they are not handled on the top level either, a runtime error will occur at the point where they were raised.

Some predefined exceptions with the prefix CX_SY_... for error situations in the runtime environment are subclasses of CX_NO_CHECK.

Exception Handling in Detail

After an exception was caught in a CATCH statement, there are different ways to handle it (besides stopping the program):



1. Continue program behind ENTRY-statement after ...
 - a) ... ignoring the exception (do nothing)
 - b) ... issuing a warning
 - c) ... writing to a protocol
 - d) ... correcting the situation
 - e) ... etc.
2. Remove the cause of the error and start again ...
 - a) ... from the beginning of the corresponding TRY block (using statement RETRY – new as of SAP NW 7.0 EhP 2)
 - b) ... from where the exception occurred (using statement RESUME – new as of SAP NW 7.0 EhP 2).
3. Raise and propagate an exception
 - a) The same exception (object) again (using RAISE EXCEPTION <obj_ref>)
 - b) Anew exception (using RAISE EXCEPTION TYPE <exc_class>).

The first option, continuing the program after having taken more or less action, is the easiest to understand and we will not discuss it further. But we will look into more detail about the second and third option.

Retry After an Exception



Hint: This technique is new with Release SAP NW 7.0 EhP 2.

When handling an exception in a CATCH block, you can use the **RETRY** statement to jump back to the TRY statement of the respective TRY-ENTRY structure, for example, if the cause for the exception was removed during the handling.



RETRY:

- jump to TRY-statement
- re-execute coding

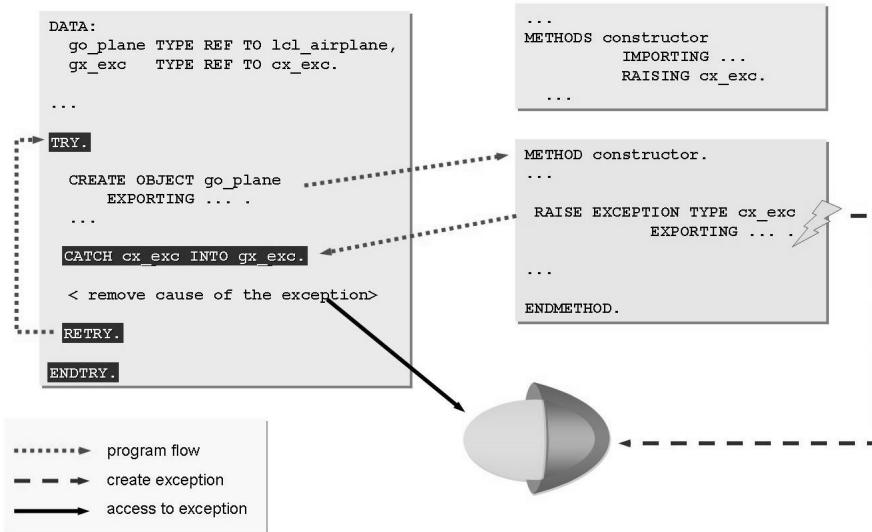


Figure 165: RETRY statement

In the example above, the exception raised and propagated by the constructor is caught by the main program. After analyzing the exception object and correcting the error situation, the main program repeats the whole TRY Block using the RETRY statement.



Caution: RETRY should be used with some caution. If you do not remove the cause of the exception properly, you will easily end up with an endless loop!

Resumable Exceptions



Hint: This technique is new with Release SAP NW 7.0 EhP 2.

With the **RESUME** statement, you can resume a program immediately after the statement that raised the exception. The following prerequisites exist for this:

1. The exception has to be caught with **CATCH BEFORE UNWIND ...**. This ensures that the context of the exception is kept alive for a possible RESUME. If, after a CATCH BEFORE UNWIND ..., the exception is not resumed, the system deletes the context of the exception when the end of the CATCH block is reached.
2. The exception has to be raised with the **RAISE RESUMABLE ...** variant of the RAISE EXCEPTION statement. This ensures that the raising processing lock is prepared for the resume.
3. If the exception is propagated, it has to be marked resumable on all hierarchy levels. This is done by the addition **RAISING RESUMABLE(...)** with the name of the exception class inside the brackets. This ensures that all methods that propagate the exception are prepared for a possible resume.



Hint: Respective checkboxes are available In *Class Builder* and *Function Builder* when defining the signature of a method or the interface of a function module.

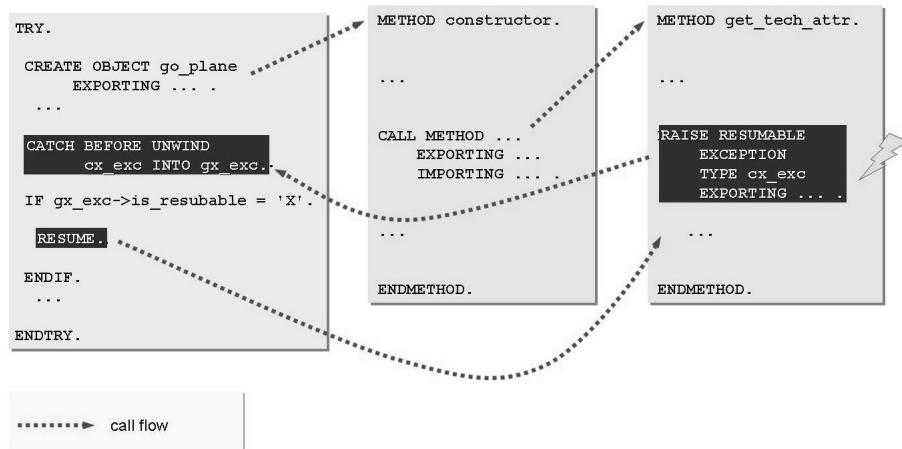
The handler of a given exception is able to find out at runtime whether a given exception was raised and propagated resumable or not. All exception objects provide the public instance attribute **IS_RESUMABLE**, which is set to 'X' or '' by the framework, depending on how the exception was raised and propagated. RESUMING a non-resumable exception will cause a runtime error (exception class **CX_SY_ILLEGAL_HANDLER**).

**Resumable Exceptions:**

- Resume after Exception
- Prerequisites:
 - raised resumable
 - propagated resumable

```
...
METHODS constructor
IMPORTING ...
RAISING
RESUMABLE(cx_exc).
...
```

```
...
METHODS get_tech_attr
IMPORTING ...
EXPORTING ...
RAISING
RESUMABLE(cx_exc).
...
```

**Figure 166: Resume Execution After a Resumable Exception**

In the example above, the exception is raised and propagated by method GET_TECH_ATTR and further propagated by the constructor. All raising and propagating is done resumable. The main program handles the exception with CATCH BEFORE UNWIND ..., checks that the exception indeed is resumable and then issues the RESUME statement. The execution of GET_TECH_ATTR is resumed right after the RAISE RESUMABLE EXCEPTION statement.



Hint: In this example, you can see very clearly the meaning of “keeping the context of the exception”. Without the BEFORE UNWIND addition, the system would delete the newly created instance before executing the CATCH block. Resuming the GET_TECH_ATTR method and the CONSTRUCTOR would not be possible.

Re-Raising Exceptions

Raising class-based exceptions is done with statement **RAISE EXCEPTION**. There are two variants of this statement:

RAISE EXCEPTION TYPE <exception_class> [EXPORTING ...]

This statement creates a **new exception object**, which is an instance of class <exception_class>. Optionally, values can be provided for the constructor using the EXPORTING addition.

RAISE EXCEPTION <object_ref>

This statement uses an **existing exception object**, namely the one <object_ref> points to.

This exception object was either created directly, using a CREATE OBJECT statement or, more commonly, was caught in a previous CATCH ... INTO ... statement and is to be passed on explicitly to the caller.



Propagation III:

- exception handled in middle method (constructor)
- existing exception object explicitly re-raised

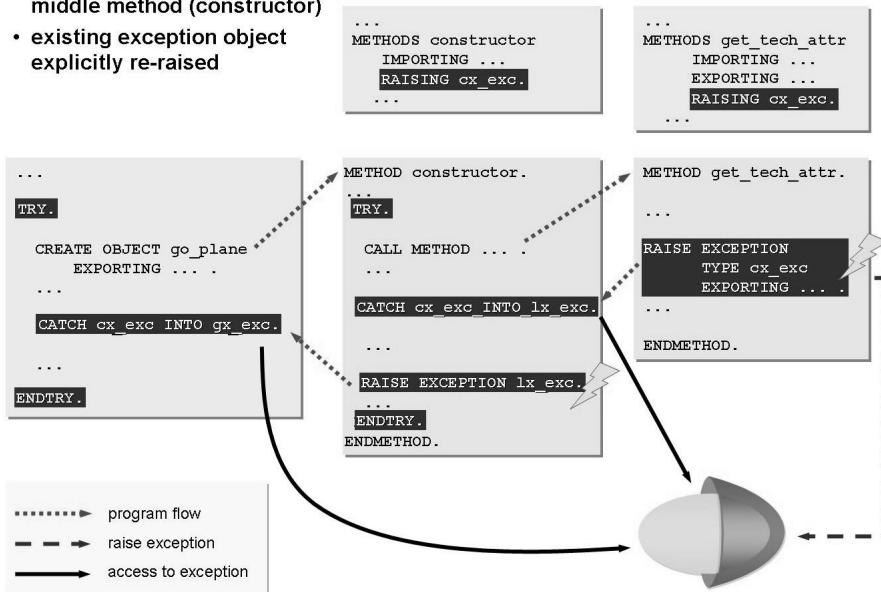


Figure 167: Re-Raising an Existing Exception Object

In the above example, the constructor catches the exception CX_EXC raised by the GET_TECH_ATTR method. After some analyzing the exception object, maybe some clearing up, issuing a warning, and so on, it decides to pass the exception on to the main program. There it is handled again.

Mapping Exceptions to Each Other

After catching an exception, the program can decide to raise a second exception, and so on. Normally, an exception object becomes invalid once the CATCH block is left. Therefore, the handler on top level only has access to the last exception object. But there is a way to “chain” exception objects, that is, to let one exception object point to the one before, which points to the one before and so on. The handler of an exception can then follow the chain and evaluate each exception object in the sequence.

All exception classes provide a public instance attribute PREVIOUS. The attribute is typed with REF TO CX_ROOT, thus can point to arbitrary exception objects. The constructors of all exception classes have an import parameter PREVIOUS of the same type, which can be used to link an existing exception object to a new one.



Mapping exceptions:

- first exception is handled in middle method (constructor)
- second exception raised there
- two objects linked together (via attribut 'previous')

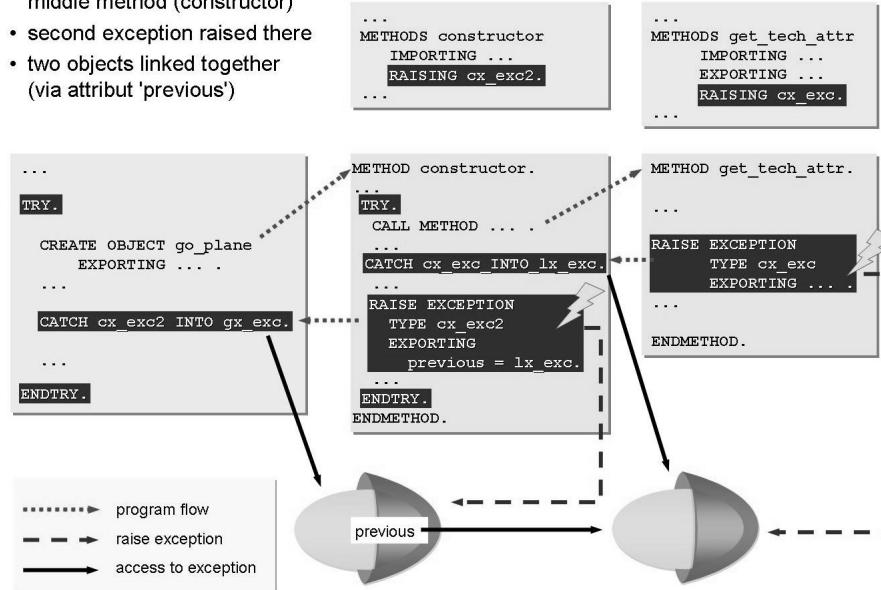


Figure 168: Mapping Exceptions to Each Other

In the example above, the constructor catches the exception raised by the GET_TECH_ATTR method. After some analyzing of the exception it raises a new exception of a different exception class. It chains the existing exception object and the new one by passing a reference to the existing object to the constructor of the new one. As a result, attribute PREVIOUS of the new exception object points to the exception object that preceded it.

The main program catches this second exception and gets access to this second exception object. However, by using the public attribute PREVIOUS, it can navigate to the other exception object and analyze it. In this way, it could access each instance in a chain of exceptions and so follow the history of the raising of the exceptions in the call hierarchy.

Exercise 22: Class-Based Exceptions

Exercise Objectives

After completing this exercise, you will be able to:

- Defining Global Exception Classes
- Raise class-based exceptions
- Propagate class-based exceptions
- Handle class-based exceptions

Business Example

Your program should use the class-based exception concept if an invalid plane type is provided when creating an airplane instance.

Template:

SAPBC401_GCL_S2

Solution:

SAPBC401_EXC_S1 (Main program)

CX_INVALID_PLANETYPE (exception class)

Task 1:

Create a global exception class for an invalid airplane type.

1. Create a global exception class called ZCX_##_INVALID_PLANETYPE (where ## is your two-digit group number).

You want the syntax check to make sure that exceptions based on this class are either handled directly or propagated. Choose the superclass accordingly.

In the exception class, you want to use texts from a message class rather than freely defined texts.



Hint: Whether messages from a message class are used or not must be stated right during creation of the exception class by selecting the *Message Class* checkbox.

Continued on next page

2. Add an attribute for the airplane type (suggested name: PLANETYPE) and assign the type S_PLANETYPE to it. Make sure a corresponding parameter is defined in the signature of constructor.



Hint: Note that you cannot define the constructor of an exception class directly. If you define the attribute as public, a corresponding import parameter is added to the constructor by the framework.

3. Create a default error message. The message text should include a placeholder so that it can be enhanced dynamically to include the airplane type.



Hint: You can define your own message class and message or use message 020 from message class BC401.

Task 2:

Raise your new exception in method GET_TECHNICAL_ATTRIBUTES of your local class LCL_AIRPLANE.

1. Complete your ZBC401_##_MAIN program or copy the template program.
2. Edit the definition of local class LCL_AIRPLANE. Remove the classical, non-class-based exception WRONG_PLANETYPE from the signature of method GET_TECHNICAL_ATTRIBUTES and replace it with your new class-based exception ZCX_##_INVALID_PLANETYPE.
3. Edit the implementation of method GET_TECHNICAL_ATTRIBUTES and raise the class-based exception instead of the classical exception.

Make sure the wrong plane type value is handed over to the constructor and stored inside the exception object.

Task 3:

Explicitly propagate the class-based exception through the entire call hierarchy so that you must only handle it once in the main program.

1. Edit the definition of LCL_AIRPLANE. Remove the classical, non-class-based exception WRONG_PLANETYPE from the signature of method GET_TECHNICAL_ATTRIBUTES and replace it with your new class-based exception ZCX_##_INVALID_PLANETYPE.
2. Edit the implementation of the constructor of local class LCL_AIRPLANE. In the call of method GET_TECHNICAL_ATTRIBUTES, remove the handling of the classical exception WRONG_PLANETYPE

Continued on next page

3. Why is it not necessary to handle the class-based exception ZCX_##_INVALID_PLANETYPE here?

4. Edit the definition of local class LCL_PASSENGER_PLANE. Remove the classical, non-class-based exception WRONG_PLANETYPE from the signature of method CONSTRUCTOR and replace it with your new class-based exception ZCX_##_INVALID_PLANETYPE.

Edit the implementation of the constructor of local class LCL_PASSENGER_PLANE. In the call of SUPER->CONSTRUCTOR, remove the handling of the classical exception WRONG_PLANETYPE

5. Repeat the previous step with the local class LCL_CARGO_PLANE.

Task 4:

Handle the raised exception in the main program whenever an airplane object is created.

1. Define a global reference variable for this purpose (suggested name: GO_INV_PLANETYPE) and type it with your exception class ZCX_##_INVALID_PLANETYPE.
2. In the CREATE OBJECT statement for airplanes (instances of LCL_AIR-PLANE, LCL_PASSENGER_PLANE and LCL_CARGO_PLANE) remove the handling of the classical, non-class-based exception WRONG_PLANETYPE.
3. Surround each CREATE OBJECT statement for airplanes with a TRY-ENDTRY structure. Add a CATCH block for your exception ZCX_##_INVALID_PLANETYPE. In the case of an exception, let the reference variable GO_INV_PLANETYPE point to the exception object.
4. Read the error text from the exception object and output it to the ABAP list.



Hint: It is not possible to use the result of functional methods directly in the write statement. You need to define a variable to store the text (suggested name: GV_TEXT with data type STRING).

5. Activate and test your program. For test purposes, create an airplane with a wrong plane type and debug the propagation of the exception.

Solution 22: Class-Based Exceptions

Task 1:

Create a global exception class for an invalid airplane type.

1. Create a global exception class called ZCX_##_INVALID_PLANETYPE (where ## is your two-digit group number).

You want the syntax check to make sure that exceptions based on this class are either handled directly or propagated. Choose the superclass accordingly.

In the exception class, you want to use texts from a message class rather than freely defined texts.



Hint: Whether messages from a message class are used or not must be stated right during creation of the exception class by selecting the *Message Class* checkbox.

- a) Create a global class.
b) On the following popup, choose the superclass CX_STATIC_CHECK.
c) Choose the *Exception Class* option and select the *Message Class* checkbox.
d) Speak to your instructor if you have any questions.
2. Add an attribute for the airplane type (suggested name: PLANETYPE) and assign the type S_PLANETYPE to it. Make sure a corresponding parameter is defined in the signature of constructor.



Hint: Note that you cannot define the constructor of an exception class directly. If you define the attribute as public, a corresponding import parameter is added to the constructor by the framework.

- a) Go to the *Attributes* tab. Enter the attribute name and type and choose the visibility.
b) Speak to your instructor if you have any questions.

Continued on next page

3. Create a default error message. The message text should include a placeholder so that it can be enhanced dynamically to include the airplane type.



Hint: You can define your own message class and message or use message 020 from message class BC401.

- a) Go to the *Texts* tab. Select the default text already there and choose *Message Text*.
- b) Enter message class and message number in the relevant fields. In the field *Attr. 1*, use the value help to choose attribute PLANETYPE.
- c) Speak to your instructor if you have any questions.

Task 2:

Raise your new exception in method GET_TECHNICAL_ATTRIBUTES of your local class LCL_AIRPLANE.

1. Complete your ZBC401_##_MAIN program or copy the template program.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
2. Edit the definition of local class LCL_AIRPLANE. Remove the classical, non-class-based exception WRONG_PLANETYPE from the signature of method GET_TECHNICAL_ATTRIBUTES and replace it with your new class-based exception ZCX_##_INVALID_PLANETYPE.
 - a) See the source code extract from the model solution.
3. Edit the implementation of method GET_TECHNICAL_ATTRIBUTES and raise the class-based exception instead of the classical exception.

Make sure the wrong plane type value is handed over to the constructor and stored inside the exception object.

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

Continued on next page

Task 3:

Explicitly propagate the class-based exception through the entire call hierarchy so that you must only handle it once in the main program.

1. Edit the definition of LCL_AIRPLANE. Remove the classical, non-class-based exception WRONG_PLANETYPE from the signature of method GET_TECHNICAL_ATTRIBUTES and replace it with your new class-based exception ZCX_##_INVALID_PLANETYPE.
 - a) See the source code extract from the model solution.
2. Edit the implementation of the constructor of local class LCL_AIRPLANE. In the call of method GET_TECHNICAL_ATTRIBUTES, remove the handling of the classical exception WRONG_PLANETYPE
 - a) See the source code extract from the model solution.
3. Why is it not necessary to handle the class-based exception ZCX_##_INVALID_PLANETYPE here?

Answer: Because this exception is propagated in the signature of the constructor of LCL_AIRPLANE.
4. Edit the definition of local class LCL_PASSENGER_PLANE. Remove the classical, non-class-based exception WRONG_PLANETYPE from the signature of method CONSTRUCTOR and replace it with your new class-based exception ZCX_##_INVALID_PLANETYPE.

Edit the implementation of the constructor of local class LCL_PASSENGER_PLANE. In the call of SUPER->CONSTRUCTOR, remove the handling of the classical exception WRONG_PLANETYPE

 - a) See the source code extract from the model solution.
5. Repeat the previous step with the local class LCL_CARGO_PLANE.
 - a) See the source code extract from the model solution.

Task 4:

Handle the raised exception in the main program whenever an airplane object is created.

1. Define a global reference variable for this purpose (suggested name: GO_INV_PLANETYPE) and type it with your exception class ZCX_##_INVALID_PLANETYPE.
 - a) See the source code extract from the model solution.

Continued on next page

2. In the CREATE OBJECT statement for airplanes (instances of LCL_AIR-PLANE, LCL_PASSENGER_PLANE and LCL_CARGO_PLANE) remove the handling of the classical, non-class-based exception WRONG_PLANETYPE.
 - a) See the source code extract from the model solution.
3. Surround each CREATE OBJECT statement for airplanes with a TRY-ENDTRY structure. Add a CATCH block for your exception ZCX_##_INVALID_PLANETYPE. In the case of an exception, let the reference variable GO_INV_PLANETYPE point to the exception object.
 - a) See the source code extract from the model solution.
4. Read the error text from the exception object and output it to the ABAP list.



Hint: It is not possible to use the result of functional methods directly in the write statement. You need to define a variable to store the text (suggested name: GV_TEXT with data type STRING).

- a) See the source code extract from the model solution.
5. Activate and test your program. For test purposes, create an airplane with a wrong plane type and debug the propagation of the exception.
 - a) Perform this step in the usual manner.

Result

Include BC401_EXC_S1_CARRIER

```
-----*
*      CLASS lcl_airplane DEFINITION
*-----*
CLASS lcl_airplane DEFINITION.

PUBLIC SECTION.

METHODS:
  constructor
  IMPORTING
    iv_name      TYPE string
    iv_planetype TYPE saplane-planetype
*   EXCEPTIONS
*     wrong_planetype,
  RAISING
```

Continued on next page

```

cx_invalid_planetype,
display_attributes.

CLASS-METHODS:
display_n_o_airplanes,
class_constructor.

EVENTS:
airplane_created.

PROTECTED SECTION.

CONSTANTS:
c_pos_1 TYPE i VALUE 30.

PRIVATE SECTION.

TYPES:
ty_planetypes TYPE STANDARD TABLE OF saplane
WITH NON-UNIQUE KEY planetype.

DATA:
mv_name      TYPE string,
mv_planetype TYPE saplane-planetype,
mv_weight    TYPE saplane-weight,
mv_tankcap   TYPE saplane-tankcap.

CLASS-DATA:
gv_n_o_airplanes TYPE i,
gt_planetypes   TYPE ty_planetypes.

CLASS-METHODS:
get_technical_attributes
IMPORTING
iv_type      TYPE saplane-planetype
EXPORTING
ev_weight    TYPE saplane-weight
ev_tankcap   TYPE saplane-tankcap
*
EXCEPTIONS
*
wrong_planetype.
RAISING cx_invalid_planetype.

ENDCLASS.          "lcl_airplane DEFINITION

```

Continued on next page

```

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

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

METHOD constructor.
  mv_name      = iv_name.
  mv_planetype = iv_planetype.

  get_technical_attributes(
    EXPORTING
      iv_type = iv_planetype
    IMPORTING
      ev_weight  = mv_weight
      ev_tankcap = mv_tankcap ).

  gv_n_o_airplanes = gv_n_o_airplanes + 1.
  RAISE EVENT airplane_created.

*  get_technical_attributes(
*    EXPORTING
*      iv_type = iv_planetype
*    IMPORTING
*      ev_weight  = mv_weight
*      ev_tankcap = mv_tankcap
*    EXCEPTIONS
*      wrong_planetype = 1 ).
*
*  IF sy-subrc <> 0.
*    RAISE wrong_planetype.
*  ELSE.
*    gv_n_o_airplanes = gv_n_o_airplanes + 1.
*    RAISE EVENT airplane_created.
*  ENDIF.

ENDMETHOD.          "constructor

METHOD display_attributes.

  WRITE:
    / icon_ws_plane AS ICON,

```

Continued on next page

```

        / 'Name of Airplane'(001) , AT c_pos_1 mv_name,
        / 'Type of Airplane:'(002), AT c_pos_1 mv_planetype,
        / 'Weight:'(003),           AT c_pos_1 mv_weight LEFT-JUSTIFIED,
        / 'Tank capacity:'(004),    AT c_pos_1 mv_tankcap LEFT-JUSTIFIED.

ENDMETHOD.          "display_attributes

METHOD display_n_o_airplanes.
SKIP.
WRITE:
/ 'Number of airplanes:'(ca1),
AT c_pos_1 gv_n_o_airplanes LEFT-JUSTIFIED.
ENDMETHOD.          "display_n_o_airplanes

METHOD get_technical_attributes.
DATA: ls_planetype TYPE saplane.

READ TABLE gt_planetypes INTO ls_planetype
WITH TABLE KEY planetype = iv_type
TRANSPORTING weight tankcap.

IF sy-subrc = 0.
ev_weight = ls_planetype-weight.
ev_tankcap = ls_planetype-tankcap.
ELSE.
*      RAISE wrong_planetype.
RAISE EXCEPTION TYPE cx_invalid_planetype
EXPORTING
planetype = iv_type.
ENDIF.
ENDMETHOD.          "get_technical_attributes

ENDCLASS.          "lcl_airplane IMPLEMENTATION

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

PUBLIC SECTION.

METHODS:
constructor

```

Continued on next page

```

IMPORTING
  iv_name      TYPE string
  iv_planetype TYPE saplane-planetype
  iv_cargo     TYPE s_plan_car
RAISING
  cx_invalid_planetype,
*  EXCEPTIONS
*  wrong_planetype,
display_attributes REDEFINITION.

PRIVATE SECTION.

DATA:
  mv_cargo TYPE s_plan_car.

ENDCLASS.          "lcl_cargo_plane DEFINITION

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

METHOD constructor.
  super->constructor(
    EXPORTING
      iv_name      = iv_name
      iv_planetype = iv_planetype ).
*  EXCEPTIONS
*  wrong_planetype = 1 .
*  IF sy-subrc <> 0.
*  RAISE wrong_planetype.
*  ENDIF.
  mv_cargo = iv_cargo.
ENDMETHOD.          "constructor

METHOD display_attributes.
  super->display_attributes( ).
  WRITE:
    / 'Max Cargo:'(005), AT c_pos_1 mv_cargo LEFT-JUSTIFIED.
  ULINE.
ENDMETHOD.          "display_attributes

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
        iv_name      TYPE string
        iv_planetype TYPE saplane-planetype
        iv_seats     TYPE s_seatsmax
    EXCEPTIONS
        wrong_planetype
    RAISING
        cx_invalid_planetype,
display_attributes REDEFINITION.

PRIVATE SECTION.

DATA:
    mv_seats TYPE s_seatsmax.
ENDCLASS.                      "lcl_passenger_plane DEFINITION

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

METHOD constructor.
    super->constructor(
        EXPORTING
            iv_name      = iv_name
            iv_planetype = iv_planetype ).

    EXCEPTIONS
        wrong_planetype = 1 .
    IF sy-subrc <> 0.
        RAISE wrong_planetype.
    ENDIF.

    mv_seats = iv_seats.

```

Continued on next page

```

ENDMETHOD.           "constructor

METHOD display_attributes.
  super->display_attributes( ).
  WRITE:
    / 'Max Seats:'(006), AT c_pos_1 mv_seats LEFT-JUSTIFIED.
    ULINE.
ENDMETHOD.          "display_attributes
ENDCLASS.           "lcl_passenger_plane IMPLEMENTATION

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

...
ENDCLASS.           "lcl_carrier DEFINITION

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

...
ENDCLASS.           "lcl_carrier IMPLEMENTATION

```

Main program SAPBC401_EXC_S1

```

REPORT  sapbc401_exc_s1.

TYPE-POOLS icon.

INCLUDE bc401_exc_s1_agency.
INCLUDE bc401_exc_s1_carrier.
INCLUDE bc401_exc_s1_rental.

DATA:
  go_hotel    TYPE REF TO cl_hotel2,
  go_vehicle  TYPE REF TO lcl_vehicle,
  go_truck    TYPE REF TO lcl_truck,
  go_bus      TYPE REF TO lcl_bus,

```

Continued on next page

```

go_rental      TYPE REF TO lcl_rental,
go_passenger   TYPE REF TO lcl_passenger_plane,
go_cargo       TYPE REF TO lcl_cargo_plane,
go_carrier     TYPE REF TO lcl_carrier,
go_agency      TYPE REF TO lcl_travel_agency.

DATA:
gv_text          TYPE string,
gx_inv_planetype TYPE REF TO cx_invalid_planetype.

START-OF-SELECTION.
***** create travel_agency *****
CREATE OBJECT go_agency
EXPORTING
iv_name = 'Travel&Smile Travel'.

***** create hotel *****
CREATE OBJECT go_hotel
EXPORTING
iv_name = 'Sleep Well Hotel'
iv_beds = 345.

***** create rental *****
CREATE OBJECT go_rental
EXPORTING
iv_name = 'Happy Car Rental'.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
iv_make = 'MAN'
iv_cargo = 45.

***** create truck *****
CREATE OBJECT go_bus
EXPORTING
iv_make      = 'Mercedes'
iv_passengers = 80.

***** create truck *****
CREATE OBJECT go_truck

```

Continued on next page

```

        EXPORTING
          iv_make  = 'VOLVO'
          iv_cargo = 48.

***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly'.

***** Passenger Plane *****
TRY.
  CREATE OBJECT go_passenger
  EXPORTING
    iv_name      = 'LH BERLIN'
    iv_planetype = '747-500'
    iv_seats     = 345.

  CATCH cx_invalid_planetype INTO gx_inv_planetype.
  gv_text = gx_inv_planetype->get_text( ).

  WRITE:
    / icon_failure AS ICON,
    gv_text.
ENDTRY.

* CREATE OBJECT go_passenger
* EXPORTING
*   iv_name      = 'LH BERLIN'
*   iv_planetype = '747-400'
*   iv_seats     = 345
* EXCEPTIONS
*   wrong_planetype = 1.
* IF sy-subrc <> 0.
*   WRITE:
*     / icon_failure AS ICON,
*     'Wrong plane type'.
* ENDIF.

***** cargo Plane *****
TRY.
  CREATE OBJECT go_cargo
  EXPORTING
    iv_name      = 'US Hercules'
    iv_planetype = '747-200F'
    iv_cargo     = 533.

```

Continued on next page

```
CATCH cx_invalid_planetype INTO gx_inv_planetype.  
    gv_text = gx_inv_planetype->get_text( ).  
  
    WRITE:  
        / icon_failure AS ICON,  
        gv_text.  
    ENDTRY.  
  
*  CREATE OBJECT go_cargo  
*  EXPORTING  
*      iv_name          = 'US Hercules'  
*      iv_planetype     = '747-200F'  
*      iv_cargo         = 533  
*  EXCEPTIONS  
*      wrong_planetype = 1.  
*  
*  IF sy-subrc <> 0.  
*  WRITE:  
*      / icon_failure AS ICON,  
*      'Wrong plane type'.  
*  ENDIF.  
  
***** show attributes of all partners of travel_agency *****  
go_agency->display_attributes( ).
```

Exercise 23: (Optional) Mapping Exceptions to Each Other

Exercise Objectives

After completing this exercise, you will be able to:

- Map an exception, when it is raised, to another exception that has already been raised
- When handling a mapped exception, evaluate the previous exception as well

Business Example

Your program should use the class-based exception concept if an invalid plane type is provided when creating an airplane instance. Instead of just propagating the exception raised by the GET_TECHNICAL_ATTRIBUTES, the constructor should handle it and raise a different exception. The two exceptions should be chained together so that the main program can evaluate both.

Template:

SAPBC401_EXC_S1

Solution:

SAPBC401_EXC_S2

Task 1:

In the constructor of your local class LCL_AIRPLANE, handle the exception raised by method GET_TECHNICAL_ATTRIBUTES.

1. Complete your ZBC401_##_MAIN program or copy the template program.
2. Edit the implementation of the constructor of local class LCL_AIRPLANE. Define a local object reference (suggested name: LX_INV_PLANETYPE) and type it in a way that it can point to the exception object raised by method GET_TECHNICAL_ATTRIBUTES.
3. Implement a TRY-ENDTRY structure for exception handling. Get hold of the exception object but leave the CATCH block empty for now.

Continued on next page

Task 2:

Raise a new exception and map it to the original one. Make sure the new exception is propagated all the way to the main program.

1. In the constructor of LCL_AIRPLANE, raise a new exception of type CX_INVALID_VALUE when the exception raised by GET_TECHNICAL_ATTRIBUTES is being handled.



Hint: Check the signature of exception class CX_INVALID_VALUE, to see if it contains any attributes for which you need to provide values.

Transfer the reference to the original exception instance to the exception instance that is to be created.

2. Adjust the signature of the constructor so that your new exception is propagated.
3. Which other method signatures do you have to adjust to propagate the exception up to the main program.

Task 3:

In the main program, handle your new exception and evaluate the information of both, the new and the original exception instance.

1. Catch your new exception in the main program wherever you create an instance of an airplane. Define a local reference variable for the purpose with a suitable type (suggested name: GX_INV_VALUE).
2. Use this reference to extract the text of your new exception. Output the text on the list.
3. Use the same variables to export the text of the original exception.
Output the text on the list

Solution 23: (Optional) Mapping Exceptions to Each Other

Task 1:

In the constructor of your local class LCL_AIRPLANE, handle the exception raised by method GET_TECHNICAL_ATTRIBUTES.

1. Complete your ZBC401_##_MAIN program or copy the template program.
 - a) Carry out this step in the usual manner. Additional information is available in the SAP Library.
2. Edit the implementation of the constructor of local class LCL_AIRPLANE. Define a local object reference (suggested name: LX_INV_PLANETYPE) and type it in a way that it can point to the exception object raised by method GET_TECHNICAL_ATTRIBUTES.
 - a) See the source code extract from the model solution.
3. Implement a TRY-ENDTRY structure for exception handling. Get hold of the exception object but leave the CATCH block empty for now.
 - a) See the source code extract from the model solution.

Task 2:

Raise a new exception and map it to the original one. Make sure the new exception is propagated all the way to the main program.

1. In the constructor of LCL_AIRPLANE, raise a new exception of type CX_INVALID_VALUE when the exception raised by GET_TECHNICAL_ATTRIBUTES is being handled.



Hint: Check the signature of exception class CX_INVALID_VALUE, to see if it contains any attributes for which you need to provide values.

- Transfer the reference to the original exception instance to the exception instance that is to be created.
- a) See the source code extract from the model solution.
 2. Adjust the signature of the constructor so that your new exception is propagated.
 - a) See the source code extract from the model solution.

Continued on next page

3. Which other method signatures do you have to adjust to propagate the exception up to the main program.

Answer: The constructors of LCL_CARGO_PLANE and LCL_PASSENGER_PLANE.

Task 3:

In the main program, handle your new exception and evaluate the information of both, the new and the original exception instance.

1. Catch your new exception in the main program wherever you create an instance of an airplane. Define a local reference variable for the purpose with a suitable type (suggested name: GX_INV_VALUE).
 - a) See the source code extract from the model solution.
2. Use this reference to extract the text of your new exception. Output the text on the list.
 - a) See the source code extract from the model solution.
3. Use the same variables to export the text of the original exception. Output the text on the list
 - a) See the source code extract from the model solution.

Result

Include BC401_EXC_S2_CARRIER

```
*-----*
*      CLASS lcl_airplane DEFINITION
*-----*
CLASS lcl_airplane DEFINITION.

PUBLIC SECTION.

METHODS:
  constructor
  IMPORTING
    iv_name      TYPE string
    iv_planetype TYPE saplane-planetype
  RAISING
    cx_invalid_value,
*    cx_invalid_planetype,
```

Continued on next page

```

        display_attributes.

CLASS-METHODS:
    display_n_o_airplanes,
    class_constructor.

EVENTS:
    airplane_created.

PROTECTED SECTION.

CONSTANTS:
    c_pos_1 TYPE i VALUE 30.

PRIVATE SECTION.

TYPES:
    ty_planetypes TYPE STANDARD TABLE OF saplane
        WITH NON-UNIQUE KEY planetype.

DATA:
    mv_name      TYPE string,
    mv_planetype TYPE saplane-planetype,
    mv_weight    TYPE saplane-weight,
    mv_tankcap   TYPE saplane-tankcap.

CLASS-DATA:
    gv_n_o_airplanes TYPE i,
    gt_planetypes   TYPE ty_planetypes.

CLASS-METHODS:
    get_technical_attributes
        IMPORTING
            iv_type      TYPE saplane-planetype
        EXPORTING
            ev_weight   TYPE saplane-weight
            ev_tankcap  TYPE saplane-tankcap
        RAISING
            cx_invalid_planetype.

ENDCLASS.                                     "lcl_airplane DEFINITION

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

```

Continued on next page

```

CLASS lcl_airplane IMPLEMENTATION.

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

METHOD constructor.
  DATA:
    lx_inv_planetype TYPE REF TO cx_invalid_planetype.

    mv_name          = iv_name.
    mv_planetype     = iv_planetype.

TRY.
  get_technical_attributes(
    EXPORTING
      iv_type = iv_planetype
    IMPORTING
      ev_weight  = mv_weight
      ev_tankcap = mv_tankcap ).

  gv_n_o_airplanes = gv_n_o_airplanes + 1.
  RAISE EVENT airplane_created.

CATCH cx_invalid_planetype INTO lx_inv_planetype.
  RAISE EXCEPTION TYPE cx_invalid_value
    EXPORTING
      parnam   = 'IV_PLANETYPE'
      previous = lx_inv_planetype.

ENDTRY.

ENDMETHOD.                                     "constructor

METHOD display_attributes.

WRITE:
  / icon_ws_plane AS ICON,
  / 'Name of Airplane'(001), AT c_pos_1 mv_name,
  / 'Type of Airplane:'(002), AT c_pos_1 mv_planetype,
  / 'Weight:'(003),           AT c_pos_1 mv_weight LEFT-JUSTIFIED,
  / 'Tank capacity:'(004),    AT c_pos_1 mv_tankcap LEFT-JUSTIFIED.

ENDMETHOD.                                     "display_attributes

```

Continued on next page

```

METHOD display_n_o_airplanes.
SKIP.
WRITE:
/ 'Number of airplanes:'(cal),
AT c_pos_1 gv_n_o_airplanes LEFT-JUSTIFIED.
ENDMETHOD.           "display_n_o_airplanes

METHOD get_technical_attributes.
DATA: ls_planetype TYPE saplane.

READ TABLE gt_planetypes INTO ls_planetype
WITH TABLE KEY planetype = iv_type
TRANSPORTING weight tankcap.

IF sy-subrc = 0.
  ev_weight = ls_planetype-weight.
  ev_tankcap = ls_planetype-tankcap.
ELSE.
*   RAISE wrong_planetype.
  RAISE EXCEPTION TYPE cx_invalid_planetype
  EXPORTING
    planetype = iv_type.
ENDIF.
ENDMETHOD.           "get_technical_attributes

ENDCLASS.           "lcl_airplane IMPLEMENTATION

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

PUBLIC SECTION.

METHODS:
constructor
IMPORTING
  iv_name      TYPE string
  iv_planetype TYPE saplane-planetype
  iv_cargo     TYPE s_plan_car
RAISING
  cx_invalid_value,

```

Continued on next page

```

*          cx_invalid_planetype,
display_attributes REDEFINITION.

PRIVATE SECTION.

DATA:
mv_cargo TYPE s_plan_car.

ENDCLASS.           "lcl_cargo_plane DEFINITION

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

...

ENDCLASS.           "lcl_cargo_plane IMPLEMENTATION

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

PUBLIC SECTION.

METHODS:
constructor
IMPORTING
iv_name      TYPE string
iv_planetype TYPE saplane-planetype
iv_seats     TYPE s_seatsmax
RAISING:
*          cx_invalid_value,
*          cx_invalid_planetype,
display_attributes REDEFINITION.

PRIVATE SECTION.

DATA:
mv_seats TYPE s_seatsmax.
ENDCLASS.           "lcl_passenger_plane DEFINITION

```

Continued on next page

```

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

...
ENDCLASS.           "lcl_passenger_plane IMPLEMENTATION

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

...
ENDCLASS.           "lcl_carrier DEFINITION

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

...
ENDCLASS.           "lcl_carrier IMPLEMENTATION

```

Main Program SAPBC401_EXC_S2

```

REPORT  sapbc401_exc_s2.

TYPE-POOLS icon.

INCLUDE bc401_exc_s2_agency.
INCLUDE bc401_exc_s2_carrier.
INCLUDE bc401_exc_s2_rental.

DATA:
  go_hotel      TYPE REF TO cl_hotel2,
  go_vehicle    TYPE REF TO lcl_vehicle,
  go_truck      TYPE REF TO lcl_truck,
  go_bus        TYPE REF TO lcl_bus,
  go_rental     TYPE REF TO lcl_rental,
  go_passenger  TYPE REF TO lcl_passenger_plane,
  go_cargo      TYPE REF TO lcl_cargo_plane,

```

Continued on next page

```

go_carrier      TYPE REF TO lcl_carrier,
go_agency       TYPE REF TO lcl_travel_agency.

DATA:
  gv_text          TYPE string,
  gx_inv_value    TYPE REF TO cx_invalid_value.
*  gx_inv_planetype TYPE REF TO cx_invalid_planetype.

START-OF-SELECTION.
***** create travel_agency *****
CREATE OBJECT go_agency
EXPORTING
  iv_name = 'Travel&Smile Travel'.

***** create hotel *****
CREATE OBJECT go_hotel
EXPORTING
  iv_name = 'Sleep Well Hotel'
  iv_beds = 345.

***** create rental *****
CREATE OBJECT go_rental
EXPORTING
  iv_name = 'Happy Car Rental'.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make  = 'MAN'
  iv_cargo = 45.

***** create truck *****
CREATE OBJECT go_bus
EXPORTING
  iv_make      = 'Mercedes'
  iv_passengers = 80.

***** create truck *****
CREATE OBJECT go_truck
EXPORTING
  iv_make  = 'VOLVO'
  iv_cargo = 48.

```

Continued on next page

```

***** Create Carrier *****
CREATE OBJECT go_carrier
EXPORTING
  iv_name = 'Smile&Fly'.

***** Passenger Plane *****
TRY.
  CREATE OBJECT go_passenger
  EXPORTING
    iv_name      = 'LH BERLIN'
    iv_planetype = '747-500'
    iv_seats     = 345.

  CATCH cx_invalid_value INTO gx_inv_value.
    gv_text = gx_inv_value->get_text( ).
    WRITE:
      / icon_failure AS ICON,
        gv_text.

    gv_text = gx_inv_value->previous->get_text( ).
    WRITE:
      / icon_failure AS ICON,
        gv_text.

*   CATCH cx_invalid_planetype INTO gx_inv_planetype.
*   gv_text = gx_inv_planetype->get_text( ).

*   WRITE:
*     / icon_failure AS ICON,
*       gv_text.

ENDTRY.

***** cargo Plane *****
TRY.
  CREATE OBJECT go_cargo
  EXPORTING
    iv_name      = 'US Hercules'
    iv_planetype = '747-200F'
    iv_cargo     = 533.

  CATCH cx_invalid_value INTO gx_inv_value.
    gv_text = gx_inv_value->get_text( ).
    WRITE:

```

Continued on next page

```
/ icon_failure AS ICON,  
  gv_text.  
  
  gv_text = gx_inv_value->previous->get_text( ).  
  WRITE:  
    / icon_failure AS ICON,  
      gv_text.  
  
*   CATCH cx_invalid_planetype INTO gx_inv_planetype.  
*   gv_text = gx_inv_planetype->get_text( ).  
  
*   WRITE:  
*     / icon_failure AS ICON,  
*       gv_text.  
  
ENDTRY.  
  
***** show attributes of all partners of travel_agency *****  
go_agency->display_attributes( ).
```



Lesson Summary

You should now be able to:

- Create exception classes
- Raise class-based exceptions in ABAP Objects programs
- Propagate class-based exceptions in ABAP Objects programs
- Handle class-based exceptions in ABAP Objects programs
- Map class-based exceptions in ABAP Objects programs to each other

Related Information

For more information, refer to the SAP Library and to the key word documentation for the individual statements.

Lesson: RTTS (Run Time Type Services)

Lesson Overview

As of SAP Web AS 6.10, ABAP Objects enables you to **query** type attributes at runtime using standard SAP classes. This concept (**RTTI** = Run Time Type Identification) has now been enhanced for SAP Web AS 6.40 to enable you to **create** (**RTTC** = Run Time Type Creation) types dynamically at runtime. 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

An airline wants to identify all its cargo planes.

RTTI (Run Time Type Identification)/ Type Analysis at Runtime

Since the introduction of ABAP Objects, there is now a class-based concept called Run Time Type Information (RTTI), which you can use to determine type attributes at runtime. This concept includes all ABAP types, and so covers all of the 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:

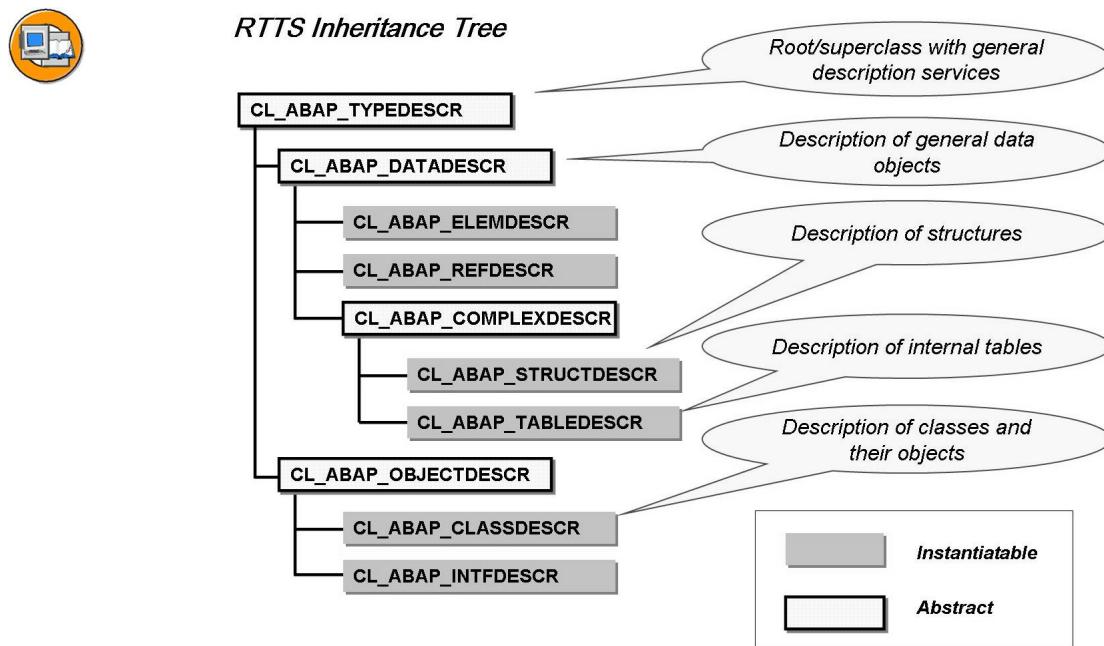


Figure 169: Dynamic Type Analysis with RTTI Classes

The class hierarchy of the description classes corresponds to the hierarchy of the types in ABAP Objects. 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, exactly one description object exists for each type. The attributes of the description object contain information on the attributes of the type.

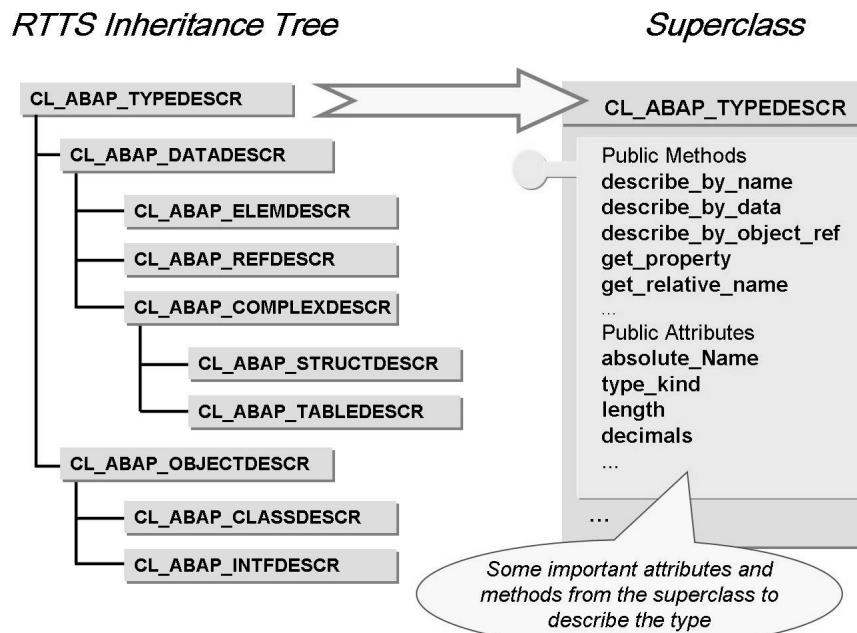


Figure 170: RTTI – Methods and Attributes of the Root Class

Dynamic Data Type Analysis – Application Example

In the following application example, the attributes of a structure are to be identified using RTTI. The subclass CL_ABAP_STRUCTDESCR is used for this purpose.

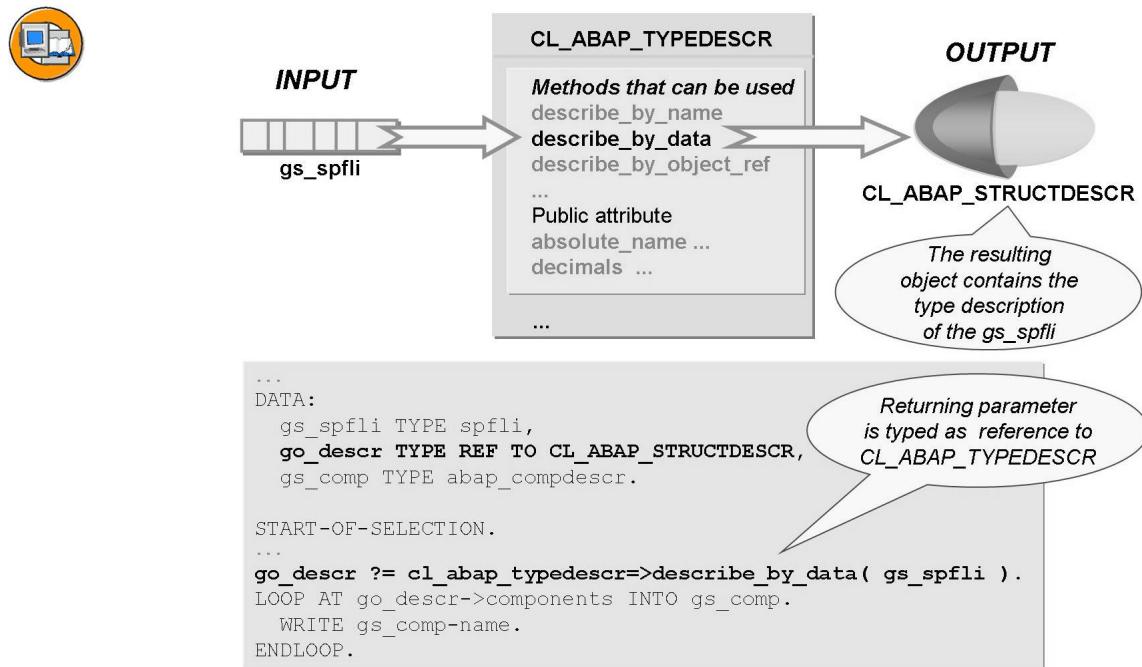


Figure 171: RTTI – Querying Structure Attributes

Since we need the attributes of a structure, we first define a reference to the appropriate description class. Instances of this class have a COMPONENTS attribute that you use to describe the individual components of the relevant structure. This attribute is an internal table. Therefore, you also need to define a work area with a compatible line type.

The functional method call provides the reference to the description instance for the structure to be transferred (the structure WA could also have been generated dynamically, but in this example, it is static as usual).

The abstract class CL_ABAP_TYPEDESCR contains the static method DESCRIBE_BY_DATA. Its RETURNING parameter is typed as a reference to this superclass. However, since the actual parameter GO_DESCR is typed on the subclass CL_ABAP_STRUCTDESCR, the object needs to be assigned using a down-cast.

You can then access the attributes of the description instance in any form. In this example, the component names are displayed as the column headers. (For the sake of clarity, we have omitted the formatting options.)

Dynamic Object Type Analysis – Application Example

In the following example, the attributes of an object are to be analyzed. As an example, we demonstrate the reference SENDER of the handler method IN_VEHICLE_CREATED for the class LCL_RENTAL.

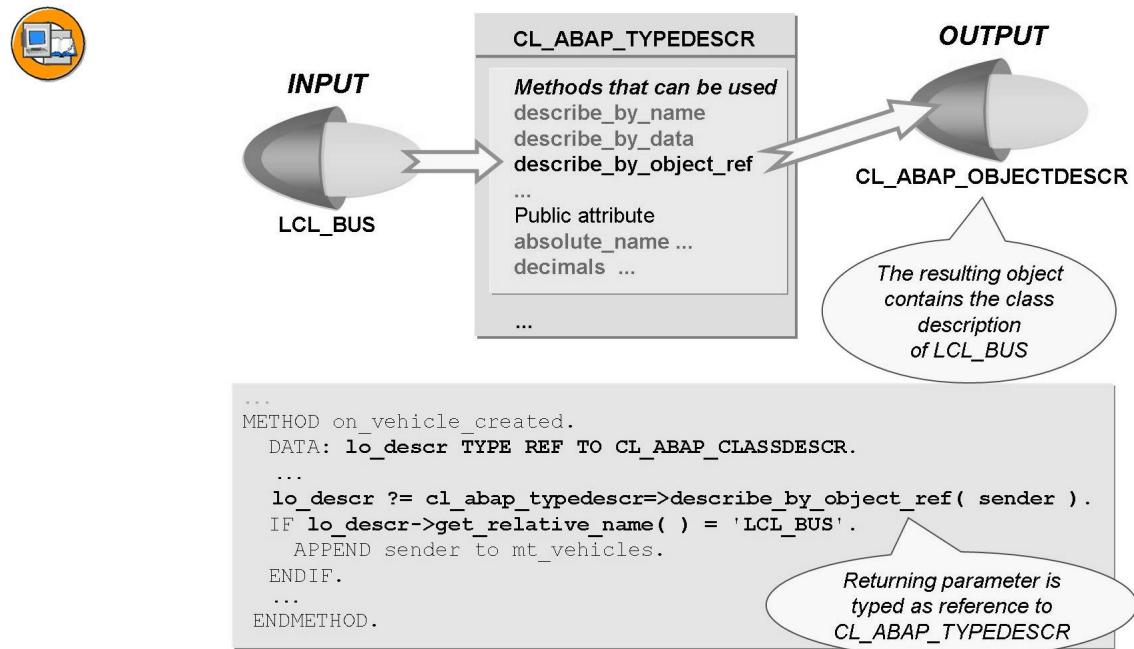


Figure 172: RTTI - Querying Object-Type Attributes

As an example, we enhance the existing application with the travel agency and its business partner. In that simplified example, we previously specified that an instance of the LCL_RENTAL vehicle rental class reacts to an event by including the vehicle instance that triggered the event in the separate mt_vehicles list. So far, the triggering instances could be buses (LCL_BUS) or trucks (LCL_TRUCK).

We will now extend this example: The vehicle rental company is now only interested in buses.

The reference to the triggering vehicle instance is contained in the SENDER parameter of the event handler method. Its dynamic object type is to be analyzed to determine whether the vehicle in question is a bus or a truck.

A functional RTTI method call returns the reference to the description instance of the transferred vehicle instance.

The abstract class CL_ABAP_TYPEDESCR has this DESCRIBE_BY_OBJECT_REF method. Its returning parameter is typed as a reference to this superclass. However, since the actual parameter R_DESCR is typed on the subclass CL_ABAP_CLASSDESCR, the object needs to be assigned using a down-cast.

You can then access the attributes of the description instance in any way. The functional method GET_RELATIVE_NAME supplies the class name.

 **Note:** You could also fulfill the same purpose without using RTTI classes. For example, you could use a down cast assignment from SENDER to a reference variable that has a LCL_BUS static type. This would result in a runtime error for vehicle instances that are **not** buses. The system could then catch this runtime error. Therefore, a runtime error not being triggered would be the criterion for including a vehicle into the vehicle list.

RTTC (Run Time Type Creation)/ Type Creation at Runtime

For SAP Web AS 6.40, RTTC was added to the RTTI type classes to facilitate the creation of types at runtime.

The properties of the types are implemented by type-object attributes. This means that each type has a type object, whose attributes describe the properties of the type.



RTTC - Run Time Type Creation

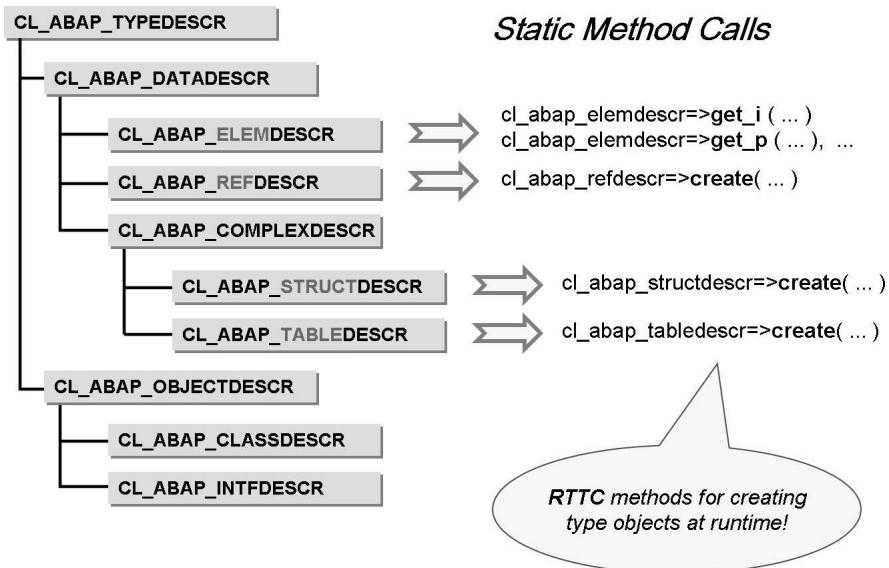
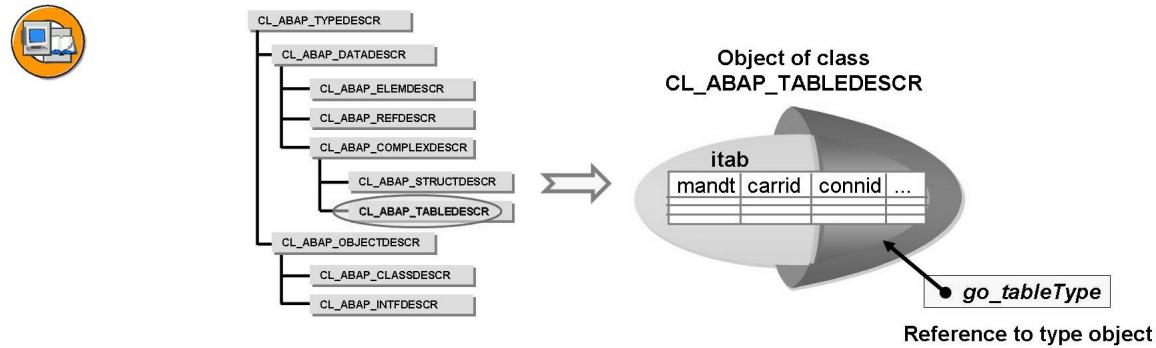


Figure 173: RTTC - Dynamic Type Creation

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 the type class methods. To obtain references to type objects, you can call specific static methods for the class CL_ABAP_TYPEDESCR or methods of the special type classes (GET_I, GET_C, ..., CREATE).



A type is defined completely by its type object

- Every type has a runtime type object (RTTS instance)
- The runtime type object describes the data type completely
- A type object is local to the program, transient, and anonymous (nameless)
- A type object cannot be deleted or changed

Figure 174: RTTC - Type Objects for the Type Description

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.

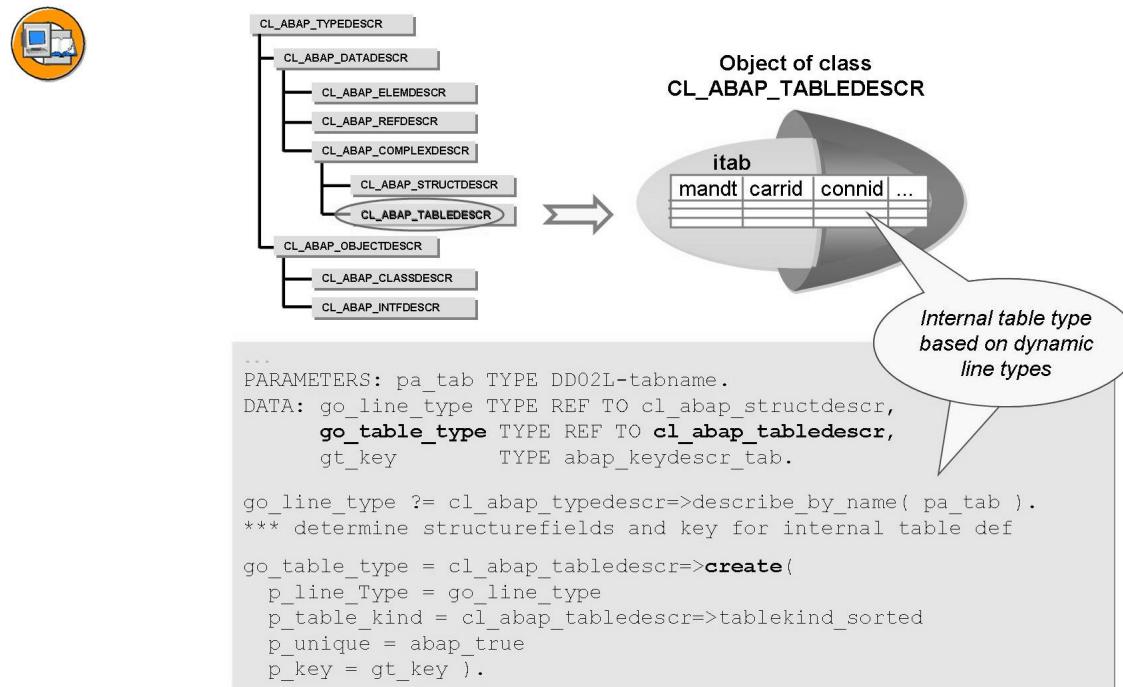


Figure 175: RTTC - Creating a Table Type

With the method **CREATE**, the class CL_ABAP_TABLEDESCR enables you to create a local table type at runtime. The parameter P_LINE_TYPE is an important formal parameter of this static method. This parameter describes the line type using a reference to the CL_ABAP_STRUCTDESCR. In the example provided here, SPFLI is used as the line type. The method DESCRIBE_BY_NAME instantiates a suitable object for this (the ABAP_TRUE parameter from the sample program originates from a type pool that is used internally in the RTTC classes).

The question that now arises is what you should technically do with the table type that has just been created. For instance, you could use it as an example of creating an internal table at runtime:

```
...
DATA gt_itab TYPE REF TO DATA,
      go_table_Type TYPE REF TO cl_abap_tabledescr.
...
* Creation of internal tabletype with static method
* CREATE of RTTS-class cl_abap_tabledescr
...
CREATE DATA gt_itab TYPE HANDLE go_table_Type.
```

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**. The type object may have been created on the existing data objects using the RTTS methods or by dynamically defining a new data type.

Exercise 24: RTTI - Querying Object Attributes

Exercise Objectives

After completing this exercise, you will be able to:

- Query attributes of objects at runtime using RTTI

Business Example

In the program in question for the travel agency model, the airline (LCL_CARRIER) wants to determine the highest cargo value of its cargo airplanes.

Template:

SAPBC401_RTT_T1

Solution:

SAPBC401_RTT_S1

Task:

Determine the highest cargo value for all cargo airplanes entered in your application. The cargo airplanes are entered in the class LCL_CARRIER (of the airline).

1. If you have completed the optional exercise for a down cast (determine the highest cargo for an airplane), you can complete your program ZBC401_##_MAIN (where ## is your two-digit group number). Otherwise, copy the template program with all its includes (suggested name for the new program: **ZBC401_##_RTTI_1**).
2. RTTI should now solve the method GET_HIGHEST_CARGO created in one of the earlier exercises for calculating the highest cargo value.

Therefore, use RTTI instead of the TRY-ENDTRY procedure.

This specifically means that you now no longer need to catch the runtime error for a down cast because the type of reference variable can be determined in advance using an appropriate RTTI call.

Solution 24: RTTI - Querying Object Attributes

Task:

Determine the highest cargo value for all cargo airplanes entered in your application. The cargo airplanes are entered in the class LCL_CARRIER (of the airline).

1. If you have completed the optional exercise for a down cast (determine the highest cargo for an airplane), you can complete your program ZBC401_##_MAIN (where ## is your two-digit group number). Otherwise, copy the template program with all its includes (suggested name for the new program: **ZBC401_##_RTTI_1**).
 - a) Copy the copy template in the usual manner or continue to work with your old program.
2. RTTI should now solve the method GET_HIGHEST_CARGO created in one of the earlier exercises for calculating the highest cargo value.

Therefore, use RTTI instead of the TRY-ENDTRY procedure.

This specifically means that you now no longer need to catch the runtime error for a down cast because the type of reference variable can be determined in advance using an appropriate RTTI call.

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

Result

Include BC401_RTT_S1_CARRIER

```
*-----*
*      CLASS lcl_airplane DEFINITION
*-----*
CLASS lcl_airplane DEFINITION.

...
ENDCLASS.          "lcl_airplane DEFINITION

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

Continued on next page

```
...  
  
ENDCLASS.          "lcl_airplane IMPLEMENTATION  
  
*-----*  
*      CLASS lcl_cargo_plane DEFINITION  
*-----*  
*  
*-----*  
CLASS lcl_cargo_plane DEFINITION INHERITING FROM lcl_airplane.  
  
...  
  
ENDCLASS.          "lcl_cargo_plane DEFINITION  
  
*-----*  
*      CLASS lcl_cargo_plane IMPLEMENTATION  
*-----*  
CLASS lcl_cargo_plane IMPLEMENTATION.  
  
...  
  
ENDCLASS.          "lcl_cargo_plane IMPLEMENTATION  
  
*-----*  
*      CLASS lcl_passenger_plane DEFINITION  
*-----*  
CLASS lcl_passenger_plane DEFINITION INHERITING FROM lcl_airplane.  
  
...  
  
ENDCLASS.          "lcl_passenger_plane DEFINITION  
  
*-----*  
*      CLASS lcl_passenger_plane IMPLEMENTATION  
*-----*  
CLASS lcl_passenger_plane IMPLEMENTATION.  
  
...  
  
ENDCLASS.          "lcl_passenger_plane IMPLEMENTATION  
  
*-----*  
*      CLASS lcl_carrier DEFINITION
```

Continued on next page

```

*-----*
CLASS lcl_carrier DEFINITION.

PUBLIC SECTION.

INTERFACES if_partner.
METHODS:
constructor IMPORTING iv_name TYPE string,
on_airplane_created FOR EVENT airplane_created
OF lcl_airplane
IMPORTING sender,
display_attributes.

PRIVATE SECTION.

DATA:
mv_name      TYPE string,
mt_airplanes TYPE TABLE OF REF TO lcl_airplane.

METHODS:
display_airplanes,
get_max_cargo RETURNING value(rv_max_cargo) TYPE s_plan_car.

ENDCLASS.          "lcl_carrier DEFINITION

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

METHOD constructor.
mv_name = iv_name.
SET HANDLER on_airplane_created FOR ALL INSTANCES.
RAISE EVENT if_partner~partner_created.
ENDMETHOD.         "constructor

METHOD if_partner~display_partner.
display_attributes( ).
ENDMETHOD.         "if_partners~display_partner

METHOD on_airplane_created.
APPEND sender TO mt_airplanes.
ENDMETHOD.         "on_airplane_created

```

Continued on next page

```

METHOD display_attributes.

DATA:
lv_max_cargo TYPE s_plan_car.

SKIP 2.
WRITE: icon_flight AS ICON,
mv_name.
ULINE.
ULINE.
me->display_airplanes( ).

lv_max_cargo = me->get_max_cargo( ).
WRITE: / 'Capacity of biggest cargo plane:'(max),
lv_max_cargo LEFT-JUSTIFIED.

ENDMETHOD.           "display_attributes

METHOD display_airplanes.
DATA: lo_plane TYPE REF TO lcl_airplane.
LOOP AT mt_airplanes INTO lo_plane.
lo_plane->display_attributes( ).
ENDLOOP.
ENDMETHOD.           "display_airplanes

METHOD get_max_cargo.
DATA:
lo_plane TYPE REF TO lcl_airplane,
lo_cargo TYPE REF TO lcl_cargo_plane.

DATA:
lo_descr TYPE REF TO cl_abap_typedescr.

LOOP AT mt_airplanes INTO lo_plane.
lo_descr = cl_abap_typedescr->describe_by_object_ref( lo_plane ).

IF lo_descr->get_relative_name( ) = 'LCL_CARGO_PLANE'.
lo_cargo ?= lo_plane.
IF rv_max_cargo < lo_cargo->get_cargo( ).
rv_max_cargo = lo_cargo->get_cargo( ).
ENDIF.
ENDIF.

```

Continued on next page

```
*      TRY.  
*          lo_cargo ?= lo_plane.  
*  
*          IF rv_max_cargo < lo_cargo->get_cargo( ).  
*              rv_max_cargo = lo_cargo->get_cargo( ).  
*          ENDIF.  
*  
*          CATCH cx_sy_move_cast_error.  
**          plane is not a cargo plane - do nothing  
*      ENDTRY.  
  
ENDLOOP.  
ENDMETHOD.  
        "get_max_cargo  
  
ENDCLASS.  
        "lcl_carrier IMPLEMENTATION
```



Lesson Summary

You should now be able to:

- Query type attributes at runtime
- Create types dynamically



Unit Summary

You should now be able to:

- Create exception classes
- Raise class-based exceptions in ABAP Objects programs
- Propagate class-based exceptions in ABAP Objects programs
- Handle class-based exceptions in ABAP Objects programs
- Map class-based exceptions in ABAP Objects programs to each other
- Query type attributes at runtime
- Create types dynamically



Test Your Knowledge

1. The new exception concept replaces the old one. Thus, all old source code sections must be rewritten. As of *SAP Web AS 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 a TRY-CATCH-ENDTRY block 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. Thus, all old source code sections must be rewritten. As of *SAP Web AS 6.20*, function modules from the SAP standard system automatically raise object-oriented exceptions.

Answer: False

Refer to the relevant section of the lesson.

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 a TRY-CATCH-ENDTRY block and, if so, how the system reacts if the exception is not caught.

Answer: True

Refer to the relevant section of the lesson.

I n t e r n a l U s e S A P P a r t n e r O n l y

I n t e r n a l U s e S A P P a r t n e r O n l y

Internal Use SAP Partner Only

Internal Use SAP Partner Only

Unit 5

Preview and Case Study

Unit Overview

Preview and Case Study



Unit Objectives

After completing this unit, you will be able to:

- Explain some of the new ABAP Objects concepts (such as ABAP Unit or Shared Objects), which are available as of SAP NetWeaver Application Server 6.40
- Explain the concepts relating to object-oriented programming with ABAP Objects demonstrated in this course and apply these in a small, simple application

Unit Contents

Lesson: Preview of Later Topics	440
Lesson: Case Study	452
Exercise 25: Create a UML Class Diagram.....	457
Exercise 26: Develop a Solution for a Test Case	461

Lesson: Preview of Later Topics

Lesson Overview

This lesson is meant to give you a short preview of later object-oriented topics. A more detailed discussion can be found in the respective documentation or in more advanced training courses.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain some of the new ABAP Objects concepts (such as ABAP Unit or Shared Objects), which are available as of SAP NetWeaver Application Server 6.40

Business Example

You want to get a rough overview of some advanced techniques of object-oriented programming with ABAP.

Preview of Later Topics - Shared Objects

As of *SAP NetWeaver Application Server 6.40*, you can buffer data across project using **shared objects**.

Potential Uses of Shared Objects



- Storing a **shopping basket**.

The shopper fills the basket, which is then read by the salesperson later.

- Storing a **catalog**.

Many users access the data at the same time.

Before the introduction of shared objects, ABAP programs could only access this memory area using the EXPORT and IMPORT statements with the additions SHARED BUFFER or SHARED MEMORY. Instances of classes existed only in the internal mode of an ABAP program. With the introduction of shared objects, the shared memory was extended to include the **shared objects memory**, in which shared objects can be stored.

→ **Note:** In general, the objects stored in the shared objects memory are instances of classes. However, as off *SAP NetWeaver 7.0* it is also possible to store dynamically created data objects in the shared objects memory.

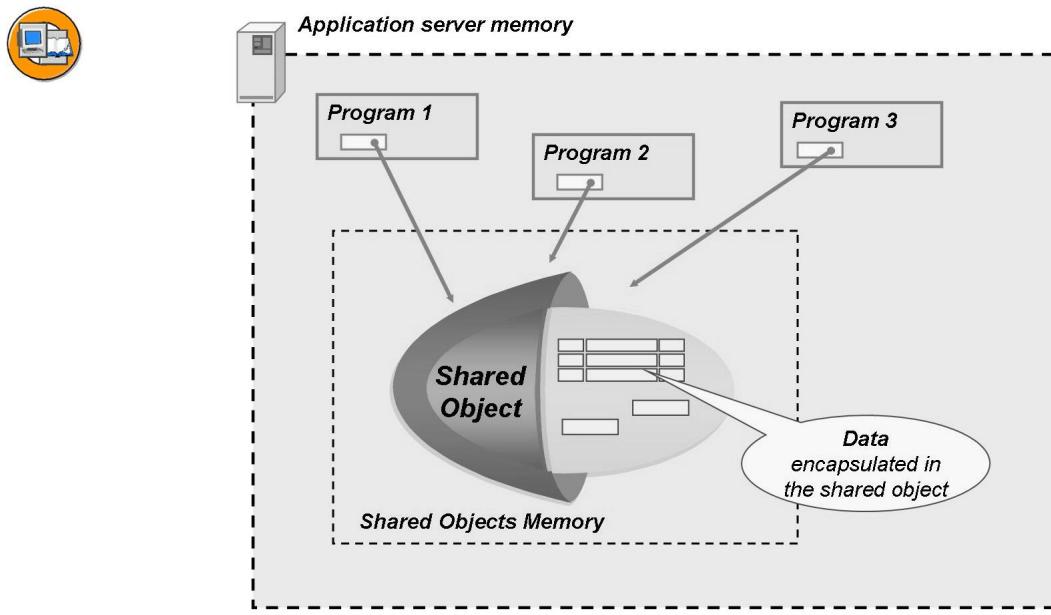


Figure 176: Shared Objects – Basics

Prerequisites for Shared Objects



- Cross-program buffering of data with a small number of write accesses but a large number of read accesses.
(Recommendation: Between once a day and once each hour)
- Simultaneous read accesses are supported.
- Access is regulated by a lock mechanism.
- Data objects that are referenced using data references cannot be stored directly.
- Memory bottlenecks result in runtime errors and have to be caught.

Shared objects are stored in areas of the shared memory. You create and manage areas and their attributes in transaction SHMA. A prerequisite for storing an object in the shared memory is that the class of the object must be defined with the addition SHARED MEMORY ENABLED of the CLASS statement. For global classes, the “shared memory enabled” attribute must be selected in the Class Builder.

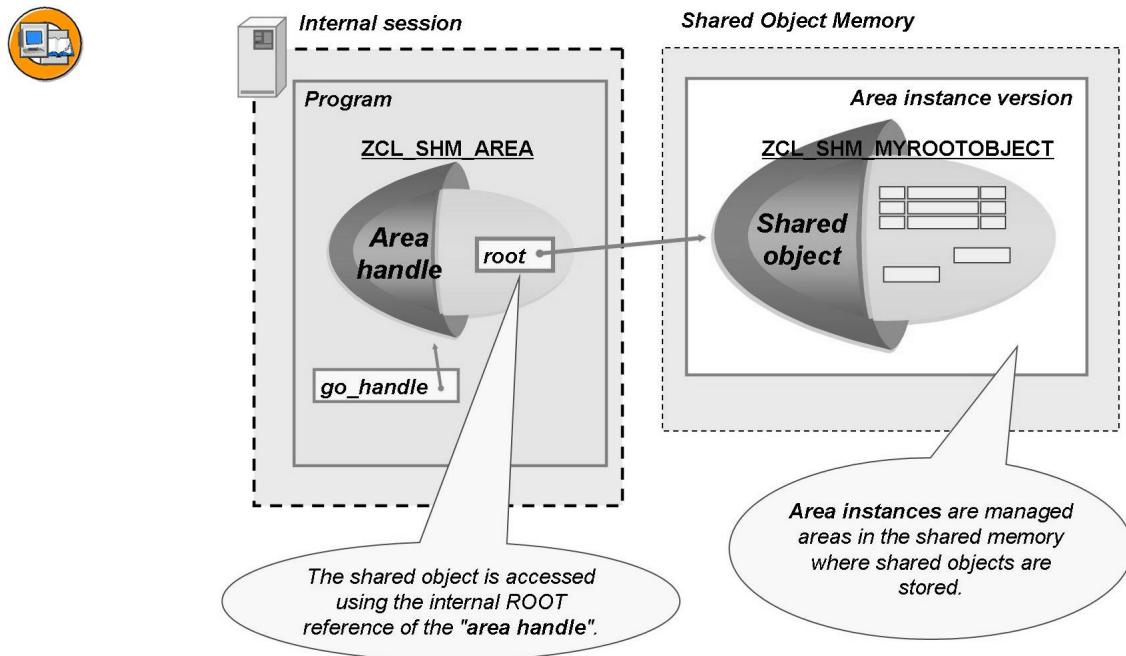


Figure 177: Shared Objects – Areas and Handles

Each area is linked with a global area root class, whose attributes may contain its own data and references to other shared memory-enabled classes. An area is a template for area instances in the shared memory. Several area instances with different names can be created for one area.

When you define an area with transaction SHMA, a global and final area class with the same name is created for the area as a subclass of CL_SHM_AREA.

In an ABAP program, an area is accessed exclusively using the methods of the generated area class. There are static methods for attaching an ABAP program (or its internal mode) to area instances in the shared memory (Attach methods).

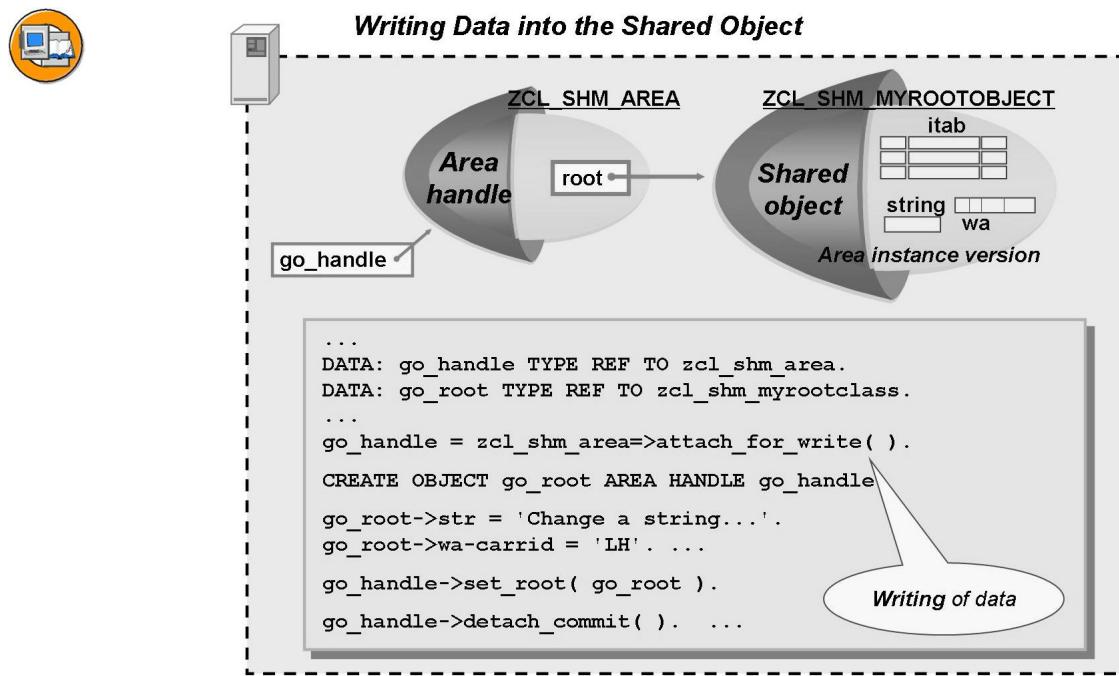


Figure 178: Shared Objects – Example I

With attachment, an instance of the area class is generated as an **area handle** and a lock is set at the same time. The ABAP program can use the area handle to access the attached area instance versions and thus also the shared objects stored there. The area handle also contains the methods for releasing the attachment or the lock (Detach methods).

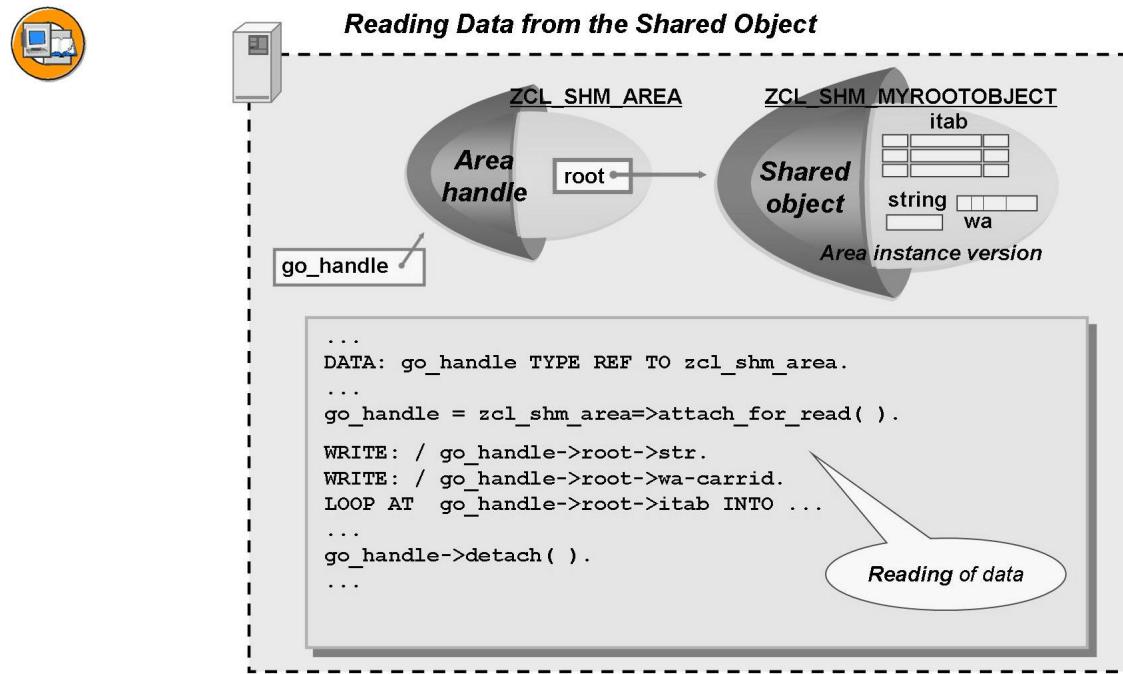


Figure 179: Shared Objects – Example II

→ **Note:** For more details and information, refer to the SAP Library or attend the relevant ABAP course for advanced developers.

Preview of Later Topics – ABAP Unit

As of *SAP NetWeaver Application Server 6.40*, you can create test classes with the **ABAP Unit** tool.

This is a test tool integrated into the ABAP runtime environment, which is used to check the functionality of the code units of a program (module tests).

Individual tests are implemented in various ABAP programs as test methods of local test classes. The tests of several ABAP programs can be grouped together in test items.

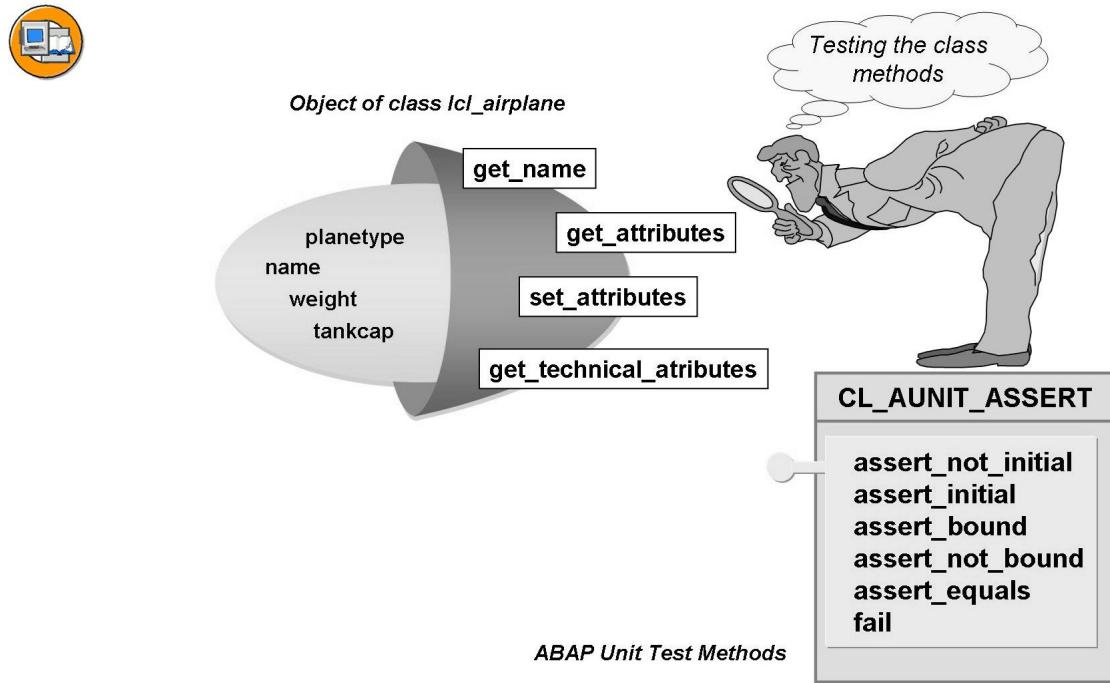


Figure 180: Overview of ABAP UNIT

- The tests are programmed in ABAP Objects.
- The test function is contained in the test methods of test classes.
- These test classes are contained locally in the program that is to be tested. This means that there are no transport or versioning problems.
- If an error occurs, the rest of the test procedure can be controlled.
- The test code is not created in the production environment and it is therefore not executed there; so there is no loss of execution speed for “normal” coding in the production environment!

To be able to test a class and its methods using ABAP UNIT, **test classes** need to be created. They are contained locally in the program to be tested. A test class is defined with the addition FOR TESTING of the CLASS statement. The following example shows a test class that tests the functionality of the method `get_attributes`.



```

REPORT abap_unit_demonstration.

...
CLASS lcl_to_be_tested DEFINITION.
  PUBLIC SECTION.
    METHODS: get_attributes EXPORTING ...
              RAISING ...
    METHODS: ...
ENDCLASS
...

CLASS lcl_test DEFINITION FOR TESTING.
  METHODS test_get_attributes FOR TESTING.
  ...
ENDCLASS.
CLASS lcl_test IMPLEMENTATION.
  METHOD test_get_attributes.
    * create one local airplane for testing
    ...
    lo_plane->get_attributes( IMPORTING
      es_plane = ls_plane_actual ).
    cl_aunit_assert->assert_equals(
      exp = ls_plane_expected
      act = ls_plane_actual ... )
  ENDMETHOD.
ENDCLASS.

```

*Does the method
get_attributes give
the desired result?*

CL_AUNIT_ASSERT

```

assert_not_initial
assert_initial
assert_bound
assert_not_bound
assert_equals
fail

```

Figure 181: ABAP UNIT with the example of the method `get_attributes`

Test methods are defined and implemented in the test class. To some degree, the test classes and test methods form the frame for the calls of the methods to be tested. Therefore, the test methods themselves do not need to be tested! As is typical in these test methods, instances of the classes to be tested are created and their methods are then called and checked.

Test methods have no parameters. They are declared with the addition **FOR TESTING** of the METHODS statement. Every test method corresponds to an ABAP unit test. In a test method, the test coding is implemented by calling certain parts of the program under certain start conditions and checking and logging results and statuses by using the methods of the system class **CL_AUNIT_ASSERT**.

The definition of test classes divides a program into production coding and test coding. The test coding of test classes is not generated in production systems. Access from the production coding to a test class is not possible.

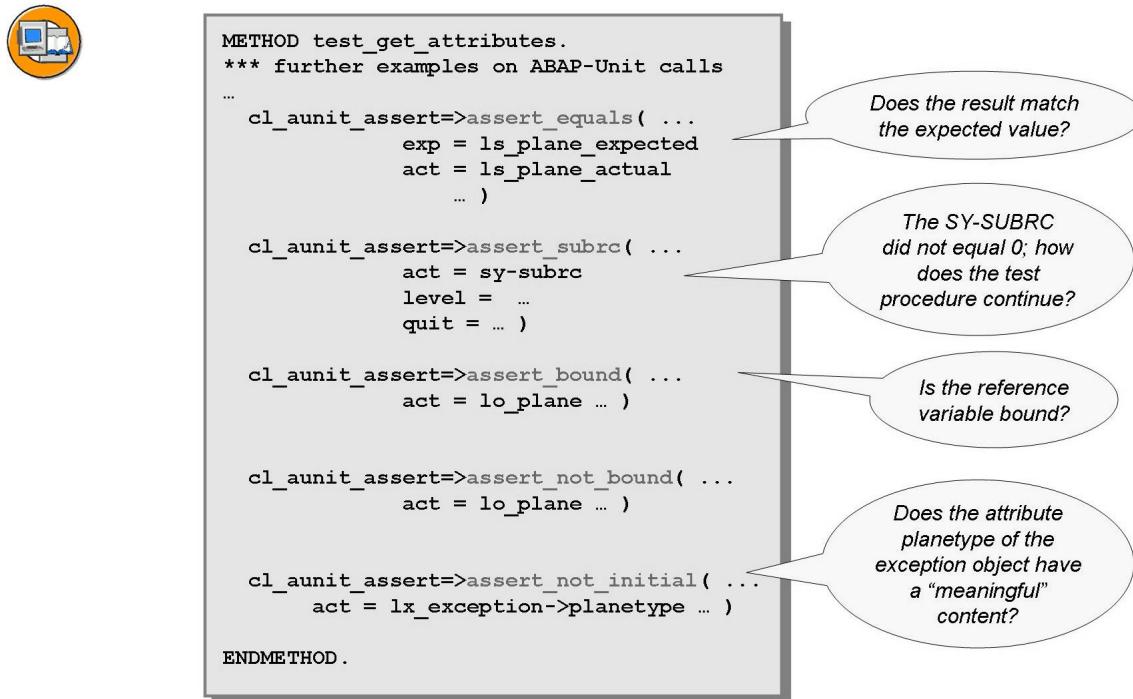


Figure 182: ABAP UNIT Test Calls

Example Program SAPBC401_PVW_UNIT_D1

```

*** Main-Program SAPBC401_PVW_UNIT_D1
REPORT  sapbc401_spc_unit_d1.

INCLUDE bc401_spc_unit_d1_airplane. "local class lcl_airplane
INCLUDE bc401_spc_unit_d1_testclass. "test class lcl_test_plane

START-OF-SELECTION.

* ... creating several planes
* ... but this is not important for testing
...
* Include bc401_spc_unit_d1_testclass
*-----
*      CLASS lcl_test_plane DEFINITION
*-----
CLASS lcl_test_plane DEFINITION FOR TESTING RISK LEVEL HARMLESS.

PRIVATE SECTION.

METHODS:

```

```

        test_create_success    FOR TESTING,
        test_create_exception  FOR TESTING.

ENDCLASS.                      "lcl_test_plane DEFINITION

*-----*
*      CLASS lcl_test_plane IMPLEMENTATION
*-----*
CLASS lcl_test_plane IMPLEMENTATION.
*-----*
*      testing successful instantiation
*-----*
METHOD test_create_success.

DATA:
  ls_plane_exp    TYPE saplane,
  ls_plane_act    TYPE saplane,
  lo_plane        TYPE REF TO lcl_airplane,
  lx_exc          TYPE REF TO cx_invalid_planetype.

* get data for a passenger plane
  SELECT SINGLE * FROM saplane INTO ls_plane_exp
    WHERE seatsmax_b GT 0.

* quit unit test class if no data found
  cl_aunit_assert=>assert_subrc(
    EXPORTING
      act           = sy-subrc
      msg          = `No passenger plane found in table SAPLANE - can't do this test
      level         = cl_aunit_assert=>fatal
      quit         = cl_aunit_assert=>class ).

* check instance creation
TRY.
  CREATE OBJECT lo_plane
    EXPORTING
      iv_name       = 'Frankfurt'
      iv_planetype = ls_plane_exp-planetype.

  CATCH cx_invalid_planetype INTO lx_exc.
ENDTRY.

* assert no exception is raised - quit test method
  cl_aunit_assert=>assert_not_bound(
    msg           = `Constructor raises exception when it shouldn't'
    act           = lx_exc

```

```

*      LEVEL          = CRITICAL
      quit           = cl_aunit_assert=>method .

ENDMETHOD.                      "test_create_success

*-----*
*   testing raising of exceptions by the constructor
*-----*

METHOD test_create_exception.

CONSTANTS:
  lc_planetype TYPE saplane-planetype VALUE '747-500'.

DATA:
  ls_plane      TYPE saplane,
  ls_plane_act  TYPE saplane,
  lv_failed(1)  TYPE c,
  lo_plane      TYPE REF TO lcl_airplane,
  lx_exc        TYPE REF TO cx_invalid_planetype.

* first, make sure planetype does not actually exist
  SELECT SINGLE * FROM saplane INTO ls_plane
    WHERE planetype = lc_planetype.

* quit this test method if planetype acually exists on DB
  cl_aunit_assert=>assert_subrc(
    exp          = 4
    act          = sy-subrc
    msg          = `Test planetype 747-500 actually exists - can't do this
    level        = cl_aunit_assert=>fatal
    quit         = cl_aunit_assert=>method ).

* check instance creation
TRY.
  CREATE OBJECT lo_plane
    EXPORTING
      iv_name      = 'Frankfurt'
      iv_planetype = lc_planetype.

  CATCH cx_invalid_planetype INTO lx_exc.
ENDTRY.

* assert exception is raised - quit test method if not
  cl_aunit_assert=>assert_bound(
    msg          = `Constructor doesn't raise exception when it should'
    act          = lx_exc

```

```
*      LEVEL      = CRITICAL  
quit      = cl_aunit_assert=>method ).
```

```
ENDMETHOD.           "test_create_exception
```

```
ENDCLASS.           "lcl_test_plane IMPLEMENTATION
```

 **Note:** For more information and details, refer to the SAP Library.



Lesson Summary

You should now be able to:

- Explain some of the new ABAP Objects concepts (such as ABAP Unit or Shared Objects), which are available as of SAP NetWeaver Application Server 6.40

Lesson: Case Study

Lesson Overview

In this lesson, you will consolidate and apply the knowledge you have gained in the course so far. You will do this in the best way possible by applying the techniques of ABAP Objects to the case study presented here.



Lesson Objectives

After completing this lesson, you will be able to:

- Explain the concepts relating to object-oriented programming with ABAP Objects demonstrated in this course and apply these in a small, simple application

Business Example

A customer wants to book a flight in a travel agency but the flight is already fully booked. However, the customer is given the option of registering for the flight and being put on the waiting list.

Case Study

The ABAP Objects program to be created here should provide the user (for example, the travel agent) with a screen containing functions for managing customers using flight waiting lists. It should be possible to maintain a waiting list for each flight (row from table SFLIGHT with the key CARRID, CONNID, FLDATE).

→ **Note:** The model shown here is one possible solution to the waiting list requirements described. Of course, the course participants are free to use their own UML class diagram, which will produce a different solution.

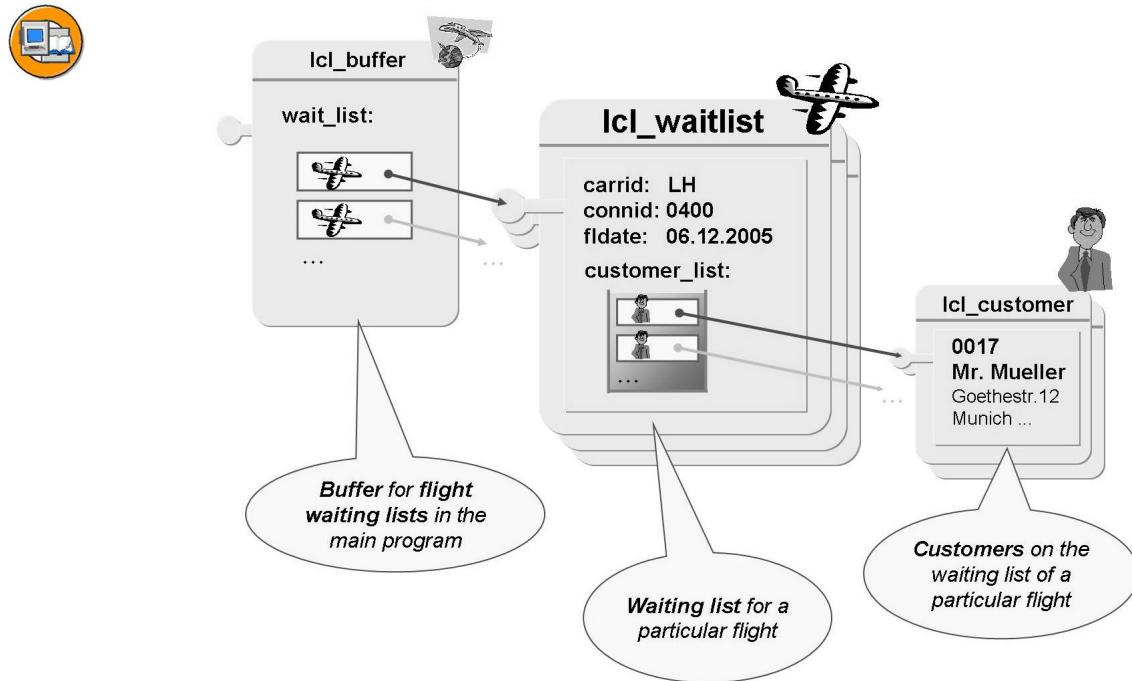


Figure 183: Case Study – Requirements I

All waiting lists are to be stored in a waiting list buffer called WAIT_LIST (which is technically an internal table). In addition to the flight data, the waiting lists are to contain a list of all customers who have registered for this flight. The functions of this application are shown in the following figure.



Maintenance Screen of Travel Agency

Waiting list Airline <input type="text" value="LH"/> Flight number <input type="text" value="0400"/> Date of flight <input type="text" value="04.12.2005"/> Waiting list functions <input type="button" value="Display"/> <input type="button" value="Create"/> <input type="button" value="Delete"/> <input type="button" value="Move up"/>	Customer Customer number <input type="text" value="00003392"/> Customer name <input type="text" value="Luc Picard"/> Date <input type="text" value="14.08.2005"/> Customer → Waiting list <input type="button" value="Add"/> <input type="button" value="Delete"/> <input type="button" value="Position"/>
--	--

Create the waiting list for flight LH, 0400 on December 04, 2005

Add a customer to the waiting list for flight LH, 0400 on December 04, 2005

Figure 184: Case Study – Requirements II

The screen and functions shown here are to be created in accordance with the requirement specifications. The screen layout and flow logic are already available. You now have to implement the actual flow, that is, the “intelligence” of the program using object-oriented techniques.

The individual buttons are to have the following functions:

Display

Displays the waiting list for the specified flight (CARRID, CONNID, FLDATE)

Create

Creates the waiting list for the specified flight (CARRID, CONNID, FLDATE)

Delete

Deletes the waiting list for the specified flight (CARRID, CONNID, FLDATE)

Add

Adds the customer selected on the right of the screen to the flight waiting list selected on the left

Delete (Customer)

Deletes the customer selected on the right of the screen from the flight displayed on the left

Position

Displays the selected customer's position on the waiting list

Customer Type (optional)

Shows whether the selected customer is a private customer or a business customer (you will find the relevant information in the CUSTTYPE field of table SCUSTOM).

Display customers Moved Up (optional)

Displays all customers from the list who were successfully moved up from the waiting list to get a seat on the flight

Status Log (optional)

Displays the number of customers from the list who were successful in getting a seat on a flight.

Exercise 25: Create a UML Class Diagram

Exercise Objectives

After completing this exercise, you will be able to:

- Create one possible UML class diagram to meet a technical specification

Business Example

A travel agency wants to offer its customers an additional service. If customers want to book a flight that is already fully booked, they can have their names put on a waiting list for this flight. If a seat becomes free at a later date, a customer from the waiting list can be placed on the flight. This customer's name is then removed from the waiting list.

Task: Create a UML Class Diagram

You can use the following suggestions but you can also choose a model of your own, which you can then use for an alternate implementation later.

If you do not wish to do this, follow the instructions below to create a class diagram.

1. The **waiting list** should be the central class in the model. Use LCL_WAITLIST as the class name (ZCL_##_WAITLIST if you are using the Class Builder).

The class should have the public attributes CARRID, CONNID, FLDATE and the private attribute CUSTOMER_LIST.

The following public methods should be defined in the UML:

CONSTRUCTOR, ADD, DELETE, GET_POS, GET_FIRST and GET_WAIT_LIST.



Hint: It would actually be more in line with the encapsulation concept to define all of the attributes here as private and then access these attributes of the waiting list with external methods. We suggest that you define the attributes as public here to save time and effort, but this is your choice!

2. **Customers** form another class. Use LCL_CUSTOMER as the class name (ZCL_##_CUSTOMER if you are using the Class Builder).

This class should have the public attribute ID and the private attributes NAME, STREET, CITY and APP_DATE.

Continued on next page

The following public methods should be defined in the UML:
CONSTRUCTOR and GET_ATTRIBUTES

3. As a straightforward solution, you could simply define the **buffer for the waiting lists** in the main program as a normal internal table. However, you want to use one class in this case and therefore the singleton concept is used. Define the buffer for the waiting lists as singleton class LCL_BUFFER (ZCL_##_BUFFER if you are using the Class Builder).

The class should have the attribute BUFFER_LIST: the private, static attribute N_O_ENTRIES and a private, static reference to itself.

The following public methods should be defined in the UML:

CLASS_CONSTRUCTOR (as a static method) and DISPLAY_N_O_ENTRIES



Hint: It would actually be more in line with the encapsulation concept to define the BUFFER_LIST attribute as private in this case and then access this attribute with methods from the main program. In this case, we suggest that you define this attribute as public and then access it externally and directly to save time and effort, but this is your choice!

If you follow the model solution for the class diagram, you must use methods.

4. Define the **relationships** between these classes and assign the **cardinalities** (multiplicity).

You also have the option of mapping the navigation options.

Solution 25: Create a UML Class Diagram

Task: Create a UML Class Diagram

You can use the following suggestions but you can also choose a model of your own, which you can then use for an alternate implementation later.

If you do not wish to do this, follow the instructions below to create a class diagram.

1. The **waiting list** should be the central class in the model. Use LCL_WAITLIST as the class name (ZCL_##_WAITLIST if you are using the Class Builder).

The class should have the public attributes CARRID, CONNID, FLDATE and the private attribute CUSTOMER_LIST.

The following public methods should be defined in the UML:

CONSTRUCTOR, ADD, DELETE, GET_POS, GET_FIRST and GET_WAIT_LIST.



Hint: It would actually be more in line with the encapsulation concept to define all of the attributes here as private and then access these attributes of the waiting list with external methods. We suggest that you define the attributes as public here to save time and effort, but this is your choice!

2. **Customers** form another class. Use LCL_CUSTOMER as the class name (ZCL_##_CUSTOMER if you are using the Class Builder).

This class should have the public attribute ID and the private attributes NAME, STREET, CITY and APP_DATE.

The following public methods should be defined in the UML:
CONSTRUCTOR and GET_ATTRIBUTES

3. a) Refer to the model solution.

3. b) As a straightforward solution, you could simply define the **buffer for the waiting lists** in the main program as a normal internal table. However, you want to use one class in this case and therefore the singleton concept is used. Define the buffer for the waiting lists as singleton class LCL_BUFFER (ZCL_##_BUFFER if you are using the Class Builder).

The class should have the attribute BUFFER_LIST: the private, static attribute N_O_ENTRIES and a private, static reference to itself.

The following public methods should be defined in the UML:

Continued on next page

CLASS_CONSTRUCTOR (as a static method) and DISPLAY_N_O_ENTRIES



Hint: It would actually be more in line with the encapsulation concept to define the BUFFER_LIST attribute as private in this case and then access this attribute with methods from the main program. In this case, we suggest that you define this attribute as public and then access it externally and directly to save time and effort, but this is your choice!

If you follow the model solution for the class diagram, you must use methods.

- a) Refer to the model solution.
4. Define the **relationships** between these classes and assign the **cardinalities** (multiplicity).

You also have the option of mapping the navigation options.

- a) Refer to the model solution.

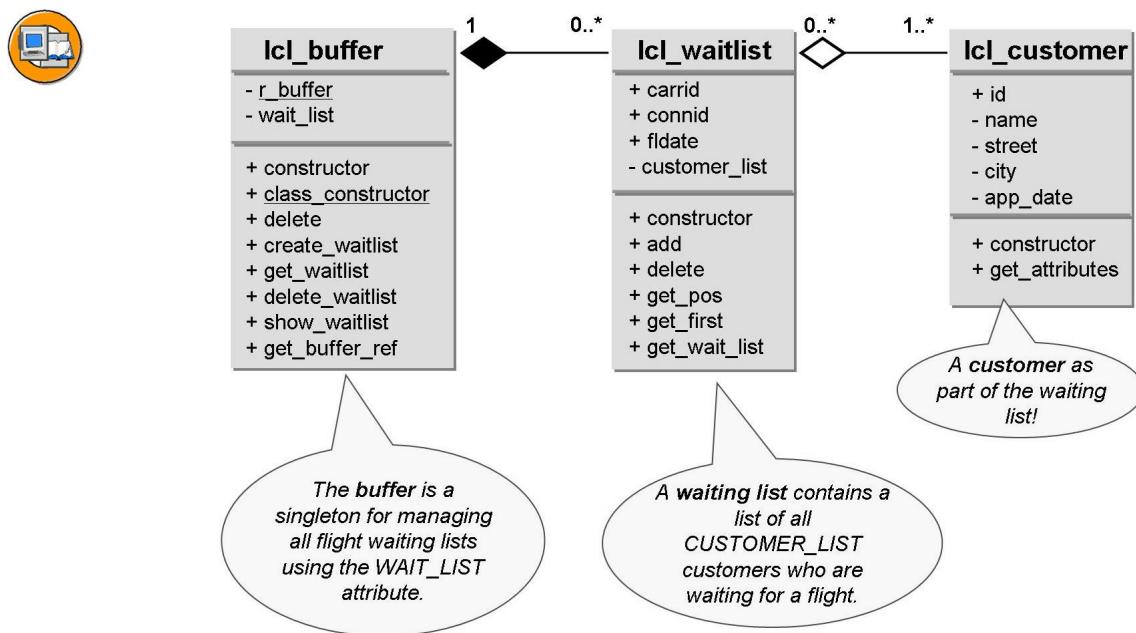


Figure 185: Waiting List: UML Class Diagram

Exercise 26: Develop a Solution for a Test Case

Exercise Objectives

After completing this exercise, you will be able to:

- Use ABAP Objects to develop a solution for a test case
- Apply some of the techniques you have learned in this course

Business Example

A customer wants to book a flight through a travel agency but the flight is already fully booked. However, the customer is offered the option of registering for the waiting list for this flight.

Task:

Convert the definitions from your class diagram into a program.

If you want to use the suggestions proposed in this unit, follow the instructions below. If you have created your own alternate UML class diagram, you can ignore these instructions.

1. Use program SAPBC401_PRJT_WAITLIST as a template. Your program should be called ZBC401_##_PRJS_WAITLIST.
Copy all Include programs from the template and rename them. You should always use the target names specified with your group number (suggested name: ZBC401_##_PRJS_INC_xxx).
2. The copied template already contains the parameters for the application to be created. Familiarize yourself with the program structure.

This program contains a screen on which buttons for all of the required functions are already provided. However, with the exception of the icons in the standard toolbar, these functions still have to be implemented.

Continued on next page

For the meaning of the individual buttons, refer to the screen shown in the figure and the explanatory text in your course notes.

- **Note:** If you have no experience with dynamic screens or dialog programming with ABAP, speak to your course instructor.
- 💡 **Hint:** The display of the customer data on the waiting list (function code SHOW) is not part of this exercise. Therefore, to save time, you can use function module BC401_DISPLAY_DATA. Later, when you are implementing this function, you should familiarize yourself with its interface and use it for display. If you have enough time, you can also use the ALV Grid Control.
- **Note:** To simplify the scenario, avoid persistent storage of the waiting list. In other words, the data is not to be saved to the database.
- **Note:** The focus of this exercise is on the creation of a viable model for which the existing components should be examined. The main point of the exercise is not to implement the program in detail.

Solution 26: Develop a Solution for a Test Case

Task:

Convert the definitions from your class diagram into a program.

If you want to use the suggestions proposed in this unit, follow the instructions below. If you have created your own alternate UML class diagram, you can ignore these instructions.

1. Use program SAPBC401_PRJT_WAITLIST as a template. Your program should be called ZBC401_##_PRJS_WAITLIST.
Copy all Include programs from the template and rename them. You should always use the target names specified with your group number (suggested name: ZBC401_##_PRJS_INC_xxx).
a) Carry out this step in the usual manner.
2. The copied template already contains the parameters for the application to be created. Familiarize yourself with the program structure.

This program contains a screen on which buttons for all of the required functions are already provided. However, with the exception of the icons in the standard toolbar, these functions still have to be implemented.

For the meaning of the individual buttons, refer to the screen shown in the figure and the explanatory text in your course notes.

 **Note:** If you have no experience with dynamic screens or dialog programming with ABAP, speak to your course instructor.



Hint: The display of the customer data on the waiting list (function code SHOW) is not part of this exercise. Therefore, to save time, you can use function module BC401_DISPLAY_DATA. Later, when you are implementing this function, you should familiarize yourself with its interface and use it for display. If you have enough time, you can also use the ALV Grid Control.



Note: To simplify the scenario, avoid persistent storage of the waiting list. In other words, the data is not to be saved to the database.



Note: The focus of this exercise is on the creation of a viable model for which the existing components should be examined. The main point of the exercise is not to implement the program in detail.

Continued on next page

- a) The model solution provided here encapsulates the buffer with the waiting lists in class LCL_BUFFER. The waiting lists in the buffer can be accessed with public methods.

It is also possible to create the buffer in the main program directly as a non-encapsulated internal table. Due to time restrictions, this is recommended, but the decision is up to the you.

Result

REPORT sapbc401_prjs_waitlist_1

```

REPORT  sapbc401_prjs_waitlist_1.

INCLUDE  sapbc401_prjs_inc_1.

DATA      ok_code  LIKE sy-ucomm.
TABLES:  sdyn_conn, scustom.

TYPES: ty_wait_list  TYPE STANDARD TABLE OF REF TO lcl_waitlist.

DATA: r_buffer  TYPE REF TO lcl_buffer,
      r_customer  TYPE REF TO lcl_customer,
      r_waitlist  TYPE REF TO lcl_waitlist,
      r_exc       TYPE REF TO cx_root.

DATA: wa_sflight  TYPE sflight.

LOAD-OF-PROGRAM.
*#####
*** Instantiate the Buffer which is a singleton !
r_buffer = lcl_buffer=>get_buffer_ref( ).

*** some dynpro initializations ****
sdyn_conn-carrid = 'LH'.
sdyn_conn-connid = '0400'.
sdyn_conn-fldate = '20040710'.
scustom-id = '00003392'.

START-OF-SELECTION.
*#####

CALL SCREEN 100.

```

Continued on next page

```
INCLUDE sapbc401_prjs_waitlist_o01.
```

Include SAPBC401_PRJS_WAITLIST_I01

```
-----
***INCLUDE BC401_CSS1_WAITLIST_I01 .
-----

*&-----*
*&      Module  check_and_create_cust  INPUT
*&-----*
*      text
*-----*

MODULE check_and_create_cust INPUT.
  SELECT SINGLE name street city FROM scustom
    INTO CORRESPONDING FIELDS OF scustom
    WHERE id = scustom-id.

  IF sy-subrc <> 0.
    CLEAR: scustom-name, scustom-city.
    MESSAGE e916(BC401) WITH scustom-id.
  * Customernumber & does not exist. Please correct !
  ENDIF.

ENDMODULE.          " check_and_create_cust  INPUT

-----*
*&      Module  user_command_0100  INPUT
*&-----*
*      text
*-----*

MODULE user_command_0100 INPUT.
  DATA: text TYPE string.
  DATA: n_o_lines TYPE i.
  DATA: save_ok TYPE sy-ucomm.

  CLEAR r_waitlist.
  save_ok = ok_code.
  CLEAR ok_code.
  CASE save_ok.
    WHEN 'BACK'.
      LEAVE TO SCREEN 0.
```

Continued on next page

```

WHEN 'CREATE'. "#####
TRY.
  r_buffer->create_wait_list( EXPORTING
    im_carrid = sdyn_conn-carrid
    im_connid = sdyn_conn-connid
    im_fldate = sdyn_conn-fldate ).

MESSAGE s905(BC401)
  WITH sdyn_conn-carrid sdyn_conn-connid sdyn_conn-fldate.

CATCH cx_wait_list INTO r_exc.
  text = r_exc->get_text( ).

MESSAGE s901(BC401) WITH sdyn_conn-carrid sdyn_conn-connid.

ENDTRY.

WHEN 'SHOW'. "#####
DATA: it_customer_list TYPE STANDARD TABLE OF bc401_typed_cust.
DATA: wa_cust LIKE LINE OF it_customer_list.
DATA: r_customer_list TYPE TABLE OF REF TO lcl_customer.
DATA: r_cust TYPE REF TO lcl_customer.

r_waitlist = r_buffer->get_wait_list(
  im_carrid = sdyn_conn-carrid
  im_connid = sdyn_conn-connid
  im_fldate = sdyn_conn-fldate ).

IF r_waitlist IS BOUND.
  r_waitlist->get_wait_list( IMPORTING
    ex_wait_list = r_customer_list ).
  n_o_lines = LINES( r_customer_list ).

  IF n_o_lines > 0.
    CLEAR it_customer_list.
    LOOP AT r_customer_list INTO r_cust.
      r_cust->get_attributes( IMPORTING ex_cust = wa_cust ).
      INSERT wa_cust INTO TABLE it_customer_list.
    ENDLOOP.

    " show dynpro with all customers and their data
    CALL FUNCTION 'BC401_DISPLAY_DATA'
      EXPORTING
        im_list = it_customer_list.
  ELSE.
    MESSAGE s903(BC401)
      WITH sdyn_conn-carrid sdyn_conn-connid
      sdyn_conn-fldate.
    *      The waitinglist for flight &1 &2 &3 has no entries !
  ENDIF.
ENDIF.

```

Continued on next page

```

ENDIF.

ELSE.
MESSAGE s902(BC401) WITH sdyn_conn-carrid sdyn_conn-connid
sdyn_conn-fldate.
* There is no waitinglist for flight Flug &1 &2 &3 !

ENDIF.

WHEN 'DELETE'.    #####"
TRY.
    r_buffer->delete_wait_list( EXPORTING
        im_carrid = sdyn_conn-carrid
        im_connid = sdyn_conn-connid
        im_fldate = sdyn_conn-fldate ).

MESSAGE s906(BC401) WITH
    sdyn_conn-carrid sdyn_conn-connid sdyn_conn-fldate.
* Waitinglist for flight &1 &2 &3 has been deleted succesfully !
CATCH cx_wait_list INTO r_exc.
    text = r_exc->get_text( ).
    MESSAGE text TYPE 'S'.
ENDTRY.

WHEN 'FIRST'.    #####"
r_waitlist = r_buffer->get_wait_list( im_carrid = sdyn_conn-carrid
im_connid = sdyn_conn-connid
im_fldate = sdyn_conn-fldate ).
IF r_waitlist IS BOUND.
TRY.
    r_waitlist->get_first( ).
    MESSAGE s907(BC401)
        WITH sdyn_conn-carrid sdyn_conn-connid
        sdyn_conn-fldate.
* The first customer on waitinglist &1 &2 &3
* has been deleted.

CATCH cx_wait_list INTO r_exc.
    text = r_exc->get_text( ).
    MESSAGE text TYPE 'I'.
ENDTRY.
ELSE.
MESSAGE s902(BC401) WITH sdyn_conn-carrid sdyn_conn-connid
sdyn_conn-fldate.
* There is no waitinglist for flight &1 &2 &3 !
ENDIF.

```

Continued on next page

```

WHEN 'ADD'.      "#####
  IF scustom-id IS INITIAL.
    MESSAGE s908(BC401).
  *      Please enter a correct customernumber !
    EXIT.
  ENDIF.
r_waitlist = r_buffer->get_wait_list( im_carrid = sdyn_conn-carrid
                                         im_connid = sdyn_conn-connid
                                         im_fldate = sdyn_conn-fldate ).
  IF r_waitlist IS BOUND.
    TRY.
      CREATE OBJECT r_customer EXPORTING im_id = scustom-id
                                         im_name = scustom-name
                                         im_street = scustom-street
                                         im_city = scustom-city
                                         im_app_date = sy-datum.
      r_waitlist->add( r_customer ).
      MESSAGE s910(BC401)
        WITH scustom-id sdyn_conn-carrid sdyn_conn-connid
             sdyn_conn-fldate.
  *      Customer &1 was appended to waitinglist for flight &2 &3 &4
      CATCH cx_wait_list INTO r_exc. "wann könnte das passieren ???
        text = r_exc->get_text( ).
        MESSAGE text TYPE 'I'.
    ENDTRY.
  ELSE.
    MESSAGE s902(BC401) WITH sdyn_conn-carrid sdyn_conn-connid
                           sdyn_conn-fldate.
  *      There is no waitinglist for flight &1 &2 &3 !
  ENDIF.

WHEN 'DEL'.      "#####
  IF scustom-id IS INITIAL.
    MESSAGE s908(BC401).
  *      Please select a valid customernumber !
    EXIT.
  ENDIF.
r_waitlist = r_buffer->get_wait_list( im_carrid = sdyn_conn-carrid
                                         im_connid = sdyn_conn-connid
                                         im_fldate = sdyn_conn-fldate ).
  IF r_waitlist IS BOUND.

```

Continued on next page

```

IF NOT r_customer IS BOUND.
MESSAGE s911(BC401).
ELSE.
TRY.
r_waitlist->delete( r_customer ).
MESSAGE s912(BC401)
WITH scustom-id sdyn_conn-carrid sdyn_conn-connid
sdyn_conn-flddate.
*
Customer &1 was deleted from waitinglist for flight &2 &3 &4 !

CATCH cx_wait_list INTO r_exc.
text = r_exc->get_text( ).
MESSAGE text TYPE 'I'.
ENDTRY.
ENDIF.
ELSE.
MESSAGE s902(BC401) WITH sdyn_conn-carrid sdyn_conn-connid
sdyn_conn-flddate.
*
There is no waitinglist for flight &1 &2 &3 !
ENDIF.

WHEN 'POS'.      #####
DATA pos LIKE sy-tabix.
DATA flight TYPE string.
DATA fldate(10) TYPE c.

IF scustom-id IS INITIAL.
MESSAGE s908(BC401).
*
Please enter a valid customernumber !
EXIT.
ENDIF.
r_waitlist = r_buffer->get_wait_list( im_carrid = sdyn_conn-carrid
im_connid = sdyn_conn-connid
im_fldate = sdyn_conn-flddate ).
IF r_waitlist IS BOUND.
TRY.
CREATE OBJECT r_customer EXPORTING im_id = scustom-id
im_name = scustom-name
im_street = scustom-street
im_city = scustom-city
im_app_date = sy-datum.
r_waitlist->get_pos( EXPORTING im_customer = r_customer
IMPORTING ex_pos      = pos ).
WRITE sdyn_conn-flddate TO fldate.

```

Continued on next page

```

        CONCATENATE sdyn_conn-carrid sdyn_conn-connid fldate
        INTO flight SEPARATED BY space.
        MESSAGE i913(BC401) WITH scustom-id flight pos.
*
* Customer &1 is held on list for flight &2 at position &3
* CATCH cx_wait_list INTO r_exc.
*   text = r_exc->get_text( ).
*   MESSAGE text TYPE 'I'.
*
*ENDTRY.

ELSE.
MESSAGE s902(BC401) WITH sdyn_conn-carrid sdyn_conn-connid
sdyn_conn-fldate.
*
* There is no waitinglist for flight &1 &2 &3 !
ENDIF.

ENDCASE.

ENDMODULE.          " user_command_0100  INPUT

*-----*
*&      Module  check_flight  INPUT
*-----*
*      text
*-----*
MODULE check_flight INPUT.
SELECT SINGLE * FROM sflight INTO wa_sflight
WHERE      carrid = sdyn_conn-carrid
AND    connid = sdyn_conn-connid
AND    fldate = sdyn_conn-fldate.

IF sy-subrc <> 0.
MESSAGE e904(BC401).
* Please select a valid flight !
ENDIF.
ENDMODULE.          " check_flight  INPUT

*-----*
*&      Module  exit  INPUT
*-----*
*      text
*-----*
MODULE exit INPUT.

```

Continued on next page

```

CASE ok_code.
  WHEN 'CANCEL'.
    CLEAR: scustom, sdyn_conn.
    LEAVE TO SCREEN 100.
  WHEN 'EXIT'.
    LEAVE PROGRAM.

ENDCASE.

ENDMODULE.          " exit INPUT

```

Include SAPBC401_PRJS_WAITLIST_001

```

*&-----
*&   Include      SAPBC401_PRJS_WAITLIST_001
*&-----

*&-----
*&   Module      STATUS_0100  OUTPUT
*&-----


MODULE status_0100 OUTPUT.
  SET PF-STATUS 'DYNPRO_100'.
  SET TITLEBAR 'TITLE_100'.

ENDMODULE.          " STATUS_0100  OUTPUT

*&-----
*&   Module      clear_ok_code  OUTPUT
*&-----
*   text
*&-----


MODULE clear_ok_code OUTPUT.
  CLEAR ok_code.
ENDMODULE.          " clear_ok_code  OUTPUT

```

Include SAPBC401_PRJS_INC_1

```

*&-----
*&   Include      SAPBC401_PRJS_INC_1
*&-----


*-----
```

Continued on next page

```

*      CLASS lcl_customer DEFINITION
*-----
CLASS lcl_customer DEFINITION.

PUBLIC SECTION.

METHODS constructor IMPORTING im_id TYPE s_customer
          im_name TYPE s_custname
          im_street TYPE s_street
          im_city TYPE city
          im_app_date TYPE d.

METHODS get_attributes EXPORTING ex_cust TYPE bc401_typd_cust.

DATA: id      TYPE s_customer READ-ONLY.

PRIVATE SECTION.

DATA: name      TYPE s_custname,
      street    TYPE s_street,
      city      TYPE city,
      app_date  TYPE d.

ENDCLASS.           "lcl_customer DEFINITION

*-----
*      CLASS lcl_customer IMPLEMENTATION
*-----
CLASS lcl_customer IMPLEMENTATION.

METHOD constructor.
  id = im_id.
  name = im_name.
  street = im_street.
  city = im_city.
  app_date = im_app_date.
ENDMETHOD.          "constructor

METHOD get_attributes.
  ex_cust-id = id.
  ex_cust-name = name.
  ex_cust-street = street.
  ex_cust-city = city.
  ex_cust-app_date = app_date.
ENDMETHOD.          "get_attributes

```

Continued on next page

Internal Use SAP Partner Only

```

ENDCLASS.                                     "lcl_customer IMPLEMENTATION

*-----*
*      CLASS lcl_waitlist DEFINITION
*-----*
CLASS lcl_waitlist DEFINITION.
PUBLIC SECTION.

TYPES ty_customer_list TYPE
STANDARD TABLE OF REF TO lcl_customer WITH DEFAULT KEY.

METHODS constructor IMPORTING im_carrid TYPE sflight-carrid
im_connid TYPE sflight-connid
im_fldate TYPE sflight-fldate.

METHODS add IMPORTING im_customer TYPE REF TO lcl_customer
RAISING   cx_wait_list.

METHODS delete IMPORTING im_customer TYPE REF TO lcl_customer
RAISING   cx_wait_list.

METHODS get_pos IMPORTING im_customer TYPE REF TO lcl_customer
EXPORTING ex_pos LIKE sy-tabix
RAISING   cx_wait_list.

METHODS get_wait_list EXPORTING ex_wait_list
TYPE ty_customer_list.

METHODS get_first     RAISING   cx_wait_list.

DATA:
carrid TYPE sflight-carrid READ-ONLY,
connid TYPE sflight-connid READ-ONLY,
fldate TYPE sflight-fldate READ-ONLY.

PRIVATE SECTION.
DATA:   customer_list TYPE ty_customer_list.

ENDCLASS.                                     "lcl_waitlist DEFINITION

*-----*
*      CLASS lcl_waitlist IMPLEMENTATION
*-----*
CLASS lcl_waitlist IMPLEMENTATION.

```

Continued on next page

Internal Use SAP Partner Only

```

METHOD constructor.
  carrid = im_carrid.
  connid = im_connid.
  fldate = im_fldate.
ENDMETHOD.           "constructor

METHOD get_wait_list.  -----
  ex_wait_list = customer_list.
ENDMETHOD.           "get_wait_list

METHOD add.  -----
  READ TABLE customer_list  WITH KEY
    table_line->id = im_customer->id TRANSPORTING NO FIELDS.
*** customer is not in wait_list
  IF sy-subrc <> 0.
    INSERT im_customer INTO TABLE customer_list.
  ELSE.
***   customer is already in wait_list for this flight !!!
    RAISE EXCEPTION TYPE cx_wait_list
      EXPORTING
        textid = cx_wait_list=>cx_wait_list_customer_there
        customer = im_customer->id
        carrid = carrid
        connid = connid
        fldate = fldate.
  ENDIF.
ENDMETHOD.           "add

METHOD get_first.    -----
  DATA r_cust TYPE REF TO lcl_customer.

  READ TABLE customer_list INTO r_cust INDEX 1.
  IF sy-subrc <> 0.

    RAISE EXCEPTION TYPE cx_wait_list
      EXPORTING
        textid = cx_wait_list=>cx_wait_list_no_entry
        carrid = carrid
        connid = connid
        fldate = fldate.
  ELSE.
    DELETE customer_list INDEX 1.
  ENDIF.
ENDMETHOD.           "get_first

```

Continued on next page

```

METHOD delete.      -----
  DELETE customer_list
    WHERE table_line->id = im_customer->id.
  IF sy-subrc <> 0.

    RAISE EXCEPTION TYPE cx_wait_list
    EXPORTING
      textid = cx_wait_list=>cx_wait_list_customer_notthere
      customer = im_customer->id
      carrid = carrid
      connid = connid
      fldate = fldate.

  ENDIF.
ENDMETHOD.          "delete

METHOD get_pos.     -----
  READ TABLE customer_list
    WITH KEY table_line->id = im_customer->id
    TRANSPORTING NO FIELDS.

  IF sy-subrc = 0.
    ex_pos = sy-tabix.
  ELSE.
    RAISE EXCEPTION TYPE cx_wait_list
    EXPORTING
      textid = cx_wait_list=>cx_wait_list_customer_notthere
      customer = im_customer->id
      carrid = carrid
      connid = connid
      fldate = fldate.

  ENDIF.
ENDMETHOD.          "get_pos

ENDCLASS. "lcl_waitlist IMPLEMENTATION

*-----*
*      CLASS lcl_buffer DEFINITION
*-----*
* implemented as SINGLETON, should exist only once !
*-----*
CLASS lcl_buffer DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
"-----

```

Continued on next page

```

CLASS-METHODS: class_constructor.

CLASS-METHODS: get_buffer_ref RETURNING value(re_buffer)
                TYPE REF TO lcl_buffer.

METHODS: create_wait_list IMPORTING im_carrid TYPE s_carr_id
                im_connid TYPE s_conn_id
                im_fldate TYPE sflight-fldate
                RETURNING value(re_waitlist)
                TYPE REF TO lcl_waitlist
                RAISING cx_wait_list.

METHODS: get_wait_list IMPORTING im_carrid TYPE s_carr_id
                im_connid TYPE s_conn_id
                im_fldate TYPE sflight-fldate
                RETURNING value(re_waitlist)
                TYPE REF TO lcl_waitlist.

METHODS: delete_wait_list IMPORTING im_carrid TYPE s_carr_id
                im_connid TYPE s_conn_id
                im_fldate TYPE sflight-fldate
                RAISING cx_wait_list.

METHODS: delete_all_wait_lists.

PRIVATE SECTION.

"-----
CLASS-DATA: r_buffer TYPE REF TO lcl_buffer.
DATA: wait_list TYPE STANDARD TABLE OF
      REF TO lcl_waitlist.

ENDCLASS.          "lcl_buffer DEFINITION

*-----*
*      CLASS lcl_buffer IMPLEMENTATION
*-----*
CLASS lcl_buffer IMPLEMENTATION.

METHOD class_constructor.
  CREATE OBJECT r_buffer.
ENDMETHOD.          "class_constructor

METHOD get_buffer_ref.
  re_buffer = r_buffer.
ENDMETHOD.          "get_buffer_ref

```

Continued on next page

```

METHOD create_wait_list.  -----
DATA: r_waitlist TYPE REF TO lcl_waitlist.
r_waitlist = get_wait_list( im_carrid = im_carrid
                           im_connid = im_connid
                           im_fldate = im_fldate ).
IF NOT r_waitlist IS BOUND.
  CREATE OBJECT r_waitlist EXPORTING im_carrid = im_carrid
                                             im_connid = im_connid
                                             im_fldate = im_fldate.
  INSERT r_waitlist INTO TABLE wait_list.
ELSE.
  RAISE EXCEPTION TYPE cx_wait_list
    EXPORTING
      carrid = im_carrid
      connid = im_connid
      fldate = im_fldate
      textid = cx_wait_list=>cx_wait_list_exists.
ENDIF.
ENDMETHOD.                               "create_wait_list

METHOD get_wait_list.  -----
READ TABLE wait_list INTO re_waitlist WITH KEY
  table_line->carrid = im_carrid
  table_line->connid = im_connid
  table_line->fldate = im_fldate.
ENDMETHOD.                               "get_wait_list

METHOD delete_wait_list.  -----
DATA: r_waitlist TYPE REF TO lcl_waitlist.
r_waitlist = get_wait_list( im_carrid = im_carrid
                           im_connid = im_connid
                           im_fldate = im_fldate ).
IF NOT r_waitlist IS BOUND.
  RAISE EXCEPTION TYPE cx_wait_list
    EXPORTING
      carrid = im_carrid
      connid = im_connid
      fldate = im_fldate.
ELSE.
  DELETE wait_list
    WHERE      table_line->carrid = im_carrid
              AND   table_line->connid = im_connid
              AND   table_line->fldate = im_fldate.
ENDIF.

```

Continued on next page

```
ENDMETHOD.           "delete_wait_list

METHOD delete_all_wait_lists.  -----
    CLEAR wait_list.
    " ATTENTION: in our case, this also deletes all customer-objects.
    " --> COMPOSITION in class-diagram ?
ENDMETHOD.           "delete_all_wait_lists

ENDCLASS.            "lcl_buffer IMPLEMENTATION
```

Flow Logic of Screen 100

PROCESS BEFORE OUTPUT.

```
MODULE status_0100.
```

```
MODULE clear_ok_code.
```

PROCESS AFTER INPUT.

```
MODULE exit AT EXIT-COMMAND.
```

CHAIN.

```
FIELD: sdyn_conn-carrid,
       sdyn_conn-connid,
       sdyn_conn-fldate.
```

```
MODULE check_flight.
```

```
ENDCHAIN.
```

CHAIN.

```
FIELD: scustom-id,
       scustom-name,
       scustom-city.
```

```
MODULE check_and_create_cust.  " ON CHAIN-REQUEST.
```

```
ENDCHAIN.
```

```
MODULE user_command_0100.
```



Lesson Summary

You should now be able to:

- Explain the concepts relating to object-oriented programming with ABAP Objects demonstrated in this course and apply these in a small, simple application



Unit Summary

You should now be able to:

- Explain some of the new ABAP Objects concepts (such as ABAP Unit or Shared Objects), which are available as of SAP NetWeaver Application Server 6.40
- Explain the concepts relating to object-oriented programming with ABAP Objects demonstrated in this course and apply these in a small, simple application

Internal Use SAP Partner Only



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

Index

A

abstraction, 17
ACTIVATION, 204
 aggregation, 22, 49
 alias name, 156–157, 165
ALV Grid Control, 264
 association, 20
 binary and recursive, 20
 attribute, 35
 AGENT, 357
 PREVIOUS, 390
 private, 37
 public, 37
 static, 39

B

basis class, 357
Business Add-Ins, 269

C

CALL METHOD, 51–52
 EXCEPTIONS, 41
 EXPORTING, 41
 IMPORTING, 41
 cardinality, 20
CHANGING, 41
 class, 15
 abstract, 316
 CL_ABAP_CLASSDE-
 SCR, 424
 CL_ABAP_STRUCTDE-
 SCR, 423
 CL_ABAP_TYPE-
 DESCR, 421, 423–424
 CL_GUI_ALV_GRID,
 265
 CL_GUI_CUS-
 TOM_CONTAINER,
 265
 definition part, 34

final, 317
 friendship, 326
 instantiation, 321
 persistent, 356
Class
 Definition, 34
CLASS, 245
 ABSTRACT, 316
 CREATE, 321
 Definition, 34
 FINAL, 317
 FRIENDS, 327
 INHERITING FROM,
 105
 PRIVATE SECTION, 38
 PUBLIC SECTION, 38
 class actor, 356
 class agent, 356
 class attribute, 39
Class Builder
 display settings, 261
 editing class components,
 251
 class diagram, 19
CLASS EVENTS, 201
 class method, 44
CLASS_CONSTRUCTOR,
 58
CLASS-METHODS, 44
 classes
 without multiple
 instantiation, 324
COMMIT WORK, 357
 component interface, 164
 composition, 22
 compound interface, 164
CONSTANTS, 36
 constructor, 56, 107
 static, 58

- CONSTRUCTOR, 56
container control, 265
CREATE OBJECT, 47
- D**
- DATA
 TYPE REF TO, 47
database management system
 object-oriented, 356
- DEFAULT, 41
- DEFINITION, 34
definition part, 34
- delegation, 25
delegation principle, 11
- DESCRIBE FIELD, 420
- DESCRIBE TABLE, 420
- destructor, 56
- down cast, 117
- Down-Cast, 161
- Down-cast assignment
 operator, 161
- downcast, 117
- E**
- encapsulation, 6, 37
Enjoy Demo Center, 268
event
 static, 198
- event control, 12
- event handler method, 199
- EVENTS, 201
- exception
 class-based, 370
 definition, 370
 mapping, 389
 propagate, 373, 381
- exception chain, 389
- exception class, 371
 CX_DY-
 NAMIC_CHECK, 372,
 384
 CX_NO_CHECK, 372,
 385
 CX_OS_OBJECT_EX-
 ISTING, 357
- CX_OS_OB-
 JECT_NOT_FOUND,
 357
CX_ROOT, 372, 390
CX_STATIC_CHECK,
 372, 384
CX_SY_MOVE_CAST_ER-
 ROR, 119, 162
global, 371
inheritance hierarchy, 384
local, 371
superclass, 384
- exception handling
 CATCH, 372–373
 CLEANUP, 373, 383
 static check, 384
 syntax check, 384
 TRY, ENDTRY, 372–373
- exception instance, 371
- F**
- friend concept, 326
friendship relationship, 326
 inheritance, 328
- function group
 data management, 6
- G**
- Garbage Collector, 48
generalization, 23, 104
global class
 print, 251
 test, 252
- H**
- handler table, 205
- I**
- indicator
 OO transaction model,
 359
information hiding, 37
inheritance, 12
instance, 16
instance attribute, 39
instance constructor, 56, 107
 visibility section, 321
- instance event, 198

instance method, 44
 instantiation
 class, 46
 interface
 class, 37
 compound, 164
 method, 41
 INTERFACE, 156
 interface resolution operator,
 156–157
 INTERFACES, 156
 IS INITIAL, 48

M

ME, 58
 method
 abstract, 316
 call, 51–52
 CREATE_PERSISTENT,
 357
 definition, 41
 DESCRIBE_BY_DATA,
 423
 DESCRIBE_BY_OB-
 JECT_REF, 424
 final, 317
 functional, 53
 GET_PERSISTENT, 357
 GET_RELAT-
 TIVE_NAME, 425
 GET_TEXT, 374, 379
 parameter, 41
 private, 43
 public, 43
 redefinition, 106
 REFRESH_TABLE_DIS-
 PLAY, 266
 SET_TA-
 BLE_FOR_FIRST_DIS-
 PLAY, 266
 signature, 41
 static, 44, 317
 METHODS, 44
 ABSTRACT, 316
 FINAL, 317
 FOR EVENT, 202
 MOVE, 48, 161

down cast assignment
 operator, 117
 multiple inheritance, 105
 multiple instantiation, 10
 multiplicity, 20

N

namespace
 class, 43
 narrowing cast assignment
 operator, 117
 null reference, 47

O

object
 behavior, 19
 classification, 15
 identity, 16
 persistent, 355–356
 status, 16, 19
 transient, 355
 object diagram, 24
 object GUID, 356
 object link, 20
 Object Management Group
 (OMG), 18
 OMG, 18
 OO transaction, 359
 OO transaction model, 359
 OPTIONAL, 41

P

package, 18
 Persistence Service, 356
 polymorphism, 12, 115
 Polymorphism, 160
 PRIVATE SECTION, 38
 programming model
 object-oriented, 10
 procedural, 5
 PROTECTED SECTION,
 109
 PUBLIC SECTION, 38

R

RAISE EVENT, 201
 RAISING, 382
 READ-ONLY, 36, 38

- redefinition, 106
Refactoring Assistant, 262
reference
 independent, 48
reference variable, 47
Registration, 203, 205
RETURNING, 41
role, 20
role design pattern, 121
RTTC, 425
RTTI, 420
Run Time Type Creation, 425
runtime error
 catchable, 370
Runtime Type Identification
 (RTTI), 420
- S**
- SAP Control Framework, 263
self-reference, 58
SENDER, 202
sequence diagram, 25
SET HANDLER, 204
 ACTIVATION, 204
 FOR ALL INSTANCES,
 204
Signature, 41
singleton, 324
- specialization, 23, 103–104
static attribute, 39
static constructor, 58
static method, 44
subclass, 103–104
SUPER, 106
superclass, 103–104
- T**
- transaction code, 359
transaction service, 359
type
 dynamic, 112
 static, 112
TYPE REF TO, 47
TYPES, 36
- U**
- UML, 18
Unified Modeling Language,
 18
Up-Cast, 158
upcast, 112
- V**
- visibility section
 protected, 109

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.