BC404 ABAP Objects: Object-Oriented Programming in R/3

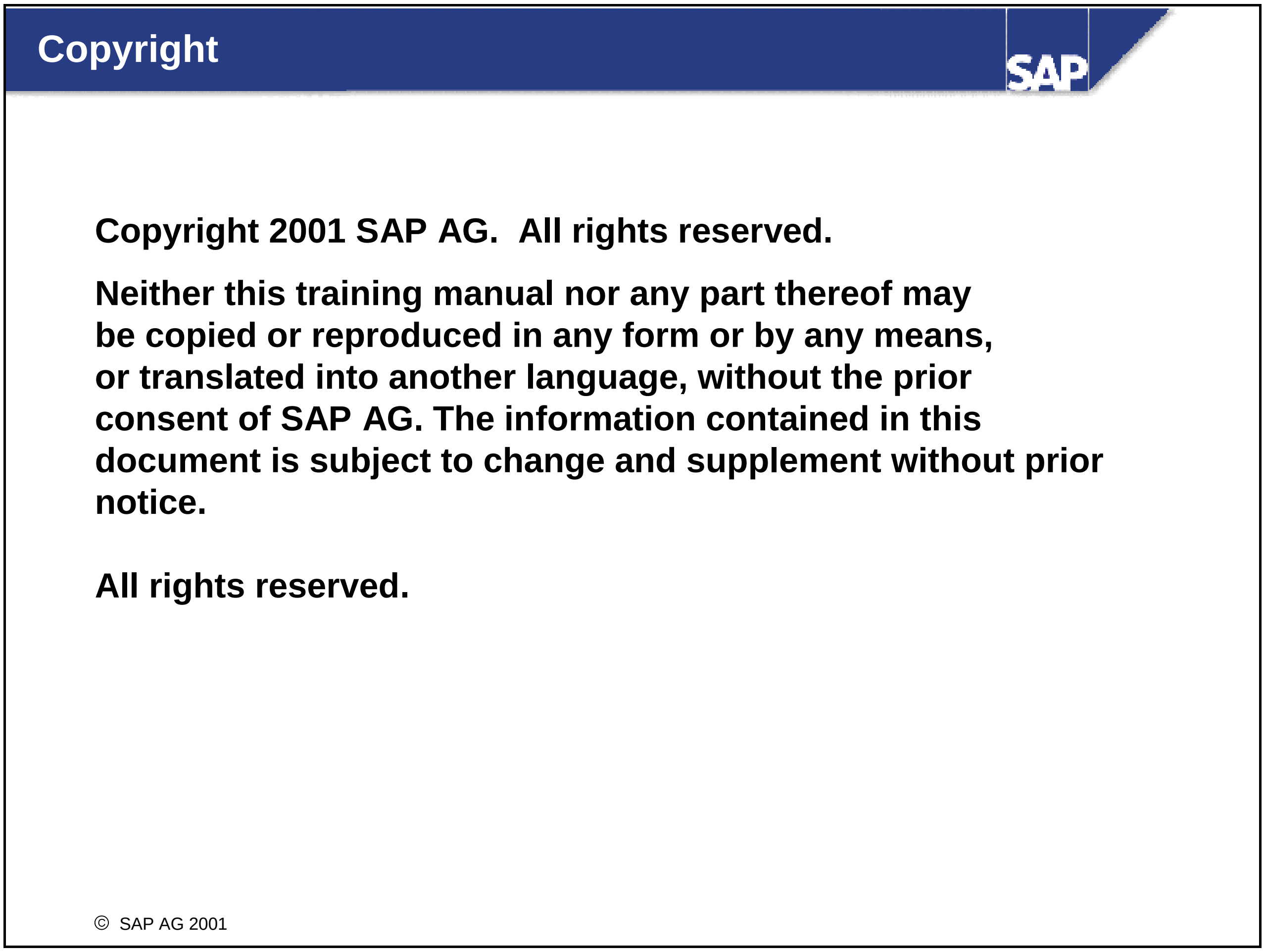
BC404

Release 46B
09.01.2003

BC404

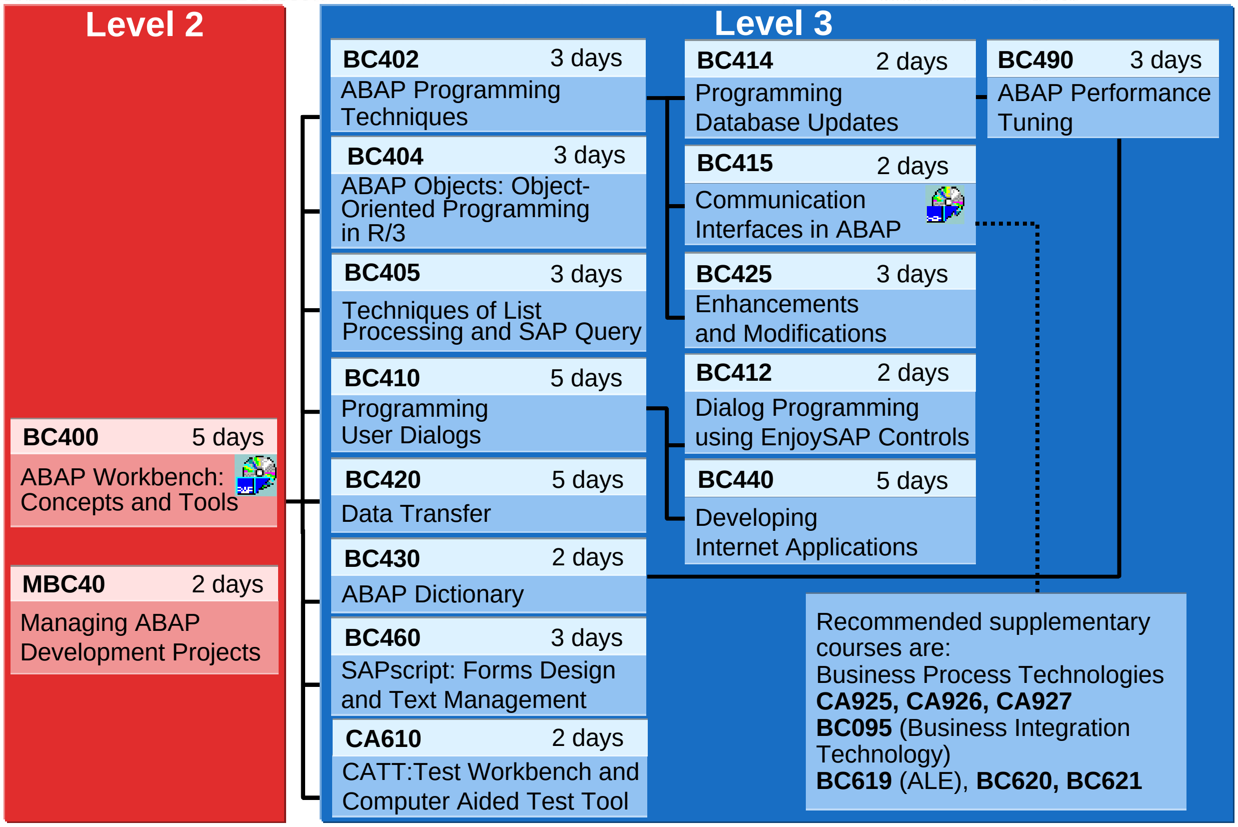
ABAP Objects: Object-Oriented Programming in R/3

R/3 System
Release 4.6A
Status: 09.99
Material no. : 50034774



Copyright

ABAP Workbench



Level 2

BC400 5 days
ABAP Workbench: Concepts and Tools

MBC40 2 days
Managing ABAP Development Projects

Level 3

BC402 3 days
ABAP Programming Techniques

BC404 3 days
ABAP Objects: Object-Oriented Programming in R/3

BC405 3 days
Techniques of List Processing and SAP Query

BC410 5 days
Programming User Dialogs

BC420 5 days
Data Transfer

BC430 2 days
ABAP Dictionary

BC460 3 days
SAPscript: Forms Design and Text Management

CA610 2 days
CATT:Test Workbench and Computer Aided Test Tool

BC414 2 days
Programming Database Updates

BC415 2 days
Communication Interfaces in ABAP

BC425 3 days
Enhancements and Modifications

BC412 2 days
Dialog Programming using EnjoySAP Controls

BC440 5 days
Developing Internet Applications

BC490 3 days
ABAP Performance Tuning

Recommended supplementary courses are:
Business Process Technologies
CA925, CA926, CA927
BC095 (Business Integration Technology)
BC619 (ALE), **BC620, BC621**

Course Prerequisites

- BC400

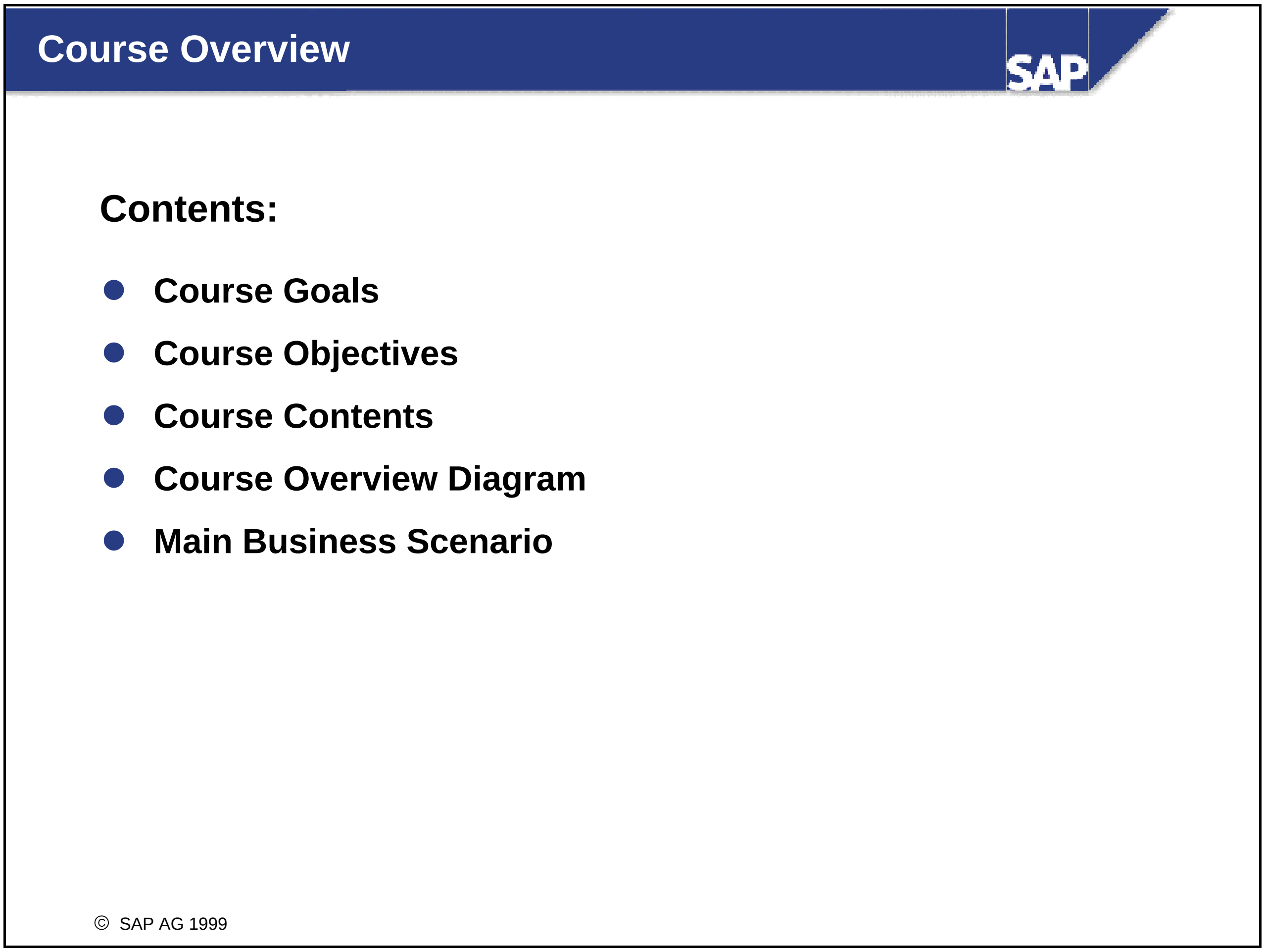
  or comparable knowledge
- Experience of programming in the R/3 environment

Target Group

- Audience:
  - IT staff
  - Project team members
- Duration: 3 days

Notes to the user

The training materials are not teach-yourself programs . They complement the course instructor’s explanations. Your material includes space for noting down this additional information.



Course Overview

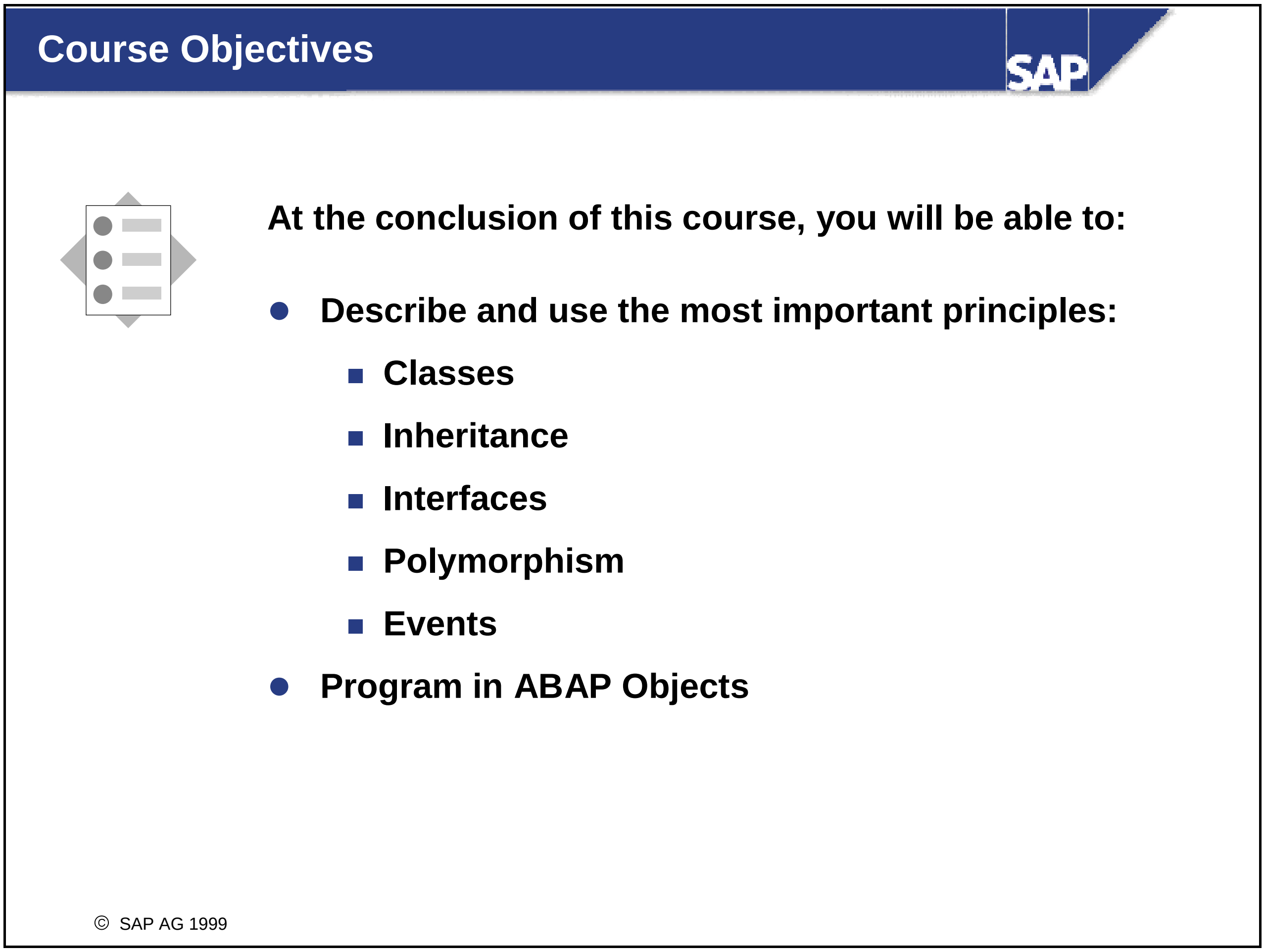
Contents:

- Course Goals
- Course Objectives
- Course Contents
- Course Overview Diagram
- Main Business Scenario

Course Goals

This course will enable you to:

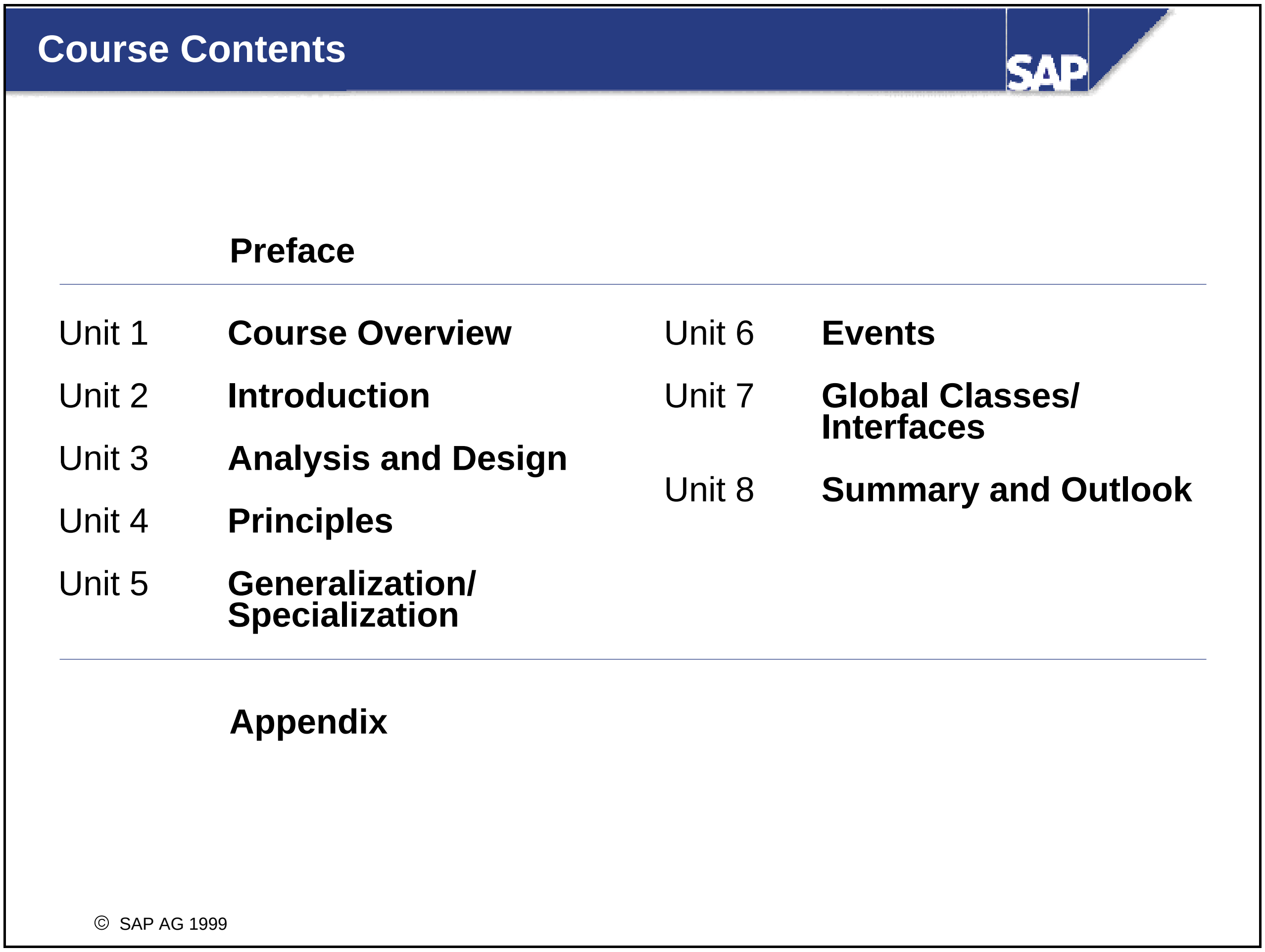
- Learn the principles of object-oriented programming
- Learn the structure and application of ABAP Objects



Course Objectives

At the conclusion of this course, you will be able to:

- Describe and use the most important principles:
  - Classes
  - Inheritance
  - Interfaces
  - Polymorphism
  - Events
- Program in ABAP Objects



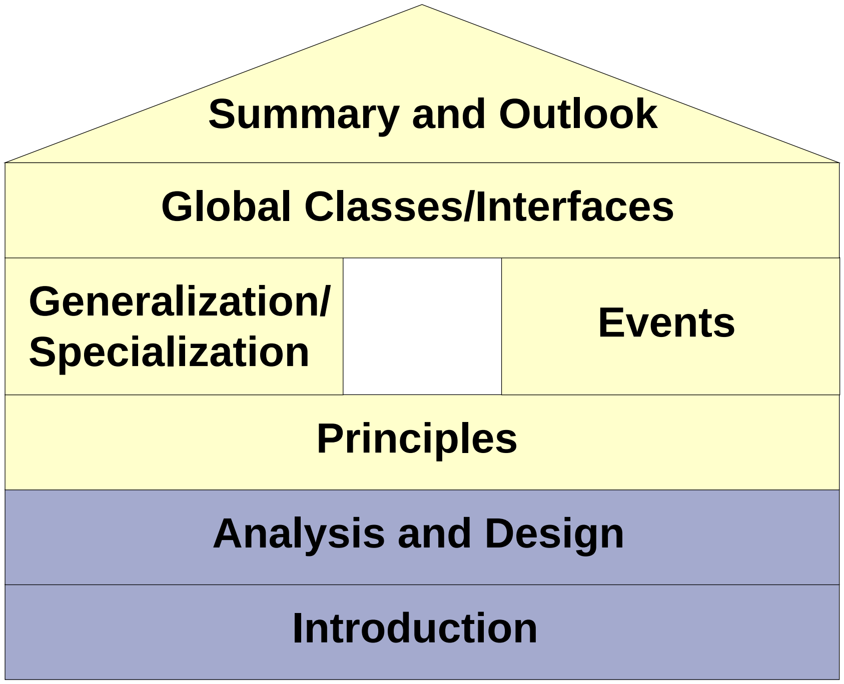
Course Contents

Preface

Unit 1	**Course Overview**
Unit 2	**Introduction**
Unit 3	**Analysis and Design**
Unit 4	**Principles**
Unit 5	**Generalization/ Specialization**
Unit 6	**Events**
Unit 7	**Global Classes/ Interfaces**
Unit 8	**Summary and Outlook**

Appendix

Course Overview Diagram



Summary and Outlook

Global Classes/Interfaces

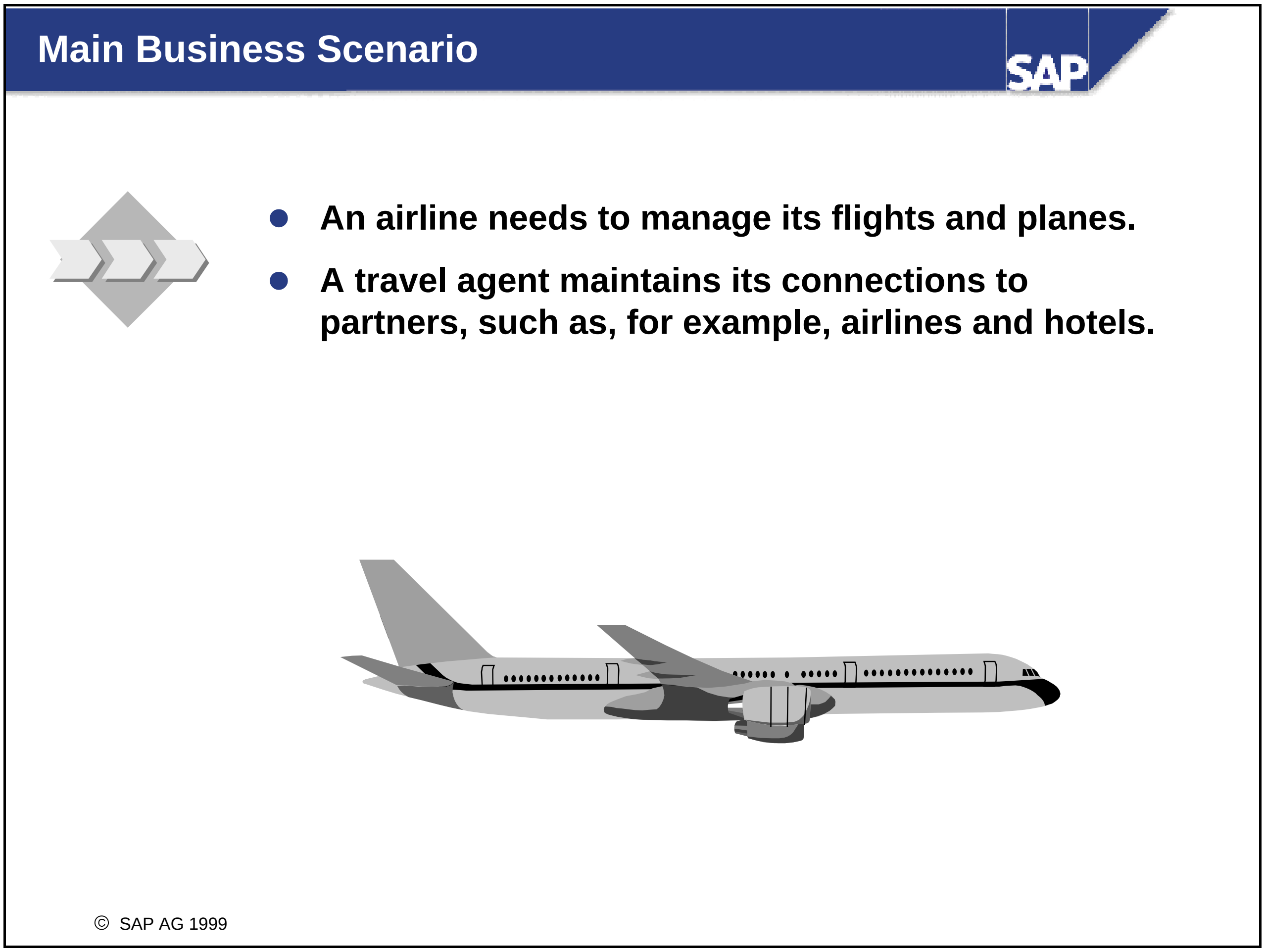
Generalization/
Specialization

Events

Principles

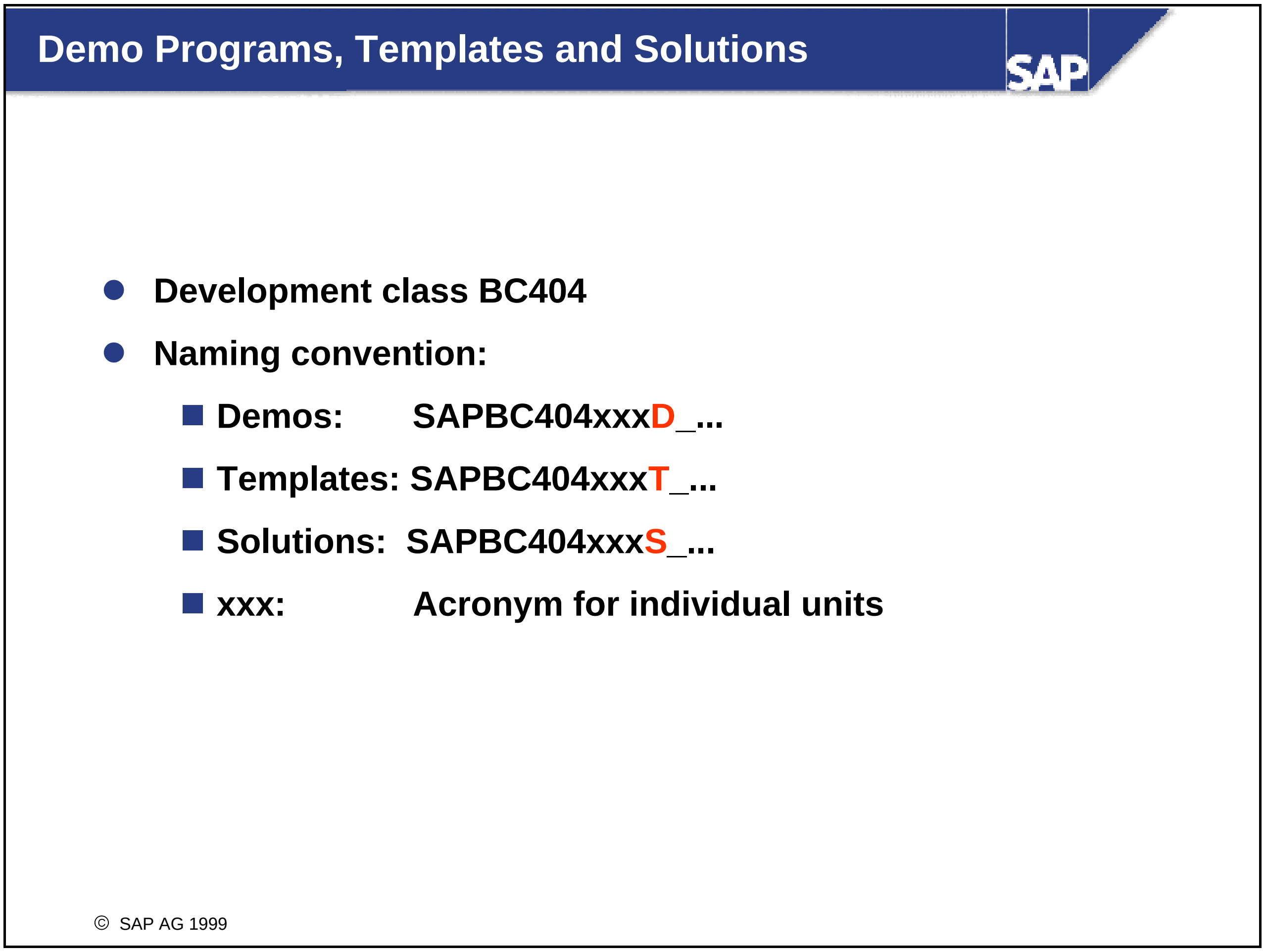
Analysis and Design

Introduction



Main Business Scenario

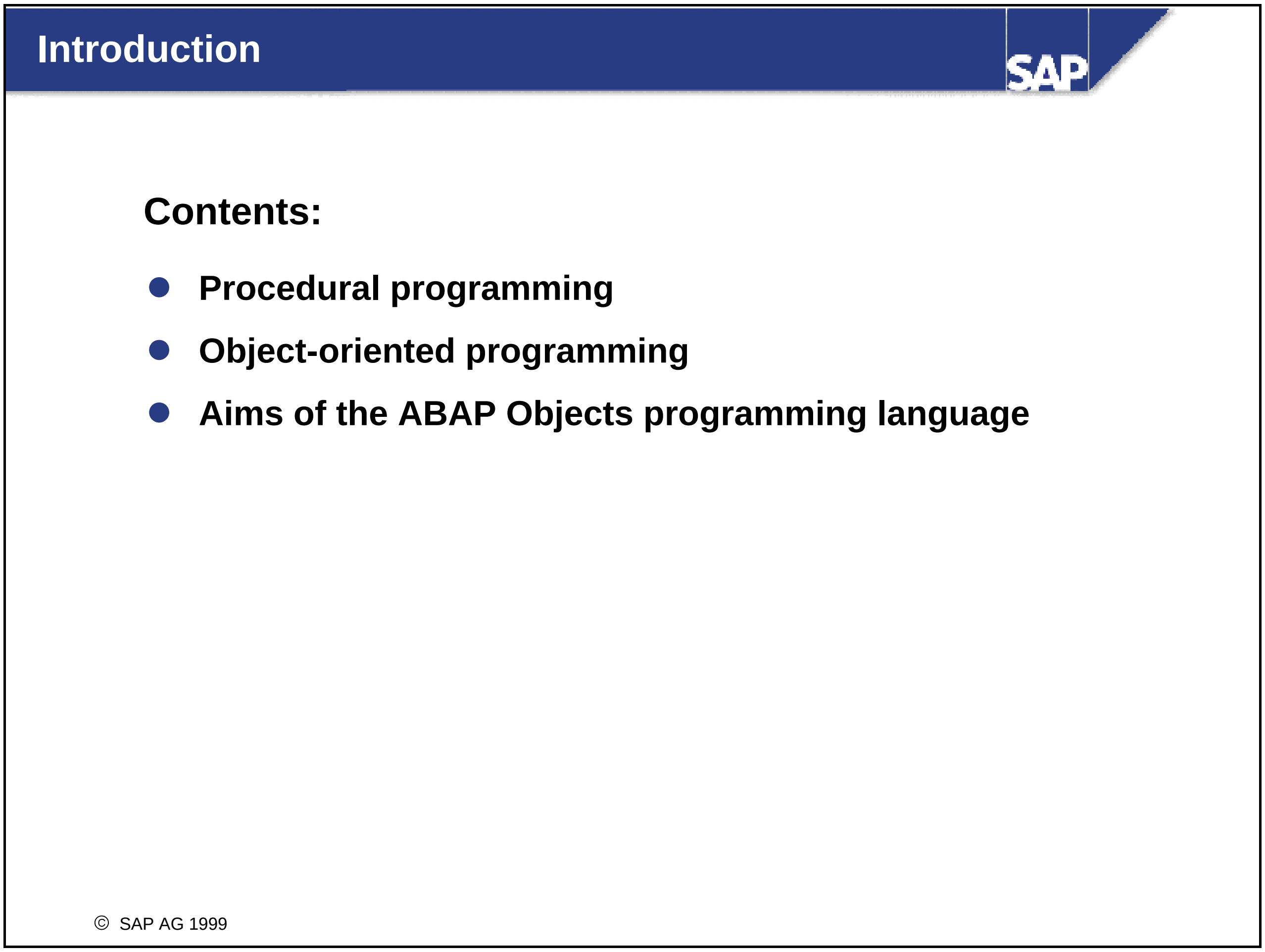
- An airline needs to manage its flights and planes.
- A travel agent maintains its connections to partners, such as, for example, airlines and hotels.



Demo Programs, Templates and Solutions

- Development class BC404
- Naming convention:
  - Demos: SAPBC404xxxD_...
  - Templates: SAPBC404xxxT_...
  - Solutions: SAPBC404xxxS_...
  - xxx: Acronym for individual units

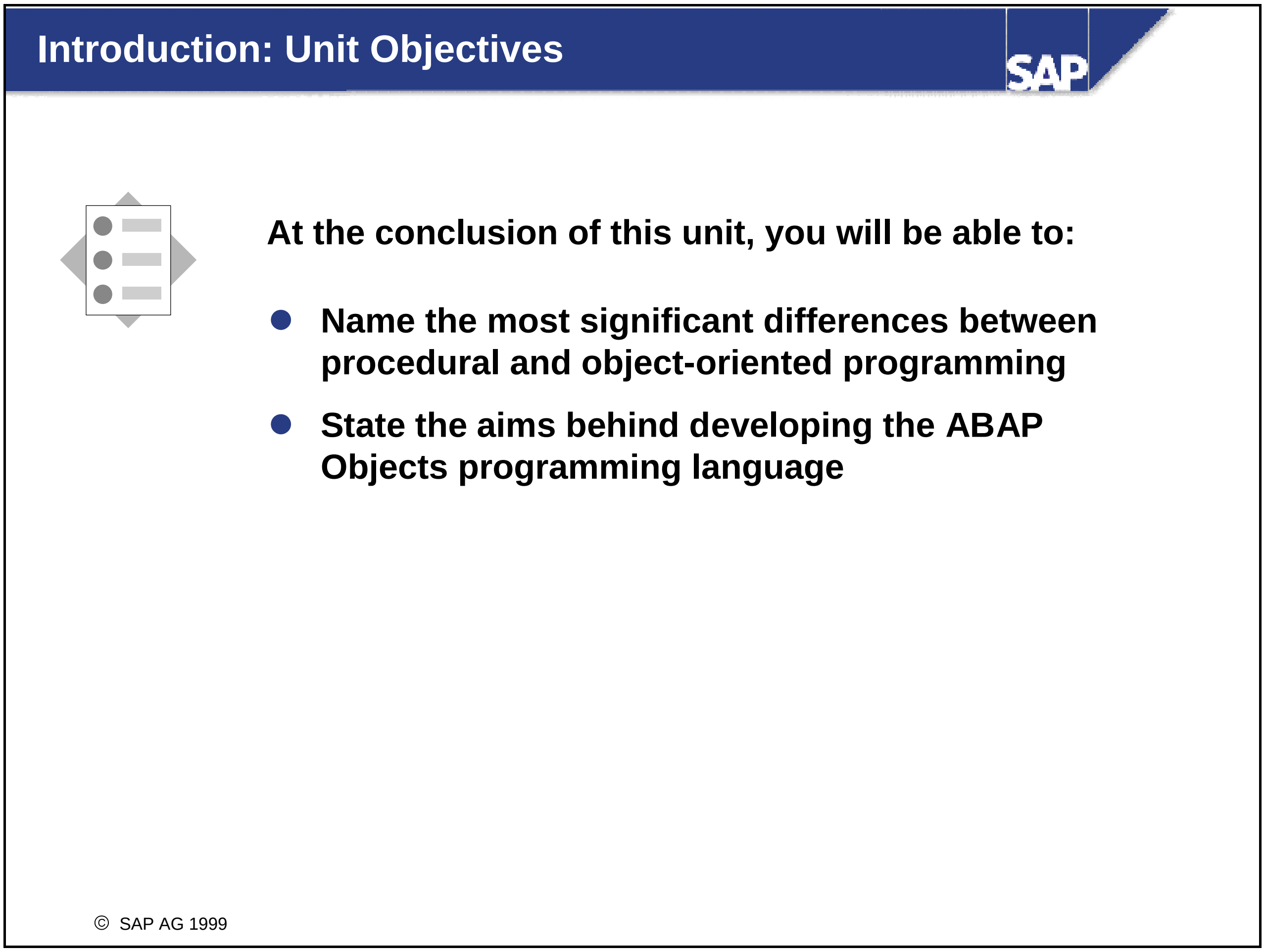
Acronyms for the individual units:
- Unit 4: BAS
- Unit 5: GEN
- Unit 6: EVE
- Unit 7: GLO



Introduction

Contents:

- Procedural programming
- Object-oriented programming
- Aims of the ABAP Objects programming language

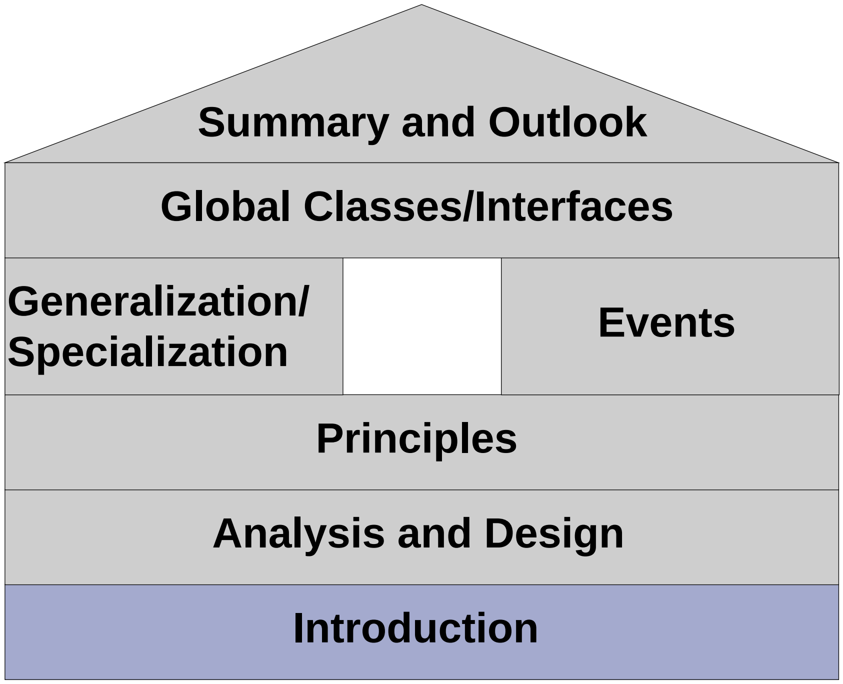


Introduction: Unit Objectives

At the conclusion of this unit, you will be able to:

- Name the most significant differences between procedural and object-oriented programming
- State the aims behind developing the ABAP Objects programming language

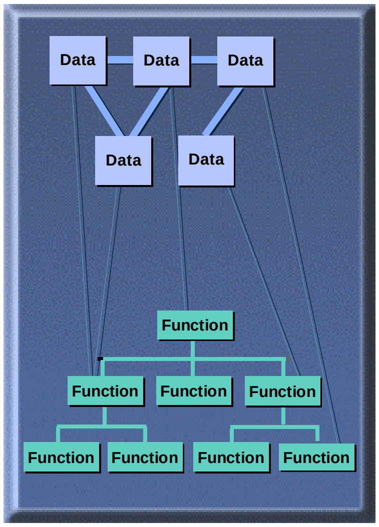
Introduction: Course Overview Diagram



Introduction: Overview (1)

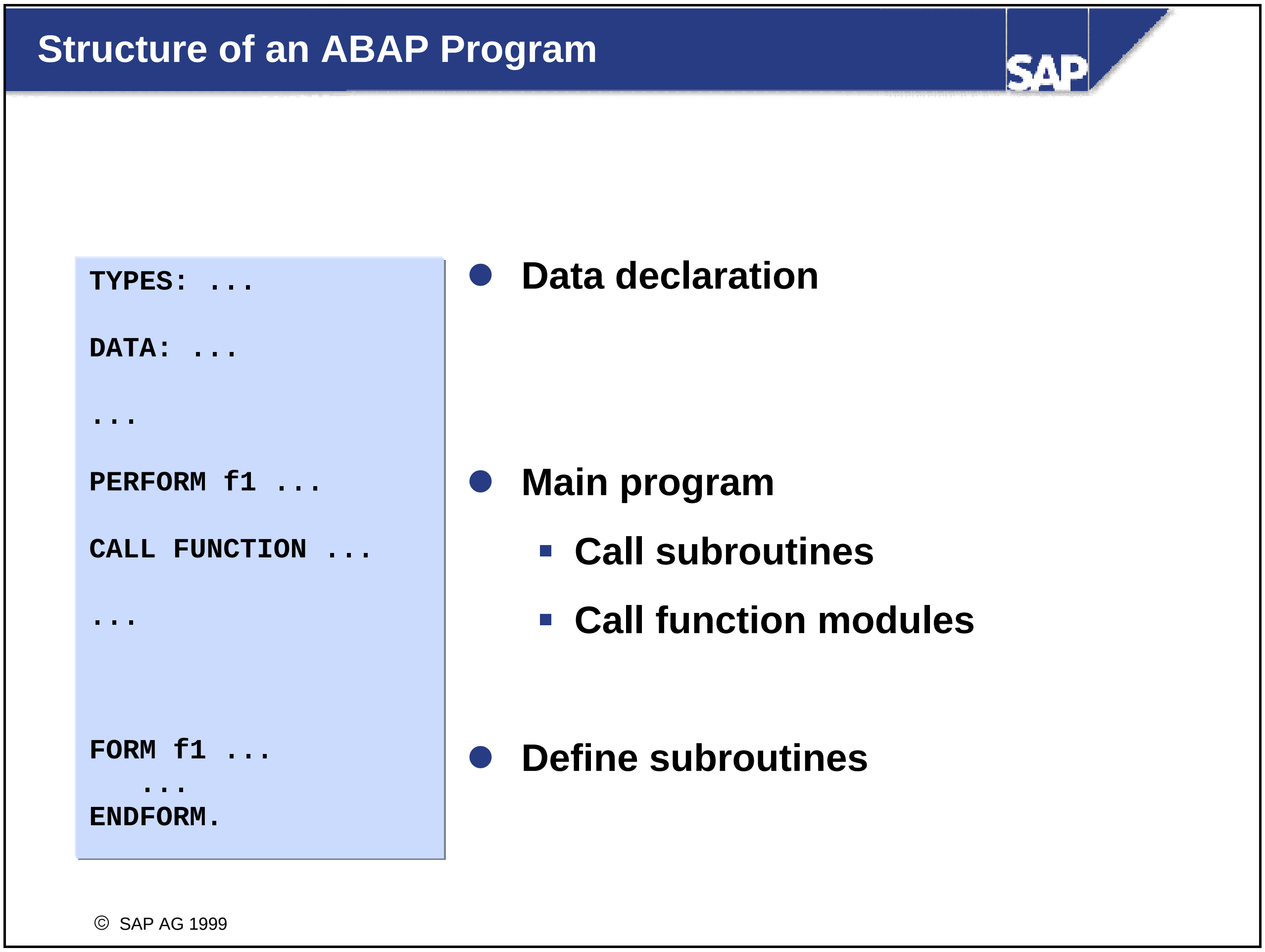
- **Procedural Programming**
- **Object-Oriented Programming**

Procedural Programming



- Functions are defined independently of data structures
- Direct access to data

Information systems used to be defined primarily by their functions: data and functions were stored separately and linked using input-output relationships.

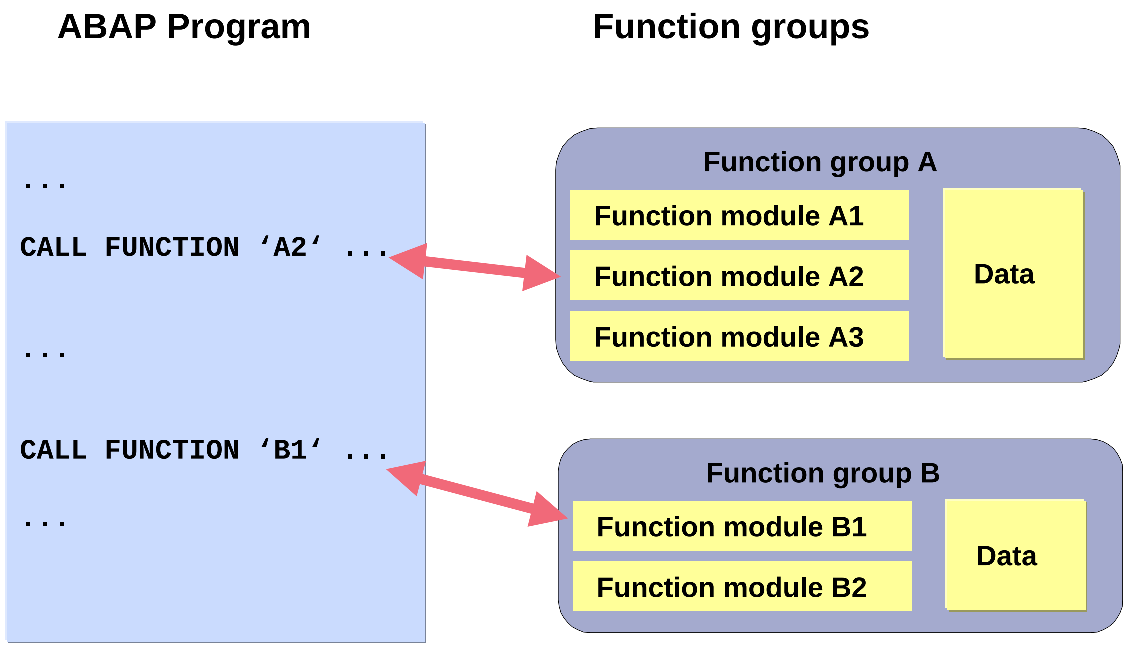


Structure of an ABAP Program

```
TYPES: ...

DATA: ...

...

PERFORM f1 ...

CALL FUNCTION ...

...



FORM f1 ...
   ...
ENDFORM.
```

- Data declaration
- Main program
  - Call subroutines
  - Call function modules
- Define subroutines

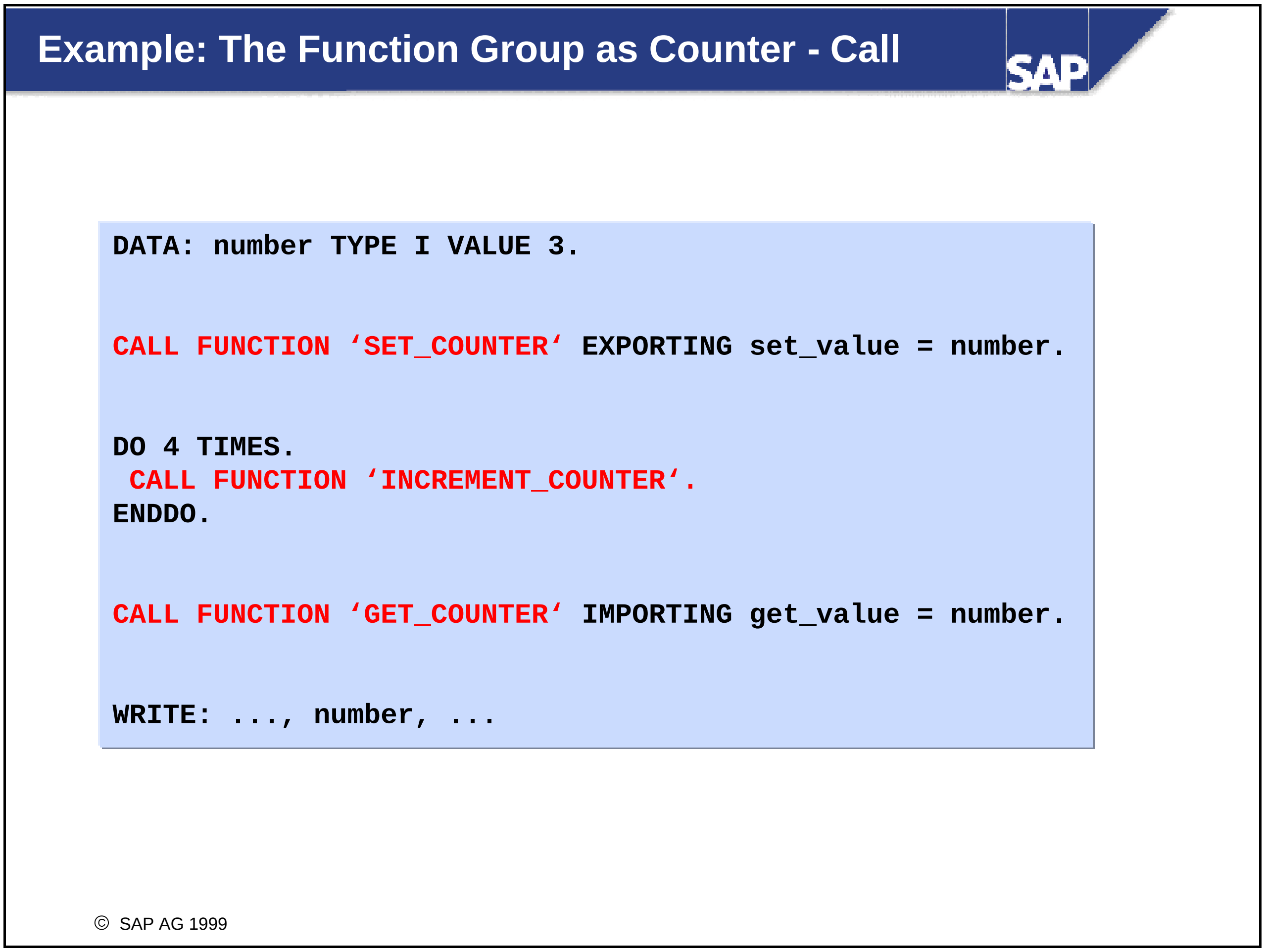
Working with Function Groups



```
...

CALL FUNCTION ‘A2‘ ...


...


CALL FUNCTION ‘B1‘ ...

...
```

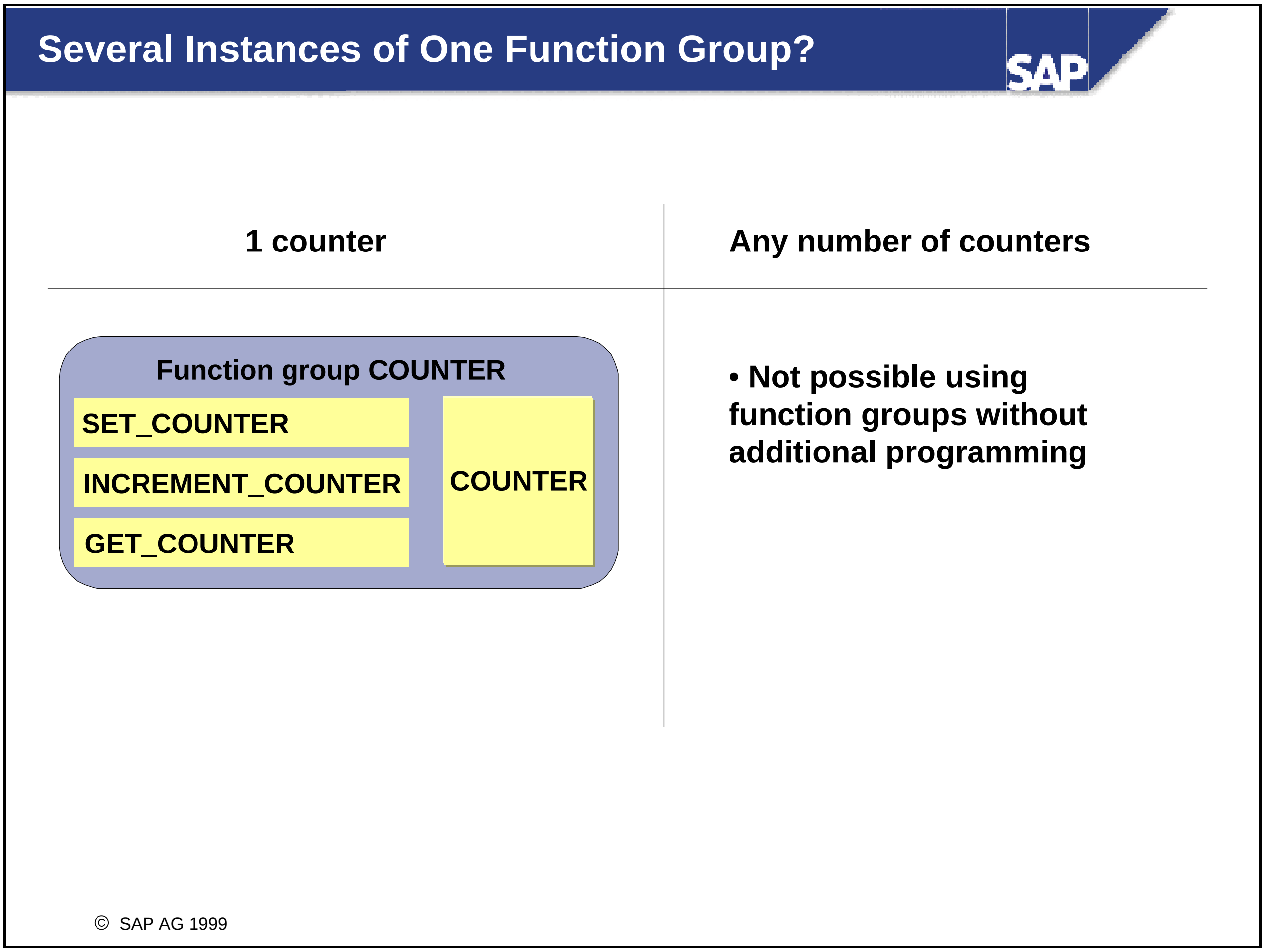
Example: The Function Group as Counter - Definition

```
FUNCTION-POOL counter.

DATA: count TYPE I.

FUNCTION SET_COUNTER.
* Local interface IMPORTING VALUE(set_value)
  count = set_value
ENDFUNCTION.

FUNCTION INCREMENT_COUNTER.
  ADD 1 TO count.
ENDFUNCTION.

FUNCTION GET_COUNTER.
* Local interface EXPORTING VALUE(get_value)
  get_value = count.
ENDFUNCTION.
```



Example: The Function Group as Counter - Call

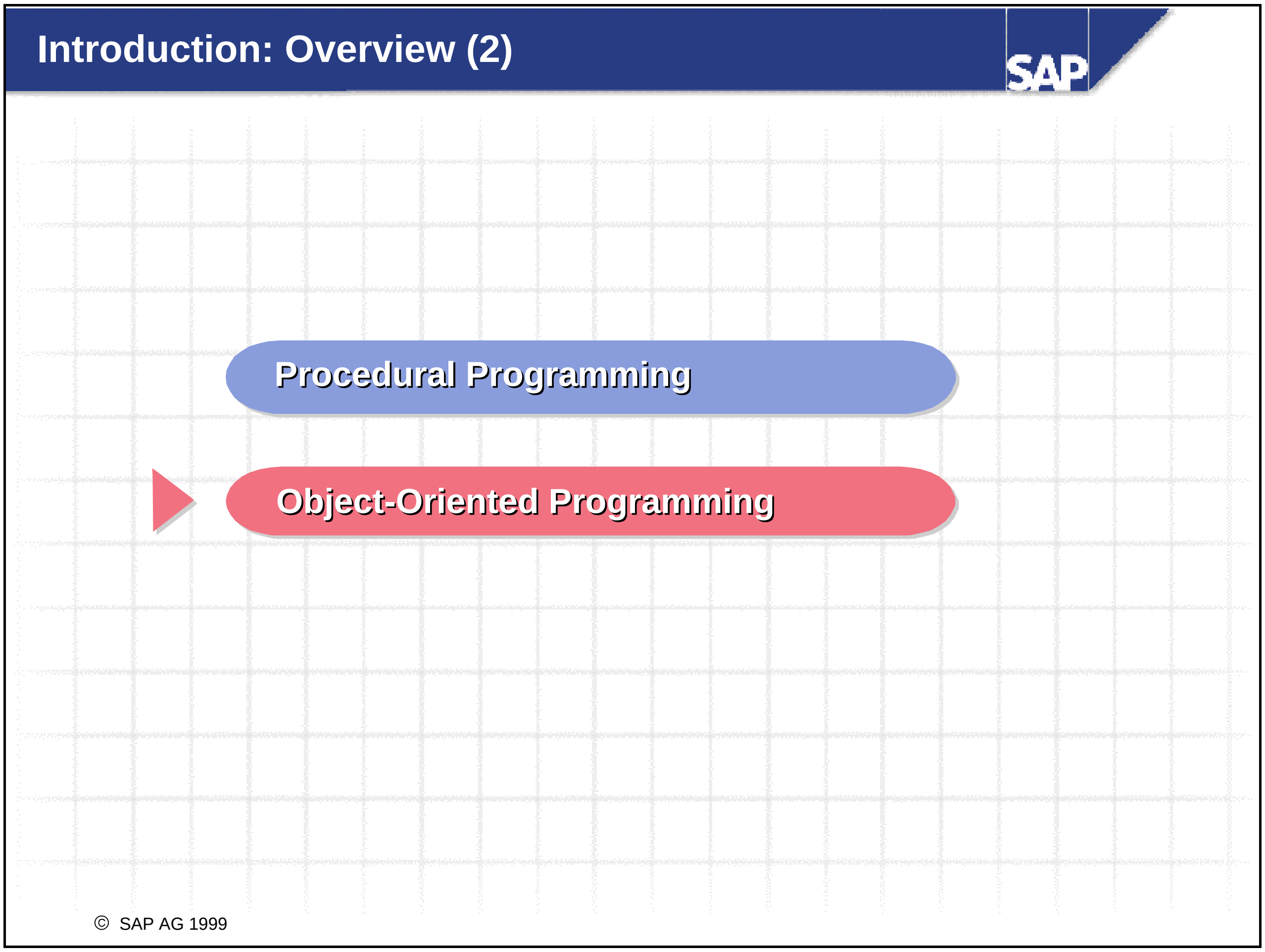
```
DATA: number TYPE I VALUE 3.


CALL FUNCTION ‘SET_COUNTER‘ EXPORTING set_value = number.


DO 4 TIMES.
 CALL FUNCTION ‘INCREMENT_COUNTER‘.
ENDDO.


CALL FUNCTION ‘GET_COUNTER‘ IMPORTING get_value = number.


WRITE: ..., number, ...
```



Several Instances of One Function Group?

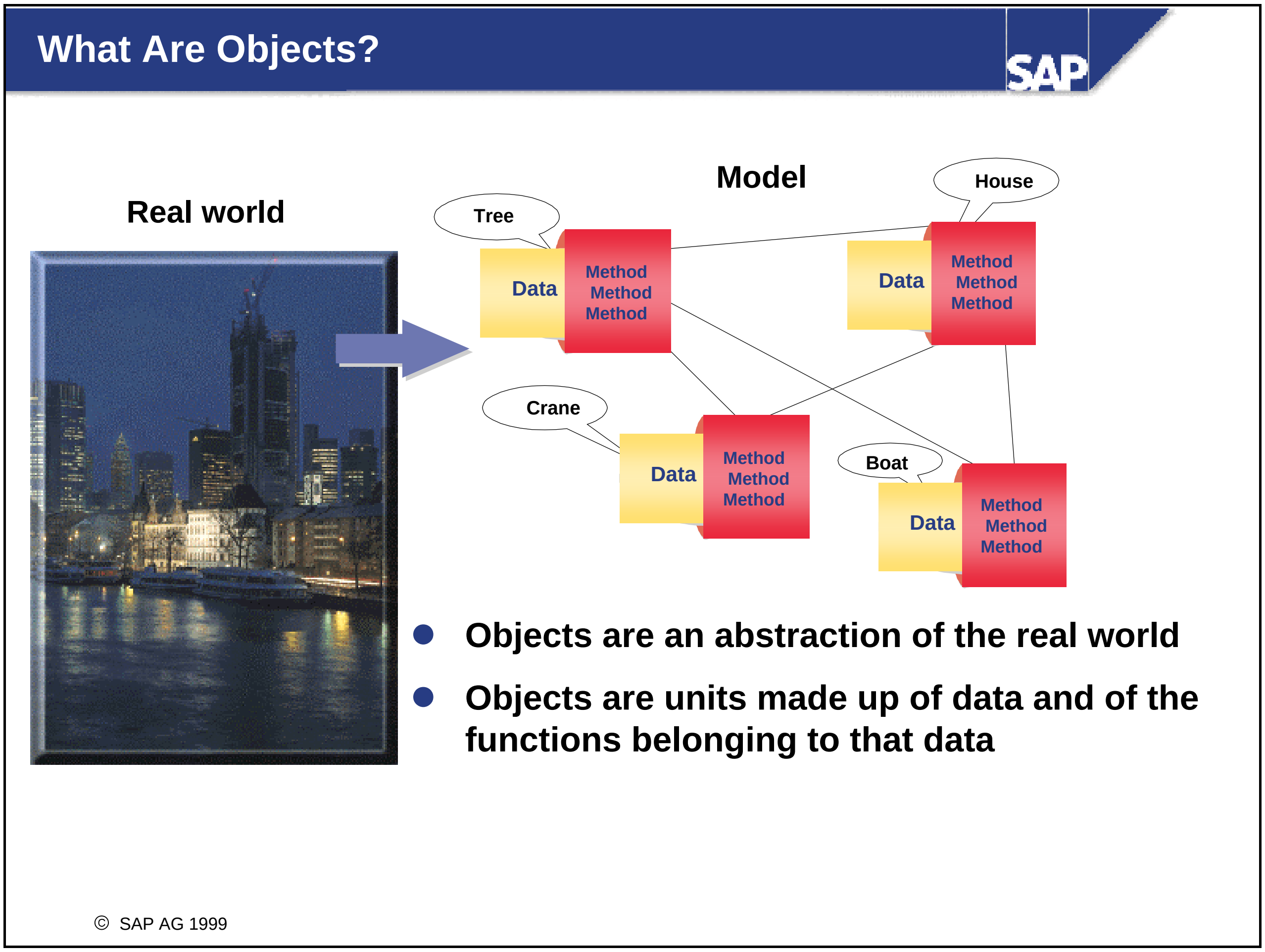
1 counter	Any number of counters
Function group COUNTER SET_COUNTER INCREMENT_COUNTER GET_COUNTER COUNTER	• Not possible using function groups without additional programming

© SAP AG 1999



Introduction: Overview (2)

- Procedural Programming
- ▶ Object-Oriented Programming



What Are Objects?

Real world

Model

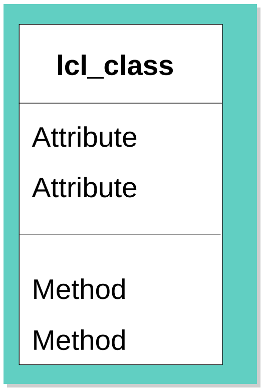
- Objects are an abstraction of the real world
- Objects are units made up of data and of the functions belonging to that data

Object orientation focuses on objects that represent either abstract or concrete things in the real world. They are first viewed in terms of their characteristics, which are mapped using the object's internal structure and attributes (data). The behavior of an object is described through methods and events (functionality).

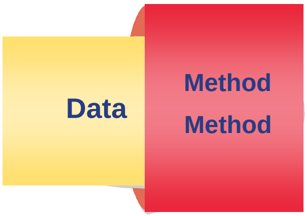
Objects form capsules containing the data itself and the behavior of that data. Objects should enable you to draft a software solution that is a one-to-one mapping of the real-life problem area.

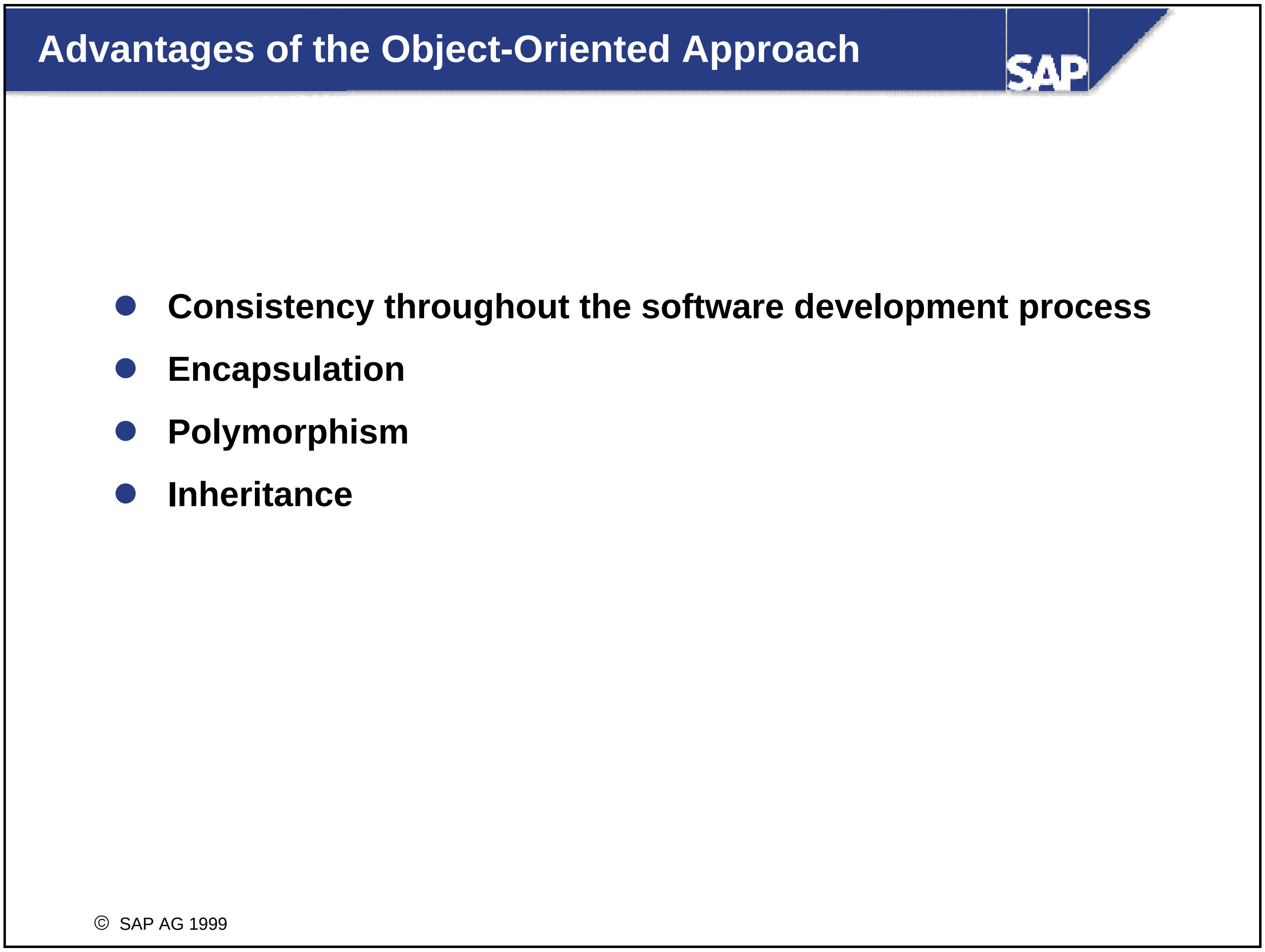
Object-Oriented Programming Model

- **Class**
  - **Gives a general description of objects (“blueprint”)**
  - **Establishes status types (attributes) and behavior (methods)**



- **Object**
  - **Reflection of real world**
  - **Specific instance of a class**





Advantages of the Object-Oriented Approach

- Consistency throughout the software development process
- Encapsulation
- Polymorphism
- Inheritance

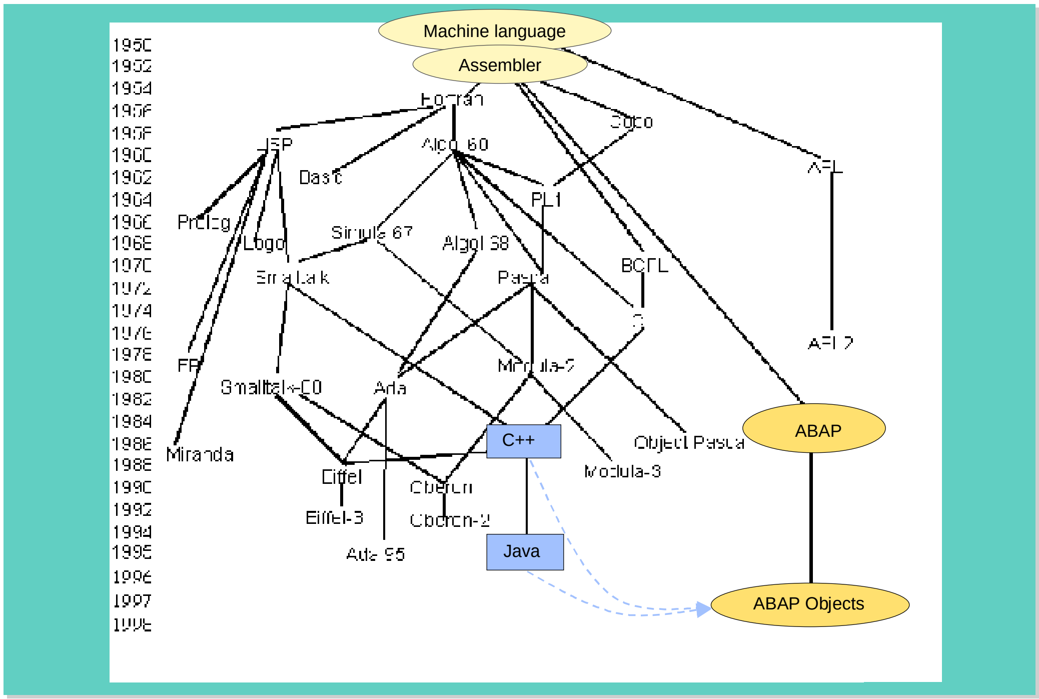
Consistency throughout the software development process
The “language” used in the various phases of software development (analysis, specification, design and implementation) is uniform. The ideal would be for changes made during the implementation phase to flow back into the design automatically.

Encapsulation
Encapsulation means that the implementation of an object is hidden from other components in the system, so that they cannot make assumptions about the internal status of the object and therefore dependencies on specific implementations do not arise.

Polymorphism
Polymorphism (ability to have multiple forms) in the context of object technology signifies that objects in different classes have different reactions to the same message.

Inheritance
Inheritance defines the implementation relationship between classes, in which one class (the subclass) shares the structure and the behavior defined in one or more other classes (superclasses).
Note: ABAP Objects only allows single inheritance.

History of Programming Languages

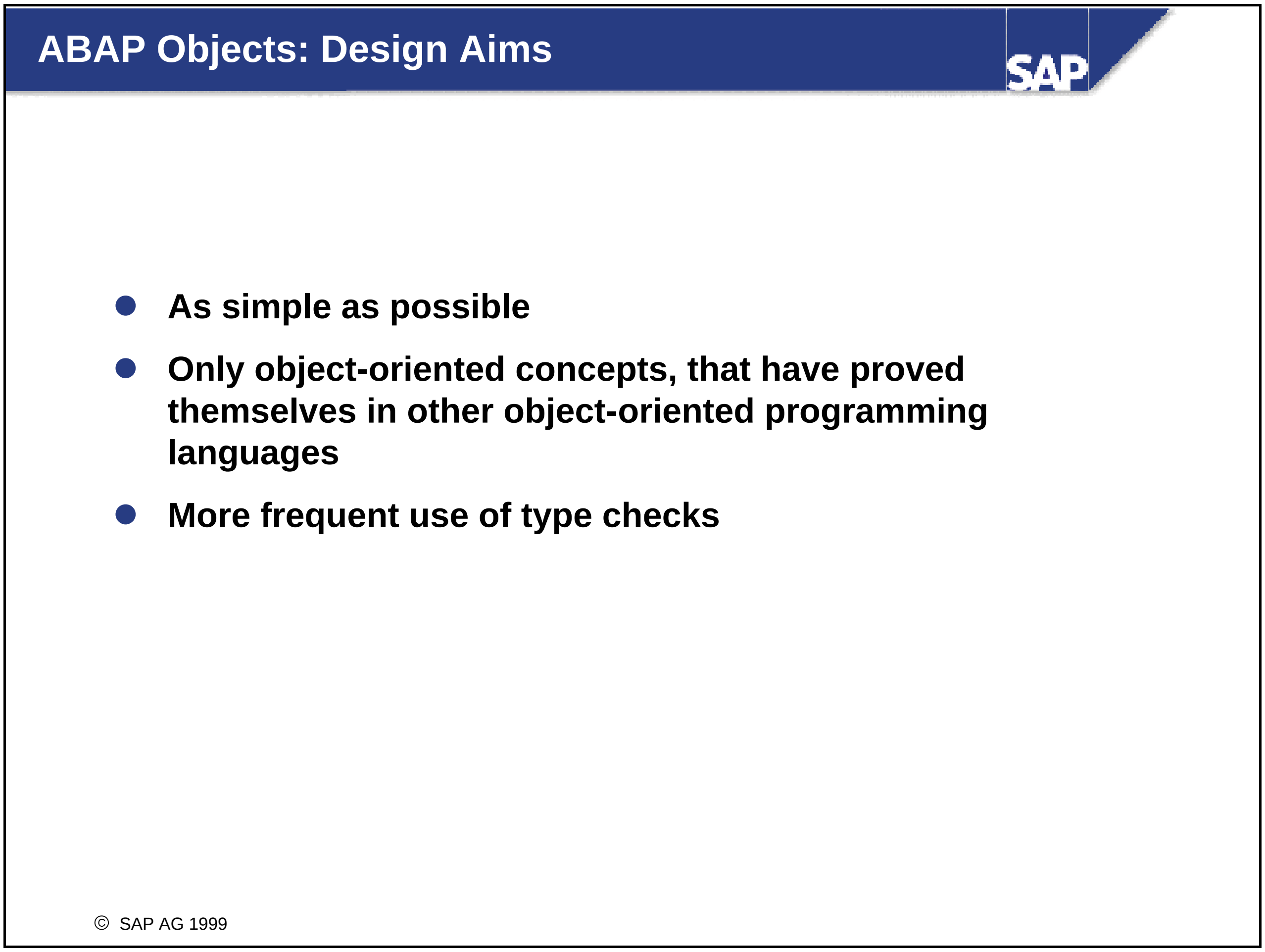


© SAP AG 1999

Before ABAP, SAP used to use a macro assembler.

ABAP was created with the intention of improving reporting. ABAP is a relatively independent in-house programming language, although it was influenced by other programming languages, for example, COBOL and PASCAL.

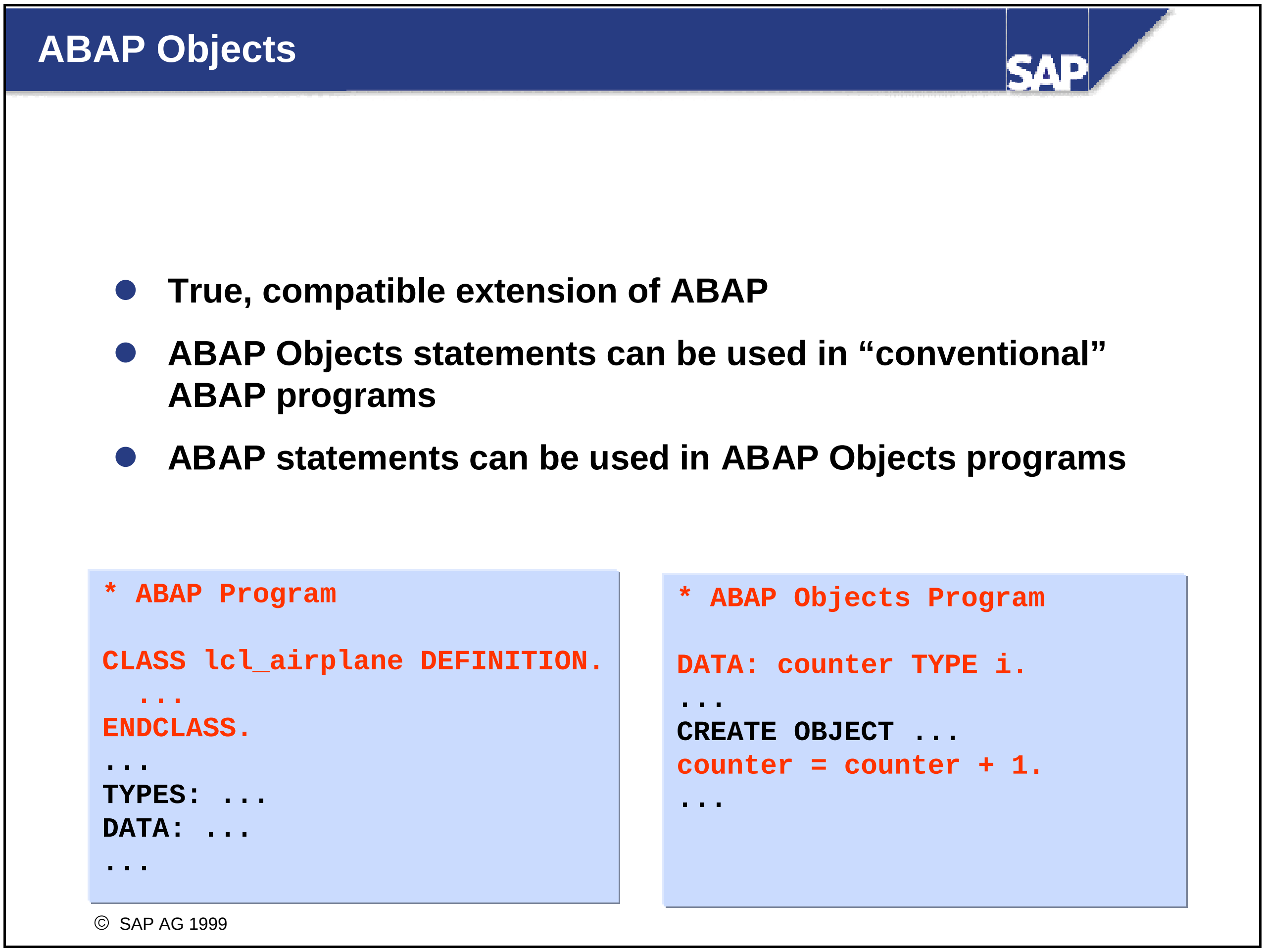
ABAP Objects is a true extension of ABAP. ABAP Objects unites the most promising aspects of other object-oriented programming languages, such as Java, C++ and Smalltalk.



ABAP Objects: Design Aims

- As simple as possible
- Only object-oriented concepts, that have proved themselves in other object-oriented programming languages
- More frequent use of type checks

You need to assign types more frequently in ABAP Objects than in ABAP. For example, in ABAP Objects, when you are defining interface parameters for methods, you must assign types to the parameters. The correct pass by value is then checked by the system when the method is called.
By comparison, in ABAP you do not need to assign types to the parameters of function modules, for example.



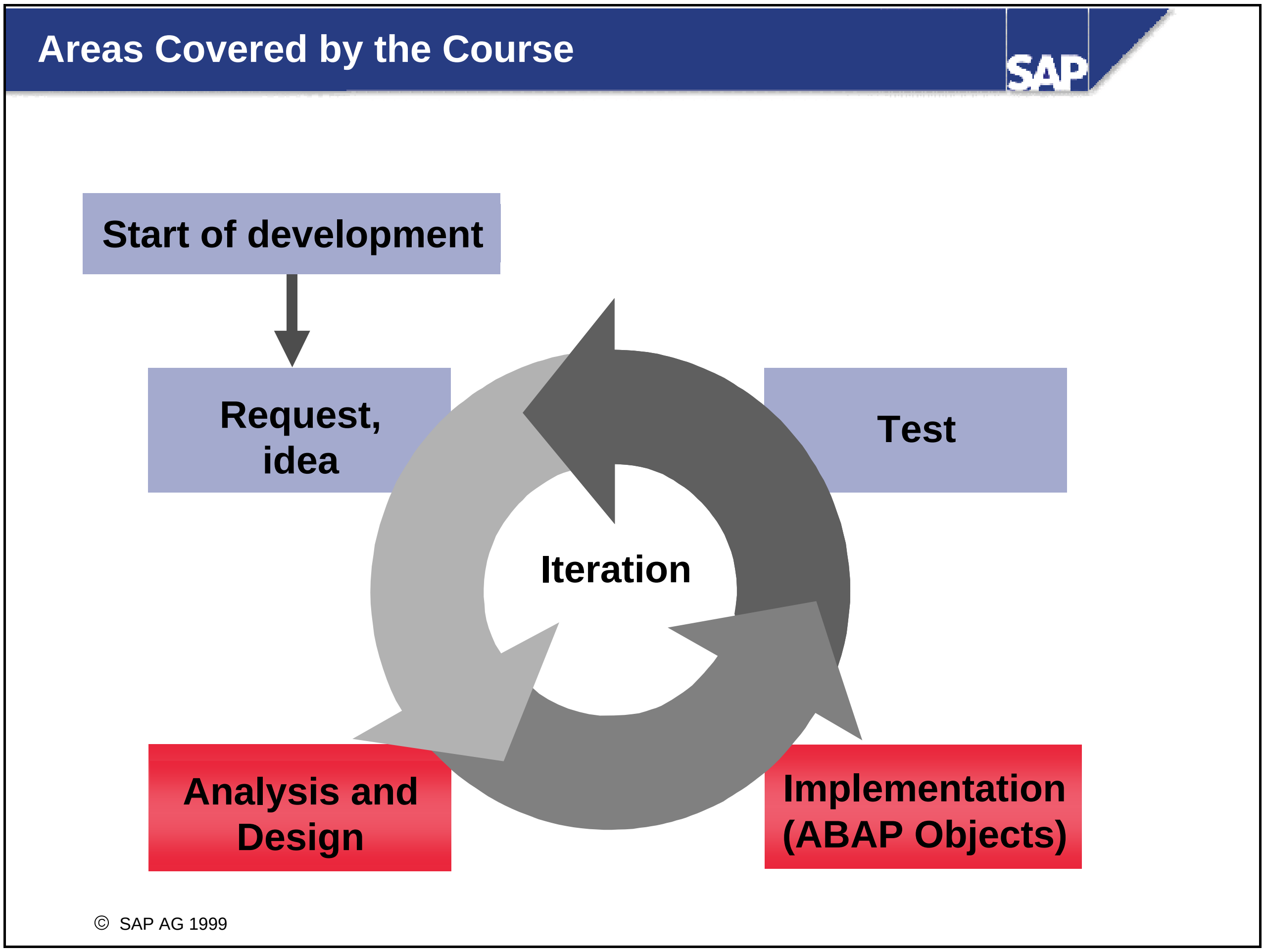
ABAP Objects

- True, compatible extension of ABAP
- ABAP Objects statements can be used in “conventional” ABAP programs
- ABAP statements can be used in ABAP Objects programs

```
* ABAP Program

CLASS lcl_airplane DEFINITION.
  ...
ENDCLASS.
...
TYPES: ...
DATA: ...
...
```

```
* ABAP Objects Program

DATA: counter TYPE i.
...
CREATE OBJECT ...
counter = counter + 1.
...
```

ABAP Objects is not a new language, but has been developed as an extension of ABAP. It integrates seamlessly into ABAP syntax and the ABAP programming model. All enhancements are strictly upward compatible.



Areas Covered by the Course

Start of development

Request, idea

Test

Iteration

Analysis and Design

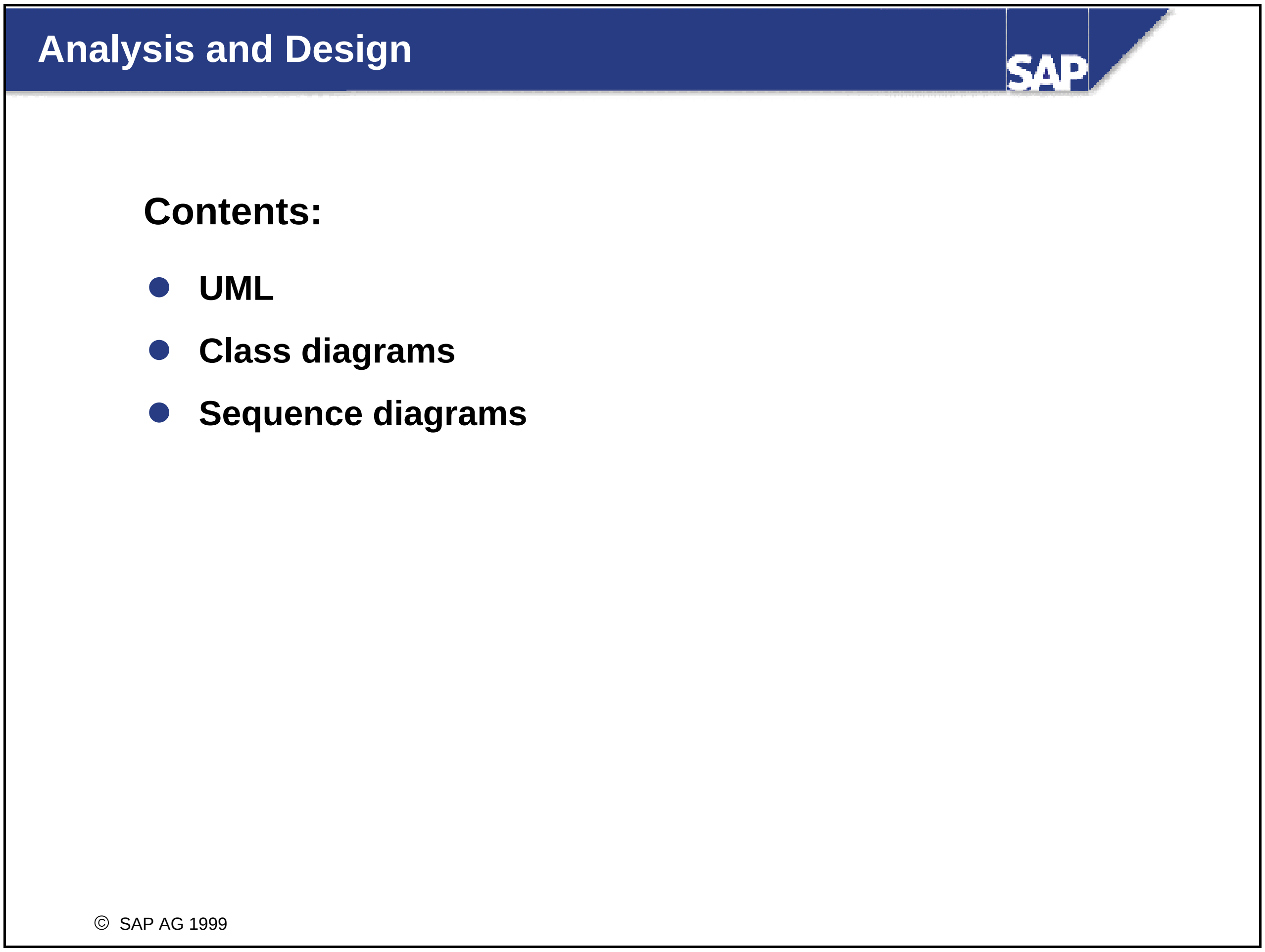
Implementation (ABAP Objects)

In object-oriented programming, the analysis and design phase is even more important than it is for procedural programming. The reason for this is that in object-oriented programming, decisions taken during the analysis and design phase have even more pronounced effects on implementation than they do in procedural programming.

Introduction: Summary

You are now able to:

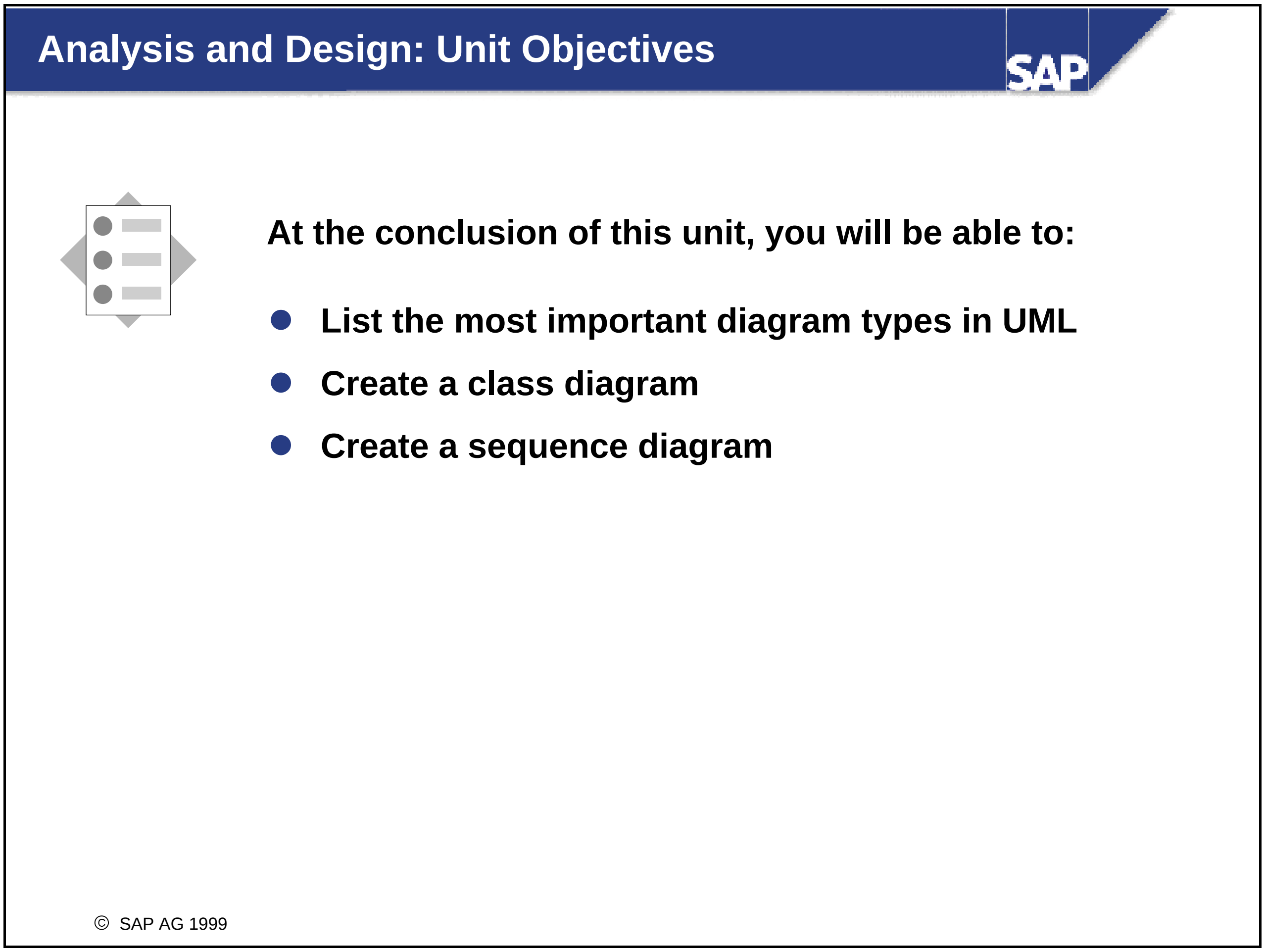
- Name the most significant differences between procedural and object-oriented programming
- State the aims behind developing the ABAP Objects programming language



Analysis and Design

Contents:

- UML
- Class diagrams
- Sequence diagrams

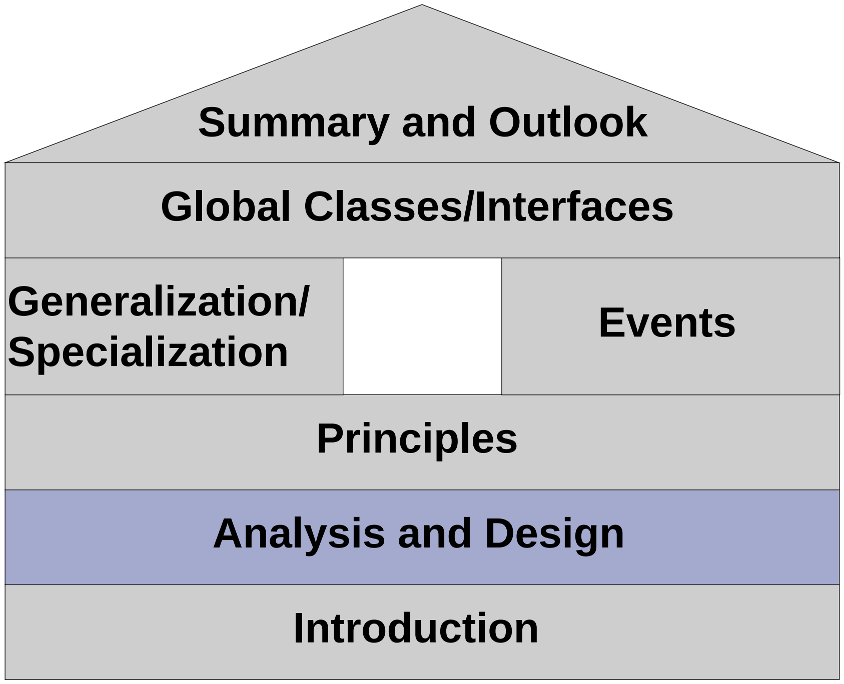


Analysis and Design: Unit Objectives

At the conclusion of this unit, you will be able to:

- List the most important diagram types in UML
- Create a class diagram
- Create a sequence diagram

Analysis and Design: Course Overview Diagram



Summary and Outlook

Global Classes/Interfaces

Generalization/ Specialization

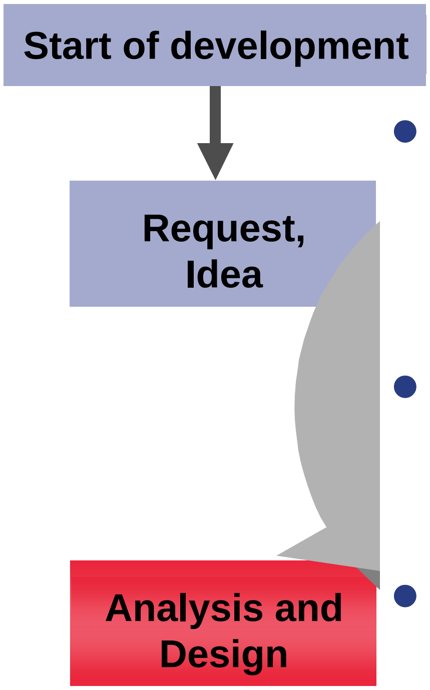
Events

Principles

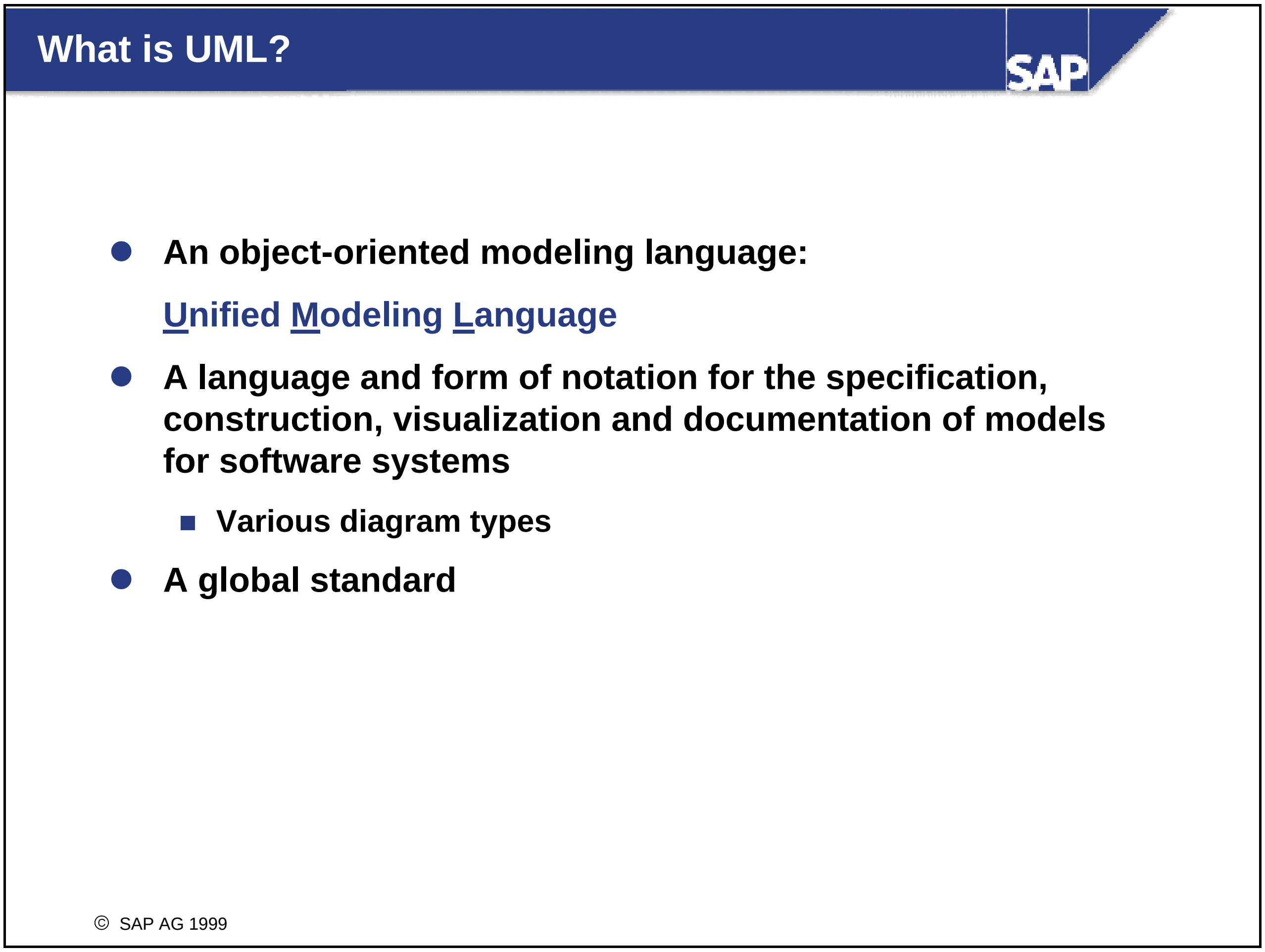
Analysis and Design

Introduction

Overview



- Analysis
  - Develop a model
  - Question: what needs to be done?
- Design
  - Establish implementation
  - Question: how should you do it?
- Standardized language for description
  - UML



What is UML?

- An object-oriented modeling language:

  Unified Modeling Language

- A language and form of notation for the specification, construction, visualization and documentation of models for software systems
  - Various diagram types
- A global standard

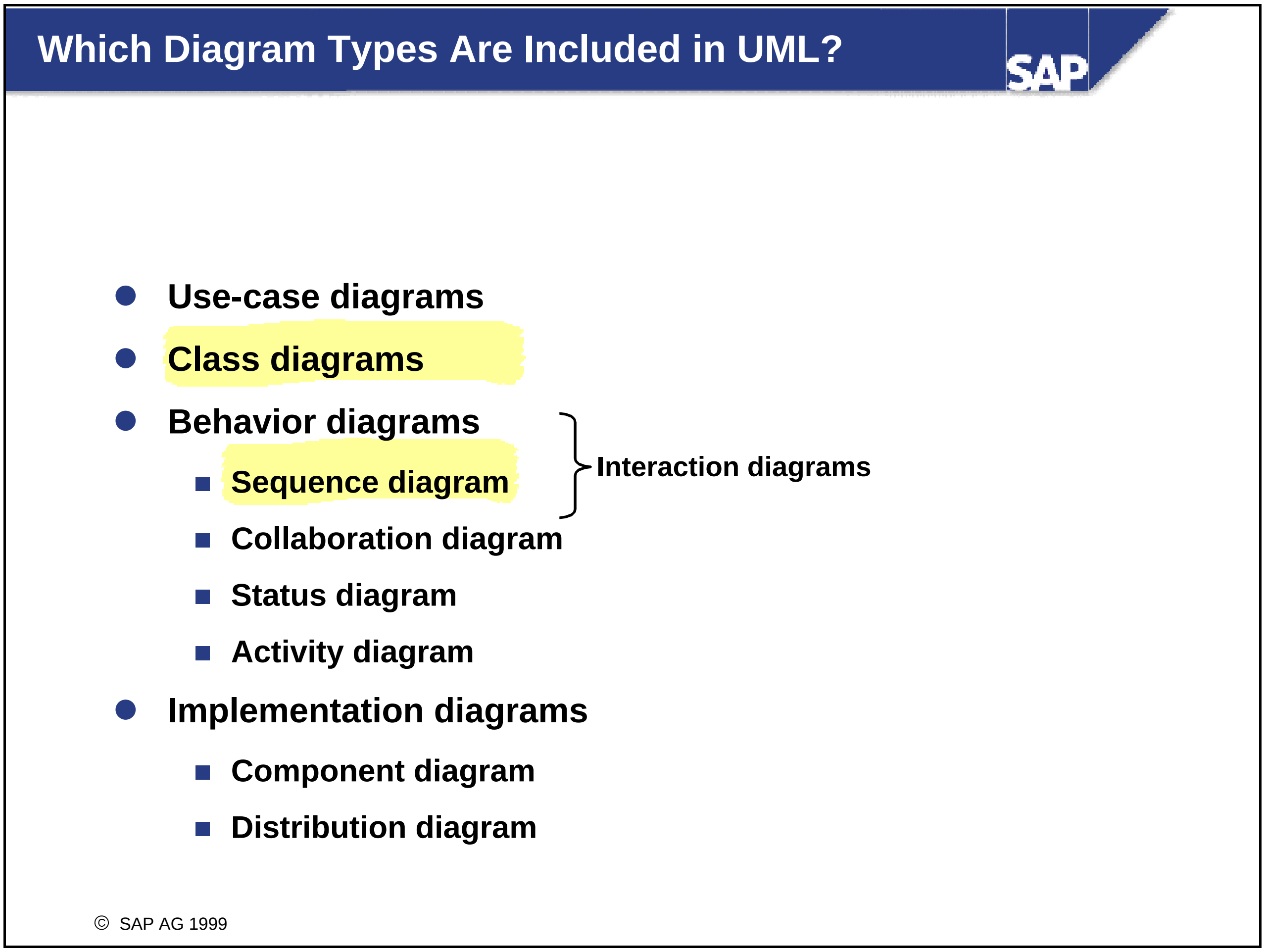
UML (Unified Modeling Language) is a standardized modeling language. It is used for the specification, construction, visualization and documentation of models for software systems and enables uniform communication between various users.

UML does not describe the steps in the object-oriented development process.

UML is an industry standard and has been standardized by the OMG (Object Management Group) since September 1997 as UML Version 1.1. The members of the OMG are continuously developing it further.

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



Which Diagram Types Are Included in UML?

- Use-case diagrams
- Class diagrams
- Behavior diagrams
  - Sequence diagram
  - Collaboration diagram
  - Status diagram
  - Activity diagram
- Implementation diagrams
  - Component diagram
  - Distribution diagram

Interaction diagrams

UML describes a number of different diagram types in order to represent different views of a system.

Use-case diagrams show the relationships between agents and actions (use cases), that is, they represent external system behavior from the user's point of view.

Class diagrams show the static view of a model.

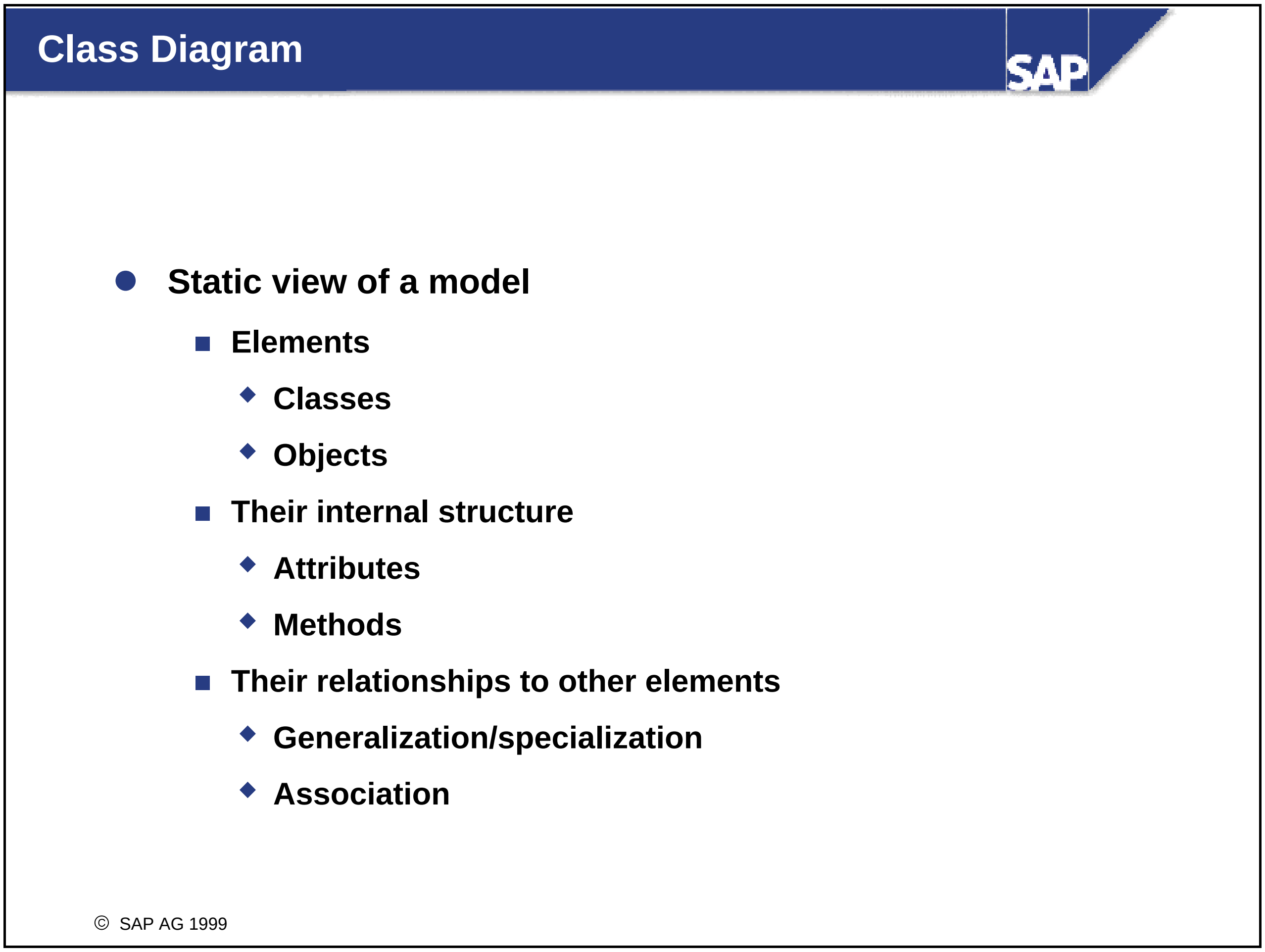
Interaction diagrams demonstrate the relationships and method calls between objects.
Sequence diagrams emphasize the timing sequence of the method calls, while collaboration diagrams focus more on the object relationships and their topology.

Status diagrams show a sequence of statuses that an object can adopt during its lifetime, and the stimuli that cause this status to change.

Activity diagrams are a special type of status diagrams. They mostly or exclusively contain activities.

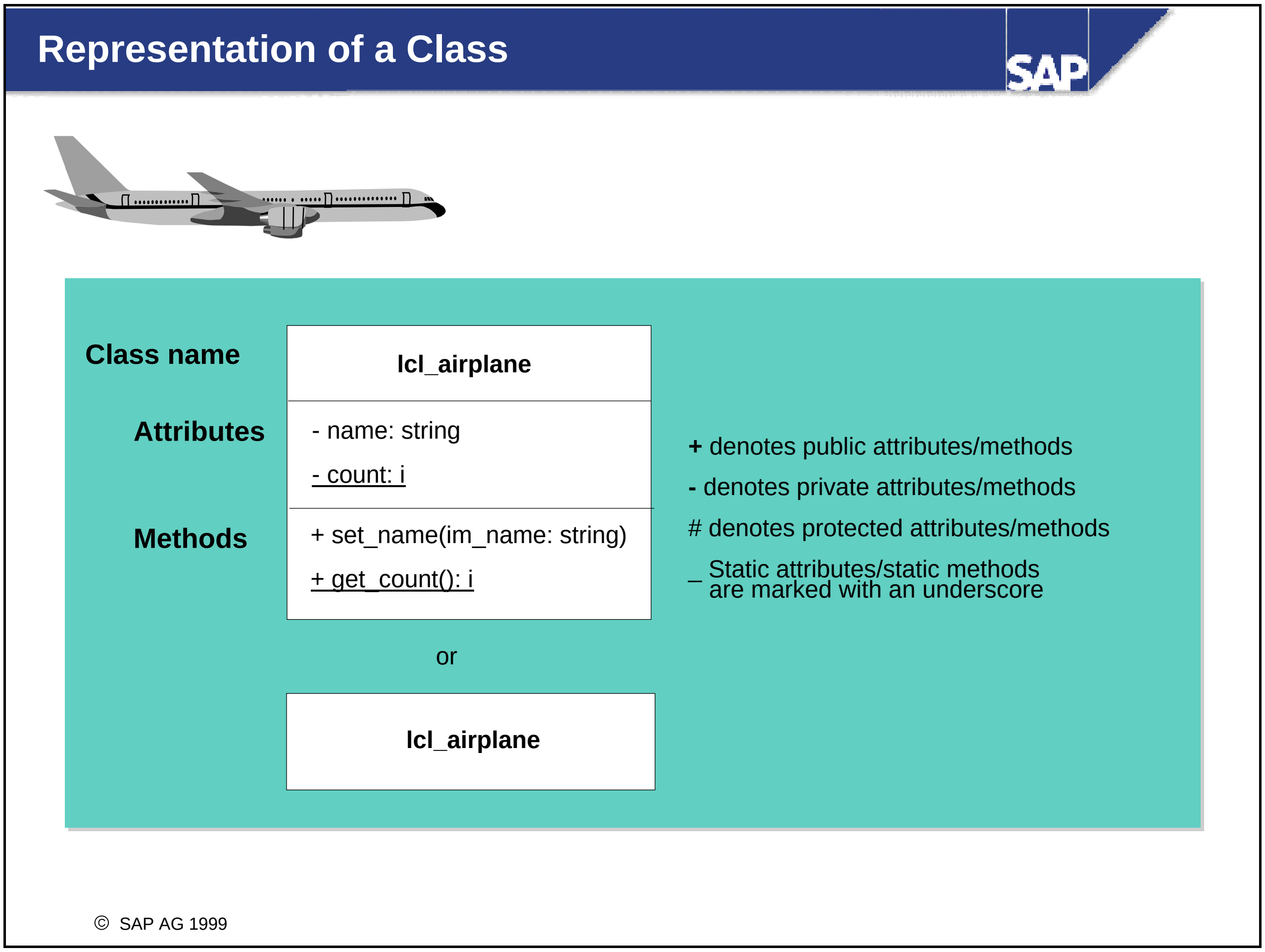
Component diagrams show the organization and dependencies of components.

Distribution diagrams represent the dependencies of software and hardware.



Class Diagram

- Static view of a model
  - Elements
    - Classes
    - Objects
  - Their internal structure
    - Attributes
    - Methods
  - Their relationships to other elements
    - Generalization/specialization
    - Association



Representation of a Class

Class name

Attributes

Methods

lcl_airplane

- name: string

- count: i

+ set_name(im_name: string)

+ get_count(): i

or

lcl_airplane

+ denotes public attributes/methods

- denotes private attributes/methods

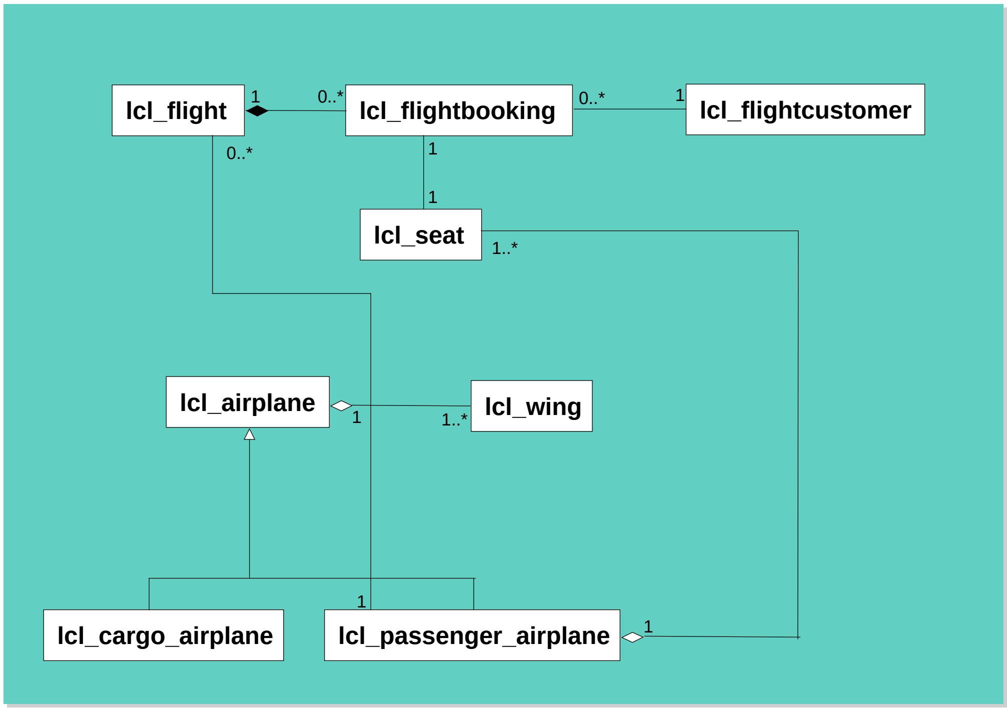
# denotes protected attributes/methods

_ Static attributes/static methods are marked with an underscore

UML notation:
The slide depicts two ways of representing classes. In the first, the class is represented by its name, attributes and methods, in the second, the name only is used. UML also offers you the option of omitting the either the attribute or the method part.

ABAP Objects events are not represented in class diagrams.

Example of a Class Diagram



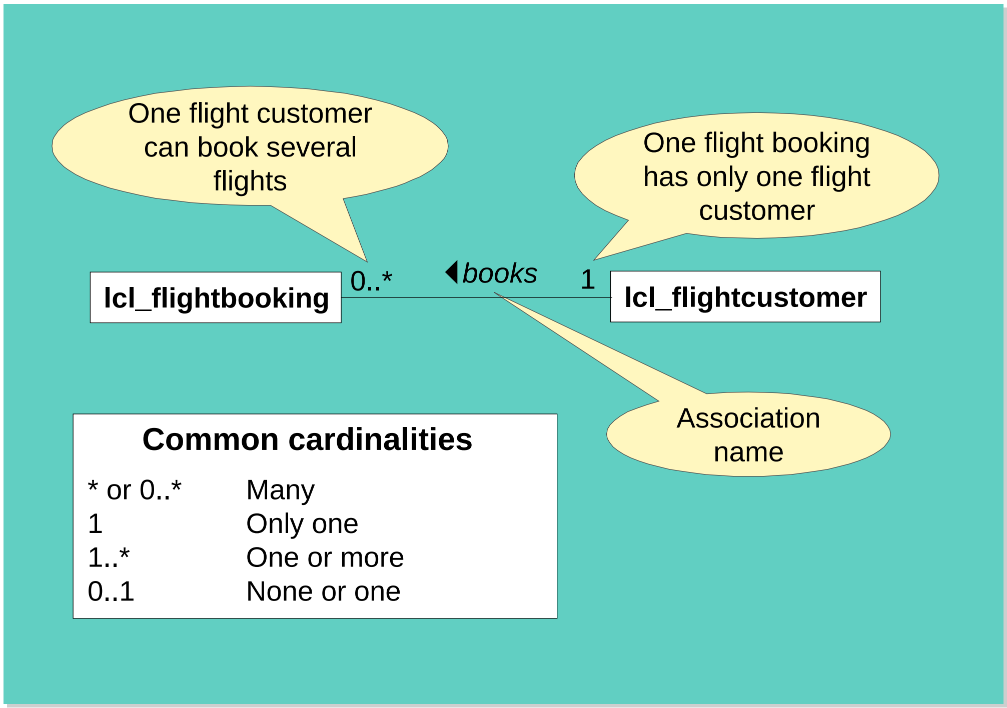
A class diagram describes the elements contained in the model and their various static relationships. There are two basic forms of static relationships:

Associations (for example, a flight customer books a flight)

Generalization/specialization (for example a cargo plane and a passenger plane are planes)

Classes can also be shown with their attributes and methods in the class diagrams.

Association


One flight customer can book several flights
One flight booking has only one flight customer
lcl_flightbooking
0..*
books
1
lcl_flightcustomer
Association name

Common cardinalities

* or 0..*	Many
1	Only one
1..*	One or more
0..1	None or one

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 an association.

An association is usually a relationship between different classes. 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 links 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 (flight booking->customer, customer -> flight booking). Roles can have names (for example, the association flight->flight booking could be called reservations).

Each role has a cardinality that shows how many instances 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.

UML notation:

An association is represented by a line between the class symbols.

The cardinality of the relationship can be shown at each end of the line.

Associations can be given a name for ease of identification (a verb or a short text). This name is written in italics above the line and may have a arrow to show the direction. Both are optional.

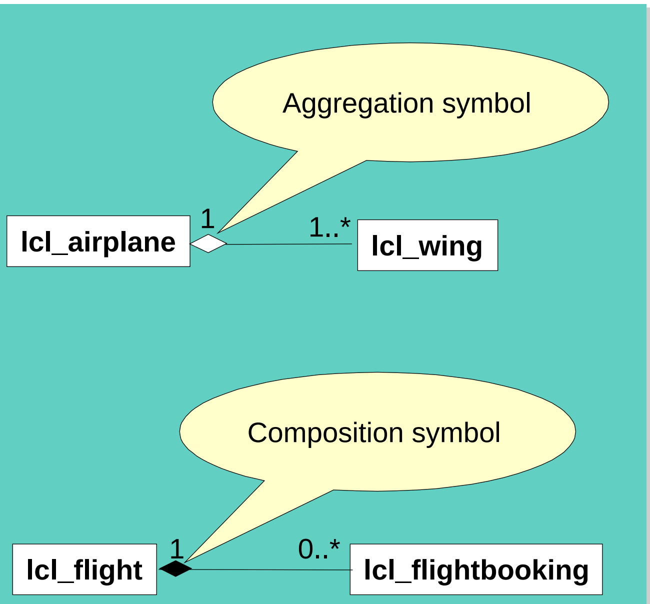
Aggregation and Composition

Aggregation

- Special type of association
- Whole-part relationships

Composition

- Special type of aggregation
- Existence-dependent whole-part relationships



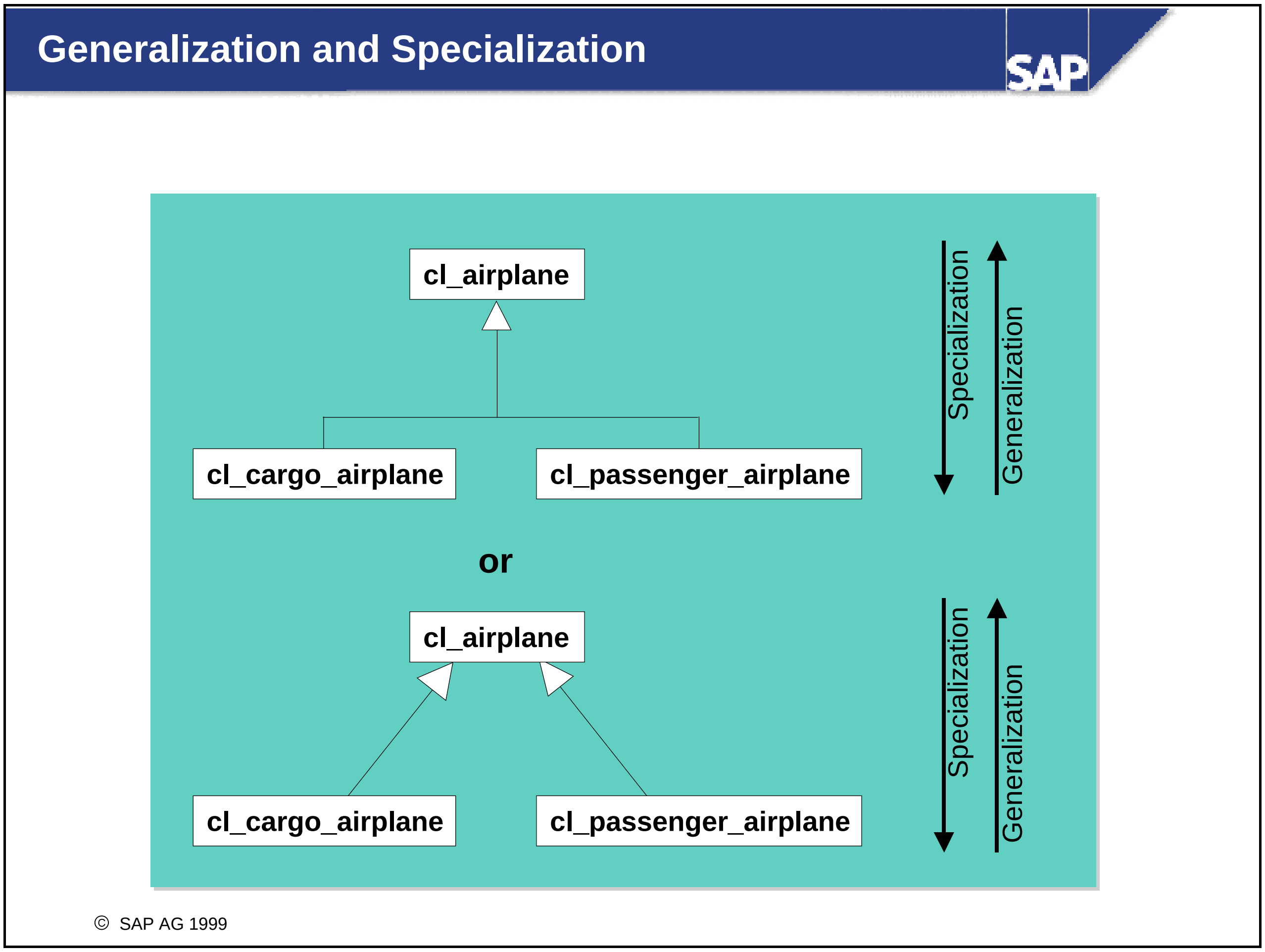
Aggregation is a special kind of association. Aggregation describes one object that contains another or consists of other objects (whole-part). An airplane consists of wings. The relationship can be described by the words “consists of” or “is a part of”.

UML notation for aggregation:
An aggregation, like an association, is represented by a line between two classes, which then additionally has a small rhombus at one end. The rhombus is always at the aggregate end, that is, the whole object end. Otherwise the notation conventions are the same as for associations.

Composition is a special kind of aggregation. Composition describes the fact that the object contained cannot exist without the aggregate (for example, a flight booking cannot exist without the relevant flight).

Differences between composition and aggregation:
The cardinality on the aggregate side can only be one. Each part is only part of one composite object, otherwise the existence dependency would be contradictory. The lifetime of the individual parts is linked to the lifetime of the aggregate: parts are created either with or immediately after the aggregate, and they are destroyed either with or immediately before the aggregate.

UML notation for composition:
Like aggregation, composition is shown as a line between two classes and marked with a small rhombus on the aggregate side. However, in contrast to aggregation, the rhombus is filled in.



Generalization and Specialization

cl_airplane

cl_cargo_airplane

cl_passenger_airplane

or

cl_airplane

cl_cargo_airplane

cl_passenger_airplane

Specialization

Generalization

Specialization

Generalization

UML notation:
Generalization and specialization are denoted by triangular arrows that point from the subordinate class to the superclass.
Several arrows can be summarized into a tree.

Behavior Diagrams: Sequence Diagrams

- **Dynamic view of a model**
  - **Objects in existence at runtime**
  - **Interaction between objects**
  - **Time sequence of the interaction**

Sequence diagrams, unlike class diagrams, show the dynamics between objects. They are used to represent a particular process or a particular situation.

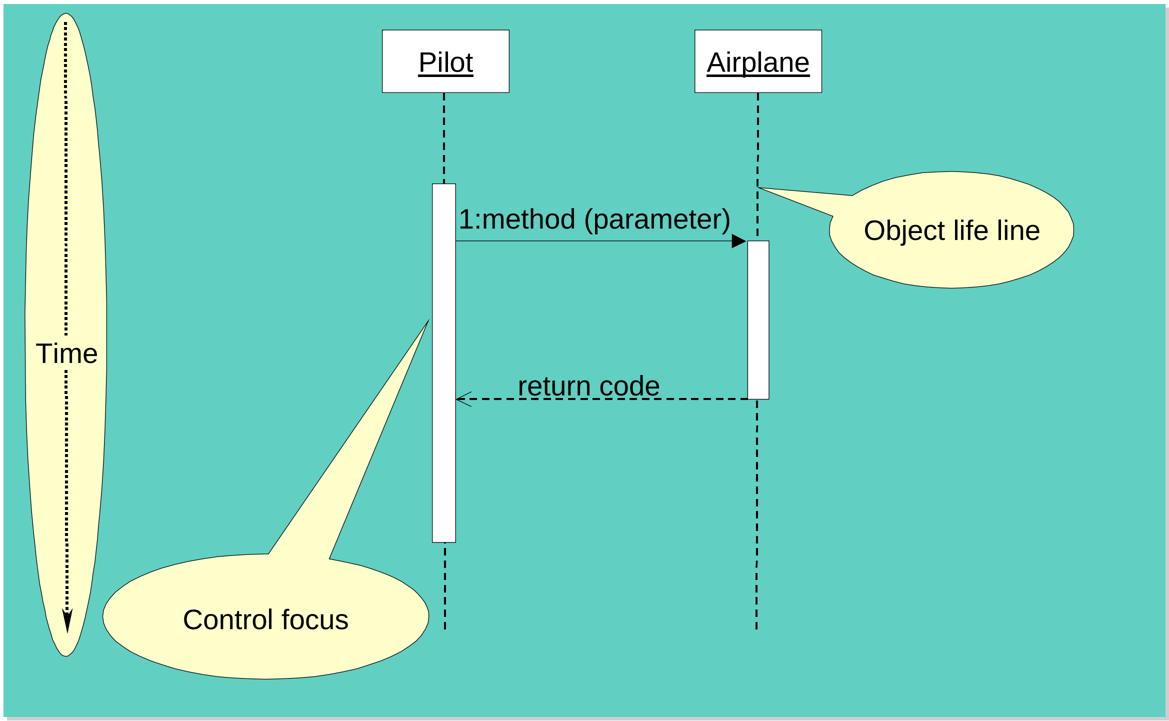
Sequence diagrams focus on the time sequence of the information exchange:

Creating and deleting objects.

Message exchange between objects.

Sequence diagrams have no notation for representing static methods.

Sequence Diagrams: Example (1)



UML notation:
Objects are represented by squares. You can write the object name in these squares in various ways:
<u>Object name</u>
<u>Object name:class name</u>
<u>:class name</u>

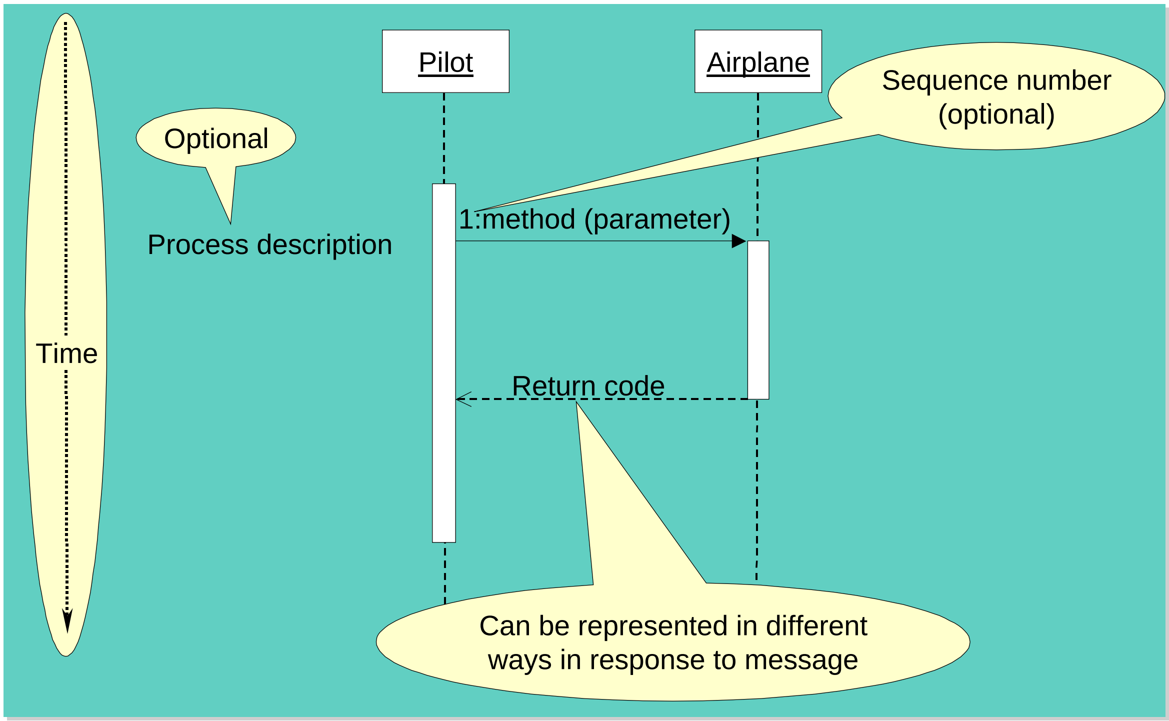
The object life line is represented by vertical dotted lines.

The control focus is shown as a vertical rectangle on the object life line. 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.

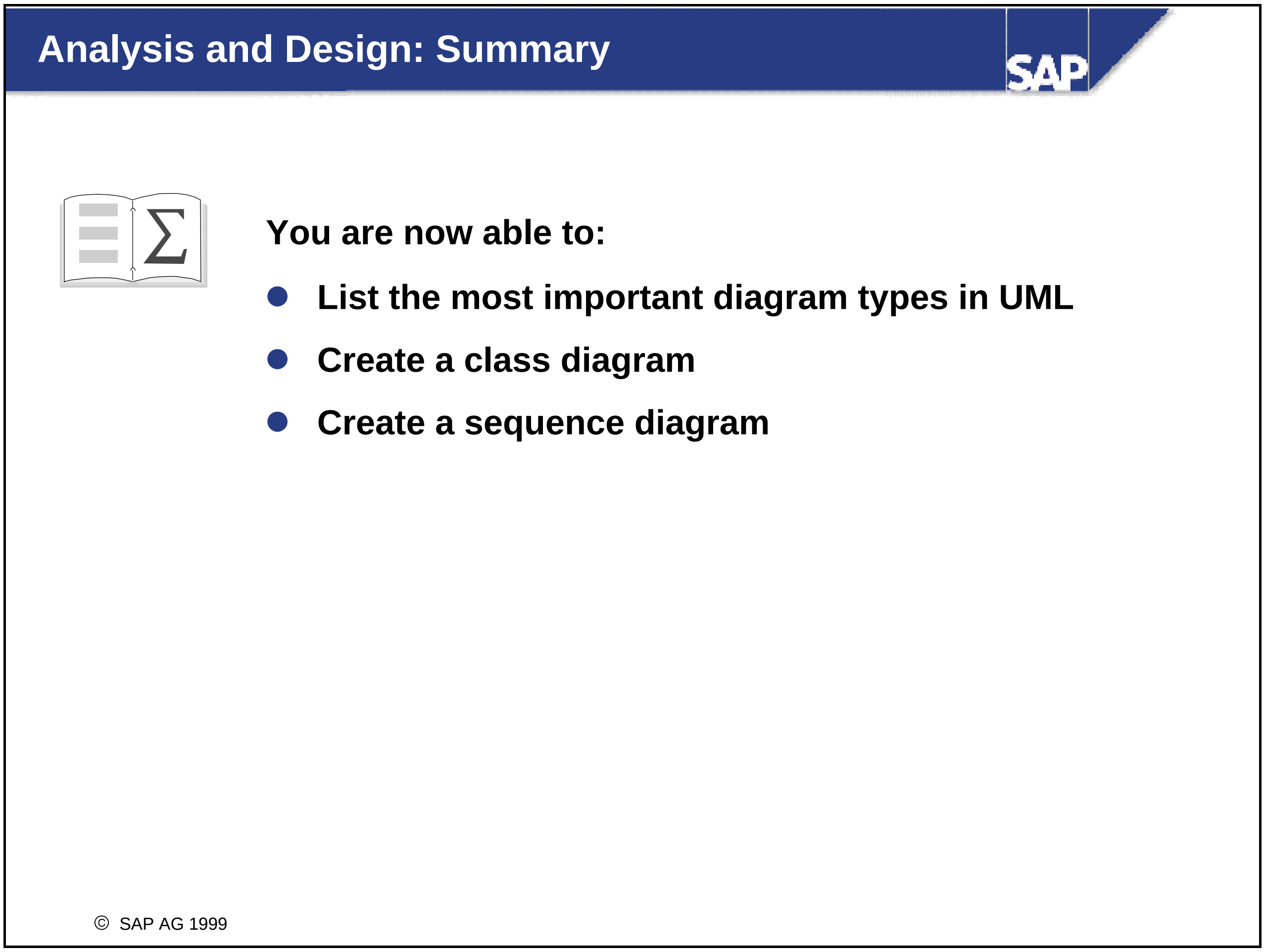
Sequence Diagrams: Example (2)


Pilot
Airplane
Sequence number (optional)
Optional
Process description
1:method (parameter)
Time
Return code
Can be represented in different ways in response to message

© SAP AG 1999

Messages are shown as horizontal arrows between the object lines. The message is written over the arrow as *Method (parameter)*. There are various options for representing the reply; in this example, the arrow is shown as a returning arrow.

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



Analysis and Design: Summary

You are now able to:

- List the most important diagram types in UML
- Create a class diagram
- Create a sequence diagram

Analysis and Design Exercises

Unit: Analysis and Design

Topic: UML Class Diagrams

At the end of this exercise, you will be able to:

- Create a UML class diagram
-

An airline needs to manage its airplanes.

1-1 On a sheet of paper, create a class diagram using UML notation that contains the following classes:

- Airline:	*lcl_carrier*
– Airplane (general):	*lcl_airplane*
– Passenger airplane:	*lcl_passenger_airplane*
– Cargo airplane:	*lcl_cargo_airplane*

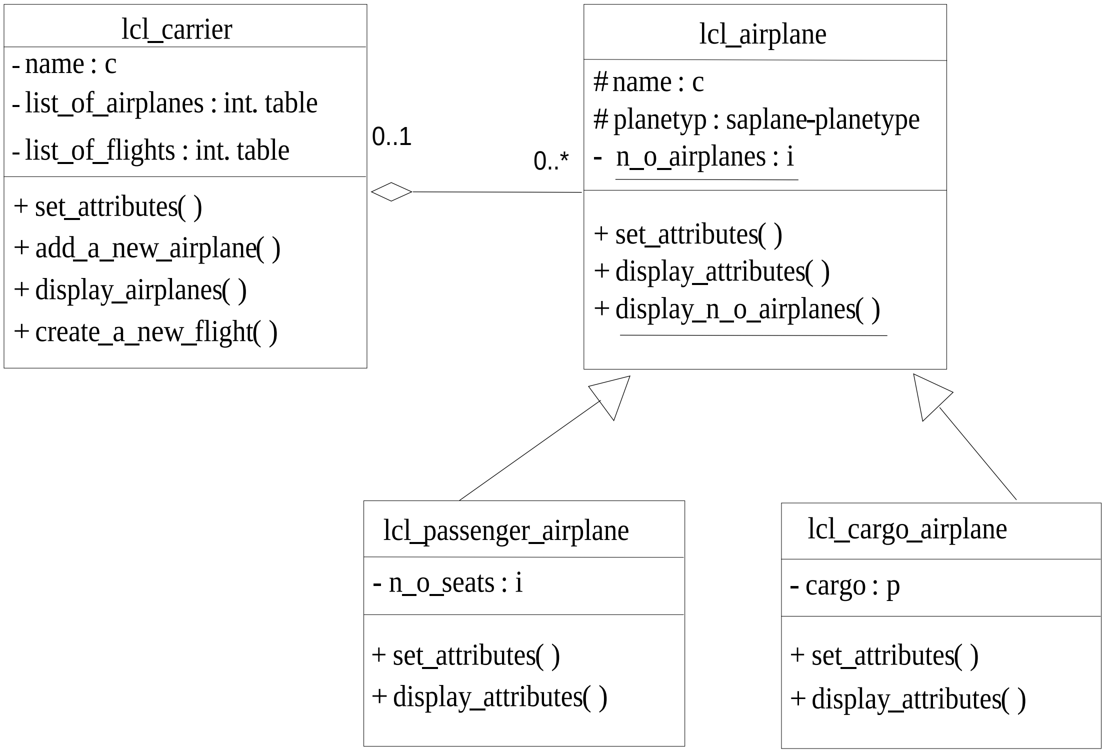
1-1-1 Choose a few useful attributes and methods for each class.

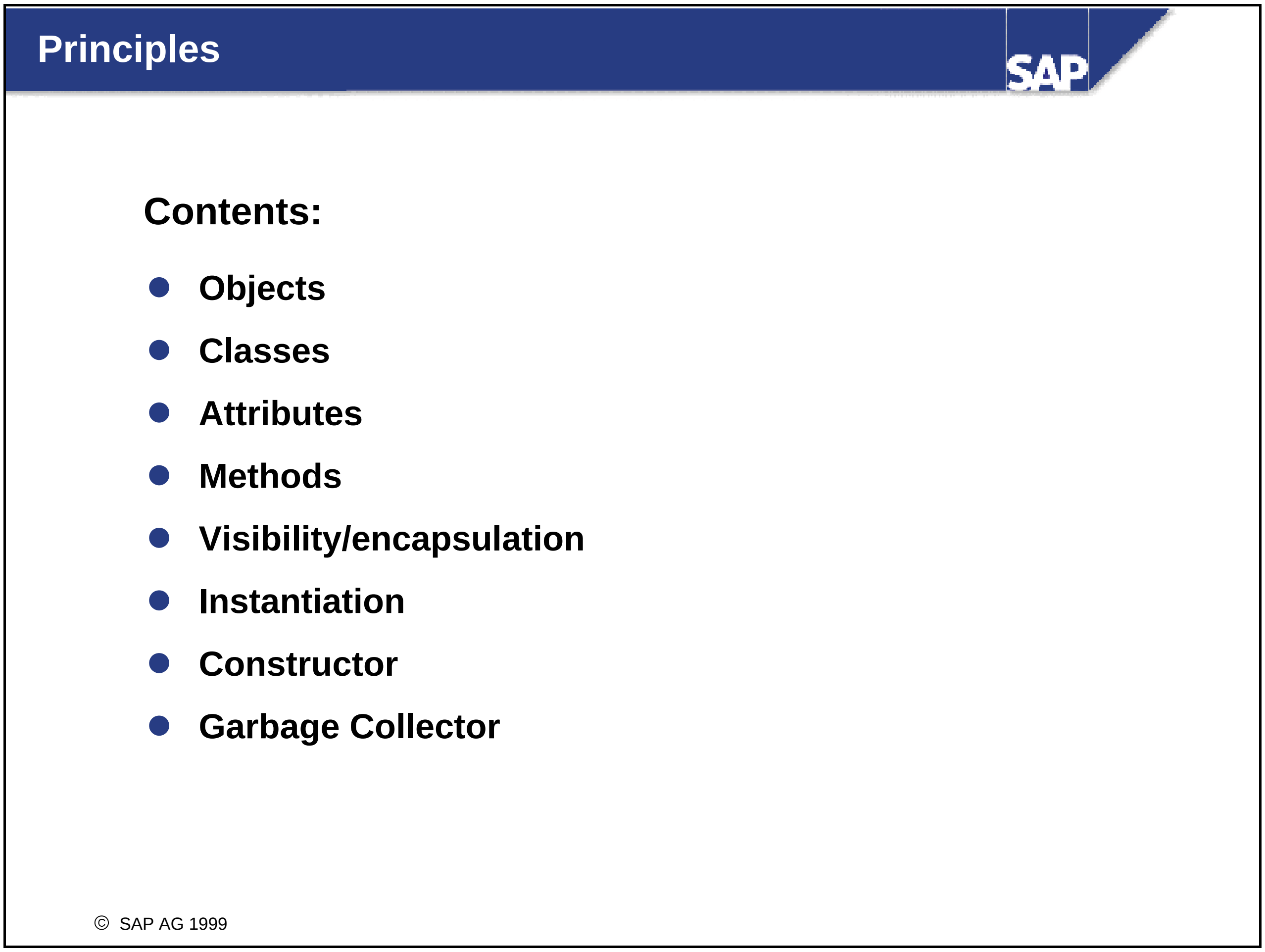
1-1-2 Fill in the relationships between the classes and add possible cardinalities.

Analysis and Design Solutions

Unit: Analysis and Design

Topic: UML Class Diagrams

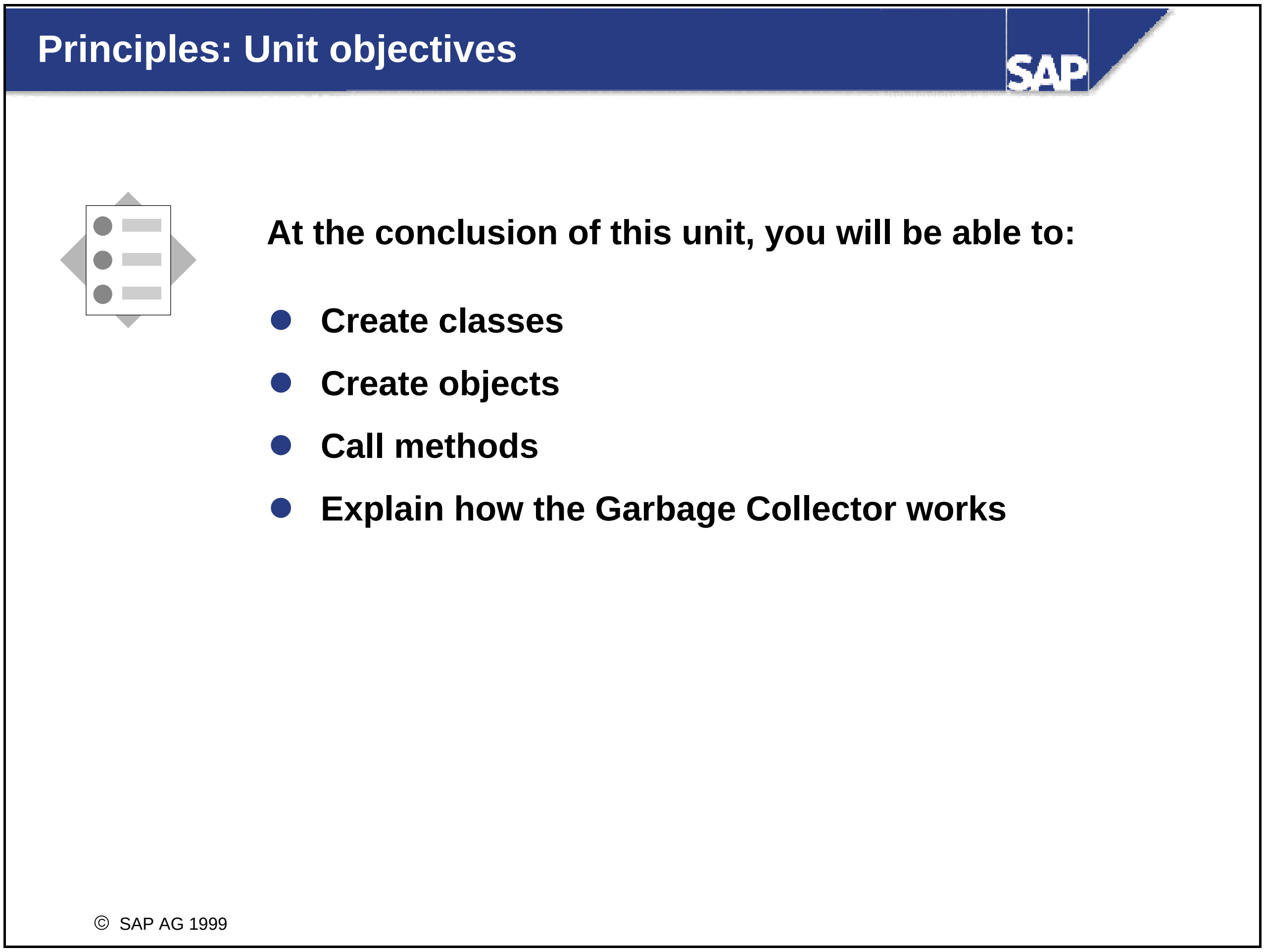




Principles

Contents:

- Objects
- Classes
- Attributes
- Methods
- Visibility/encapsulation
- Instantiation
- Constructor
- Garbage Collector



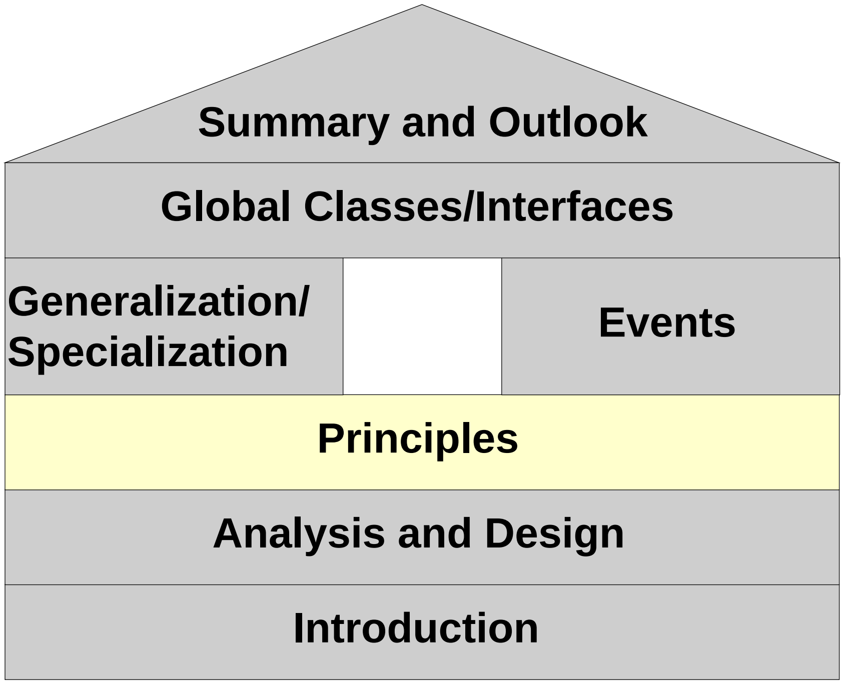
Principles: Unit objectives

SAP

At the conclusion of this unit, you will be able to:

- Create classes
- Create objects
- Call methods
- Explain how the Garbage Collector works

Principles: Course Overview Diagram



Summary and Outlook

Global Classes/Interfaces

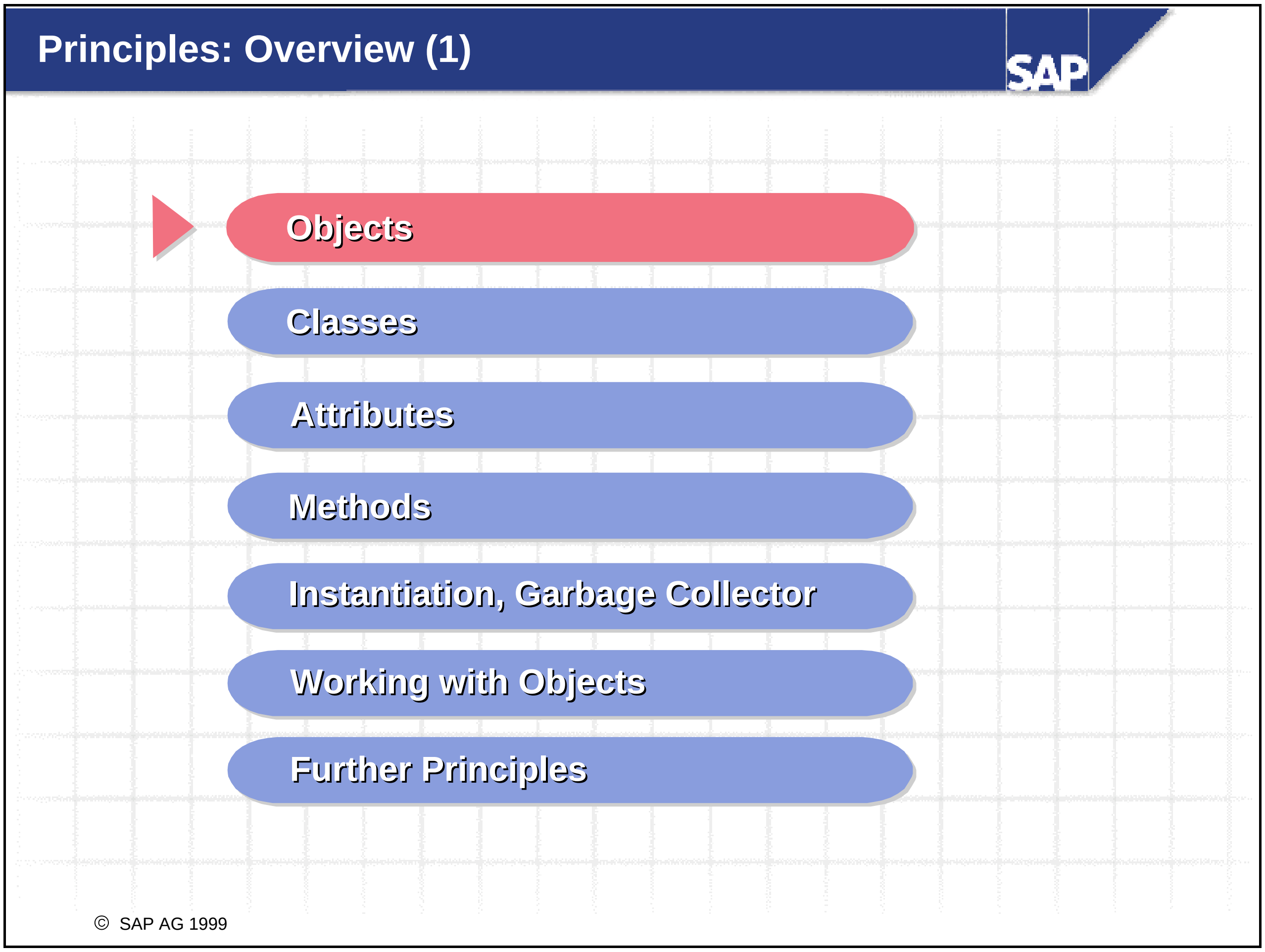
Generalization/ Specialization

Events

Principles

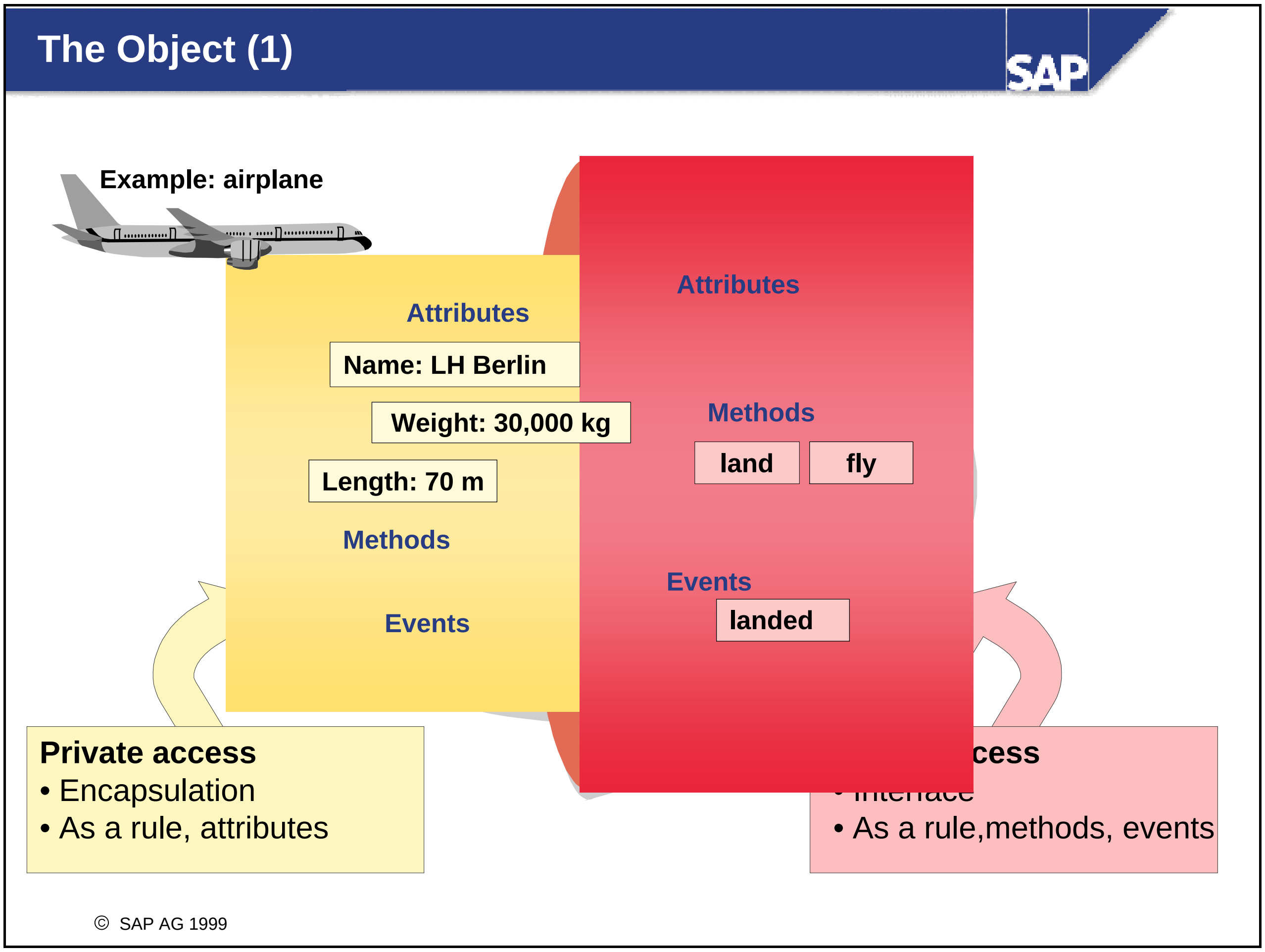
Analysis and Design

Introduction



Principles: Overview (1)

- Objects
- Classes
- Attributes
- Methods
- Instantiation, Garbage Collector
- Working with Objects
- Further Principles



The Object (1)

Example: airplane

Attributes

Name: LH Berlin

Weight: 30,000 kg

Length: 70 m

Methods

Events

Attributes

Methods

land fly

Events

landed

Private access
- Encapsulation
- As a rule, attributes

cess
- interface
- As a rule,methods, events

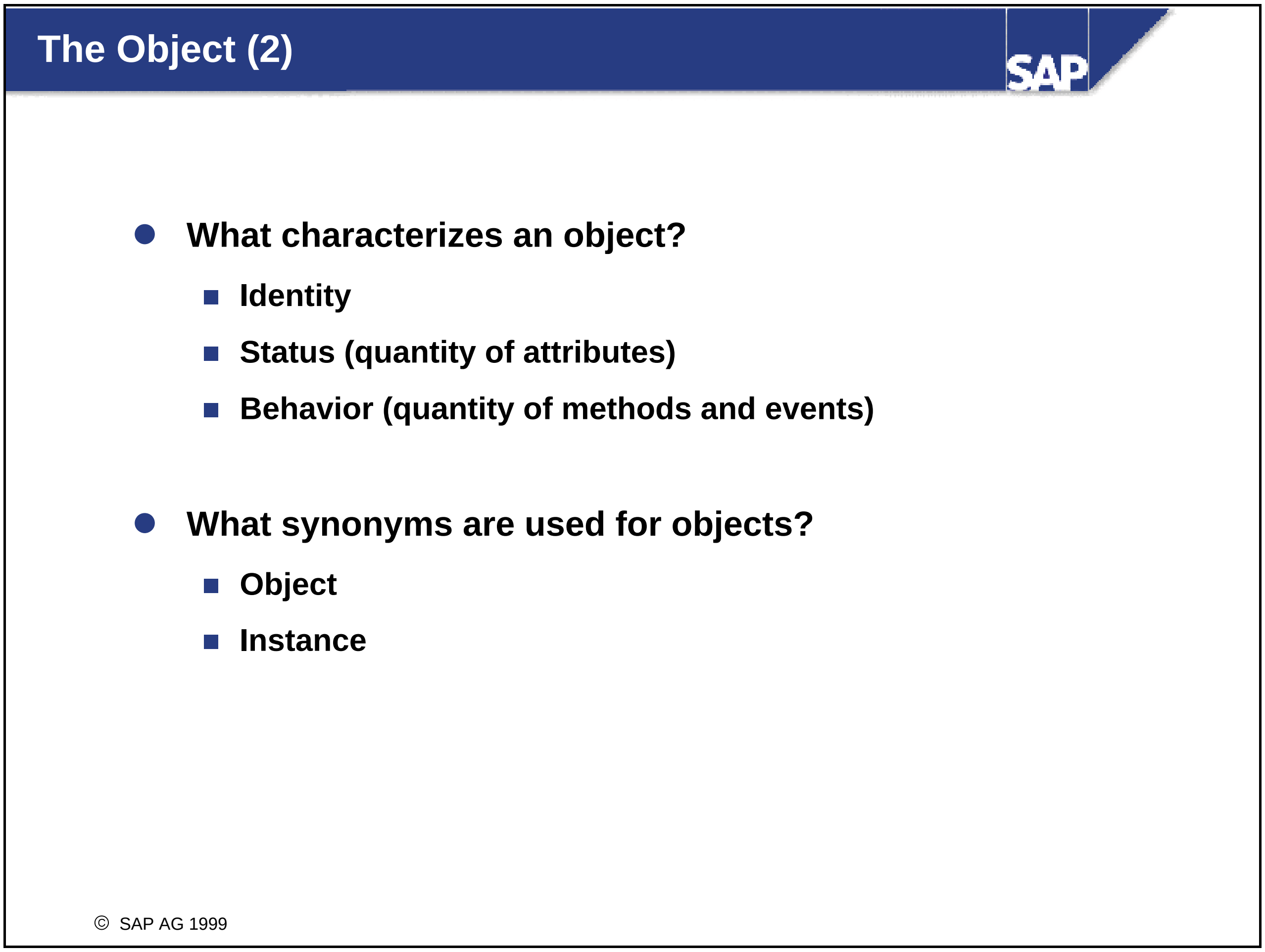
© SAP AG 1999

The object in the above model has two layers: an outer shell and an inner core. Users can only see the outer shell, while the inner core remains hidden (the internal status of an object can only be seen within the object itself).

Public components (outer shell): the outer shell contains the components of the object that are visible to users, such as attributes (data), methods (functions) and events. All users have direct access to these components. The public components of an object form its external point of contact.

Private components (inner core): the components of the inner core (attributes, methods and events) are only visible within the object itself. The attributes of an object are generally private. These private attributes of an object can only be accessed using the methods of that object itself.

Why are the private components of an object “hidden”?
This principle is called “information hiding” or “encapsulation” and is used to protect the user.
Let us assume that an object changes its private components, while its external point of contact remains unchanged. Any user who simply needs to access the object’s external point of contact can carry on working with the object as usual. The user does not notice the change.
However, if an object changes its public components, then any user who accesses these public components must take these changes into account.

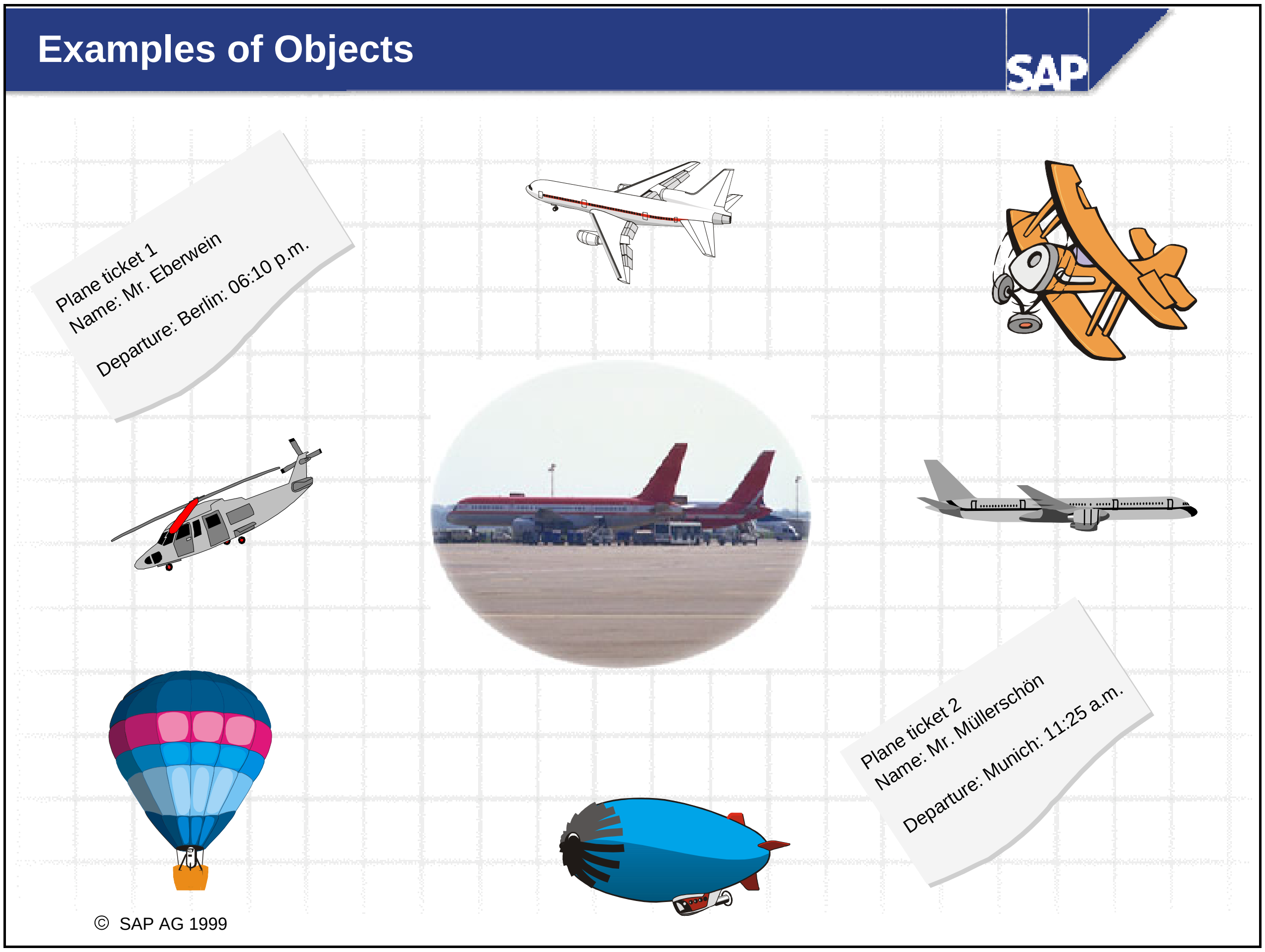


The Object (2)

- What characterizes an object?
  - Identity
  - Status (quantity of attributes)
  - Behavior (quantity of methods and events)
- What synonyms are used for objects?
  - Object
  - Instance

Every object has an identity, a status (quantity of attributes) and behavior (quantity of methods and events). The structure and behavior of similar objects are defined in a class which they share.

Identity is a characteristic that differentiates each object from all other objects. Identity is often confused with having the same attribute values or with a unique name. Two different objects can have identical attribute values and still not be identical.
Example:
Two coffee cups are the same height and diameter, have the same handle and are both white. Although they look exactly the same, they are still two separate cups.

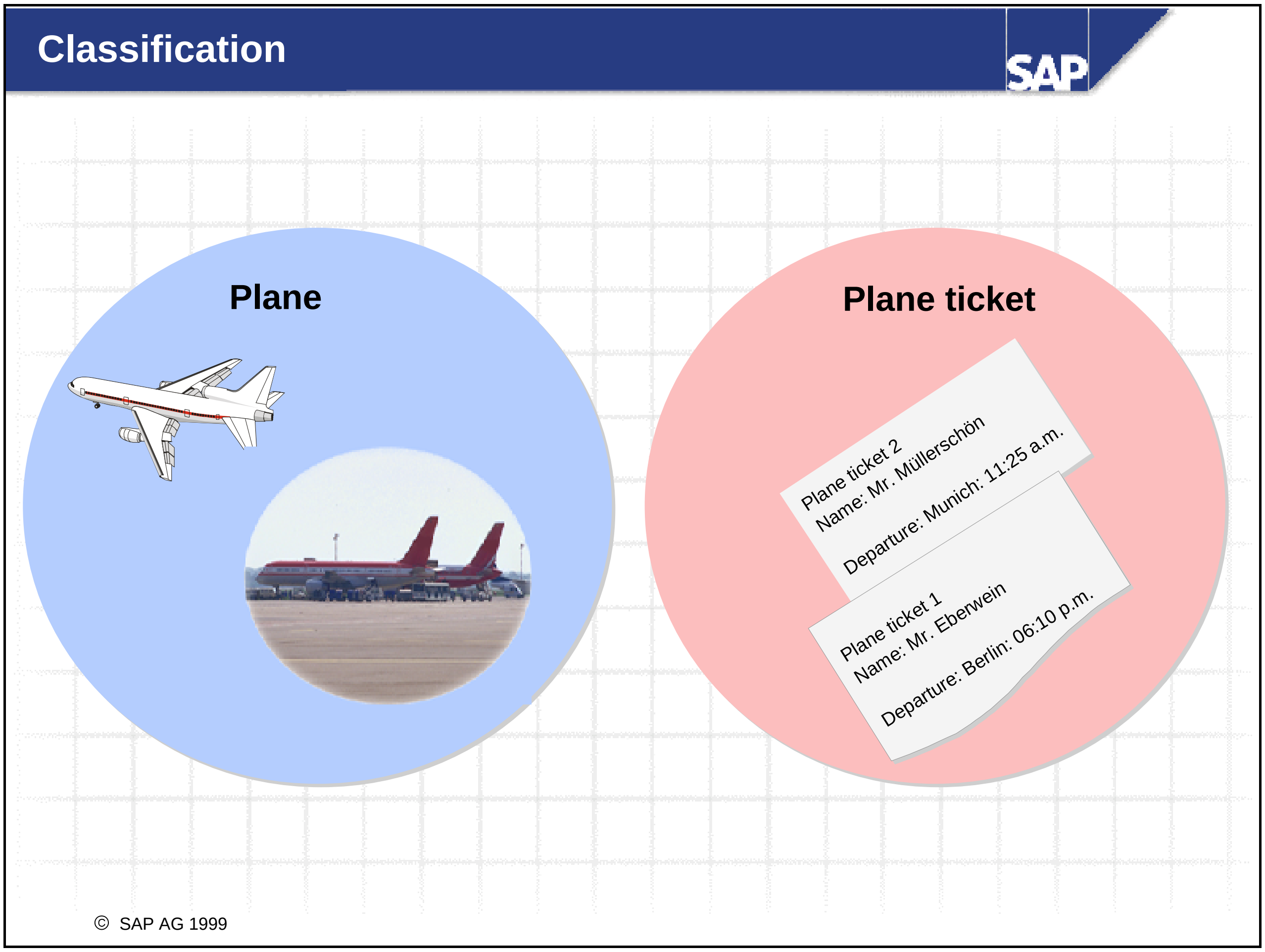


Examples of Objects

Plane ticket 1
Name: Mr. Eberwein
Departure: Berlin: 06:10 p.m.

Plane ticket 2
Name: Mr. Müllerschön
Departure: Munich: 11:25 a.m.

A number of different objects are shown on this slide. Similar objects can be grouped into classes.



Classification

Plane ticket 2
Name: Mr. Müllerschön
Departure: Munich: 11:25 a.m.

Plane ticket 1
Name: Mr. Eberwein
Departure: Berlin: 06:10 p.m.

In the real world, there are objects, such as various airplanes and plane tickets. Some of these objects are very similar, that is, they can be described using the same attributes or characteristics and provide the same functions.

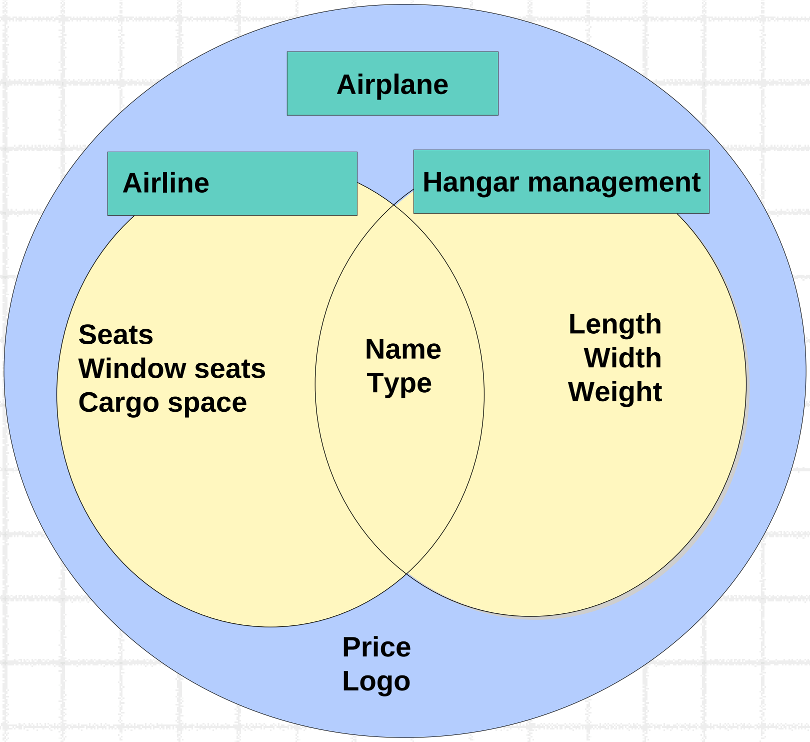
Similar objects are grouped together in classes. Each class is described once, and each object is then created in accordance with this blueprint.
A class is therefore a description of a quantity of objects characterized by the same structure and the same behavior.

An object is a concrete example of a class, the airplane class is a description of the objects LH Munich, LH New York etc.. Objects that belong to the same class have the same attributes and can be accessed using the same methods. There is only one of each class within a software system, but each class can contain several objects.

Principles: Overview (2)

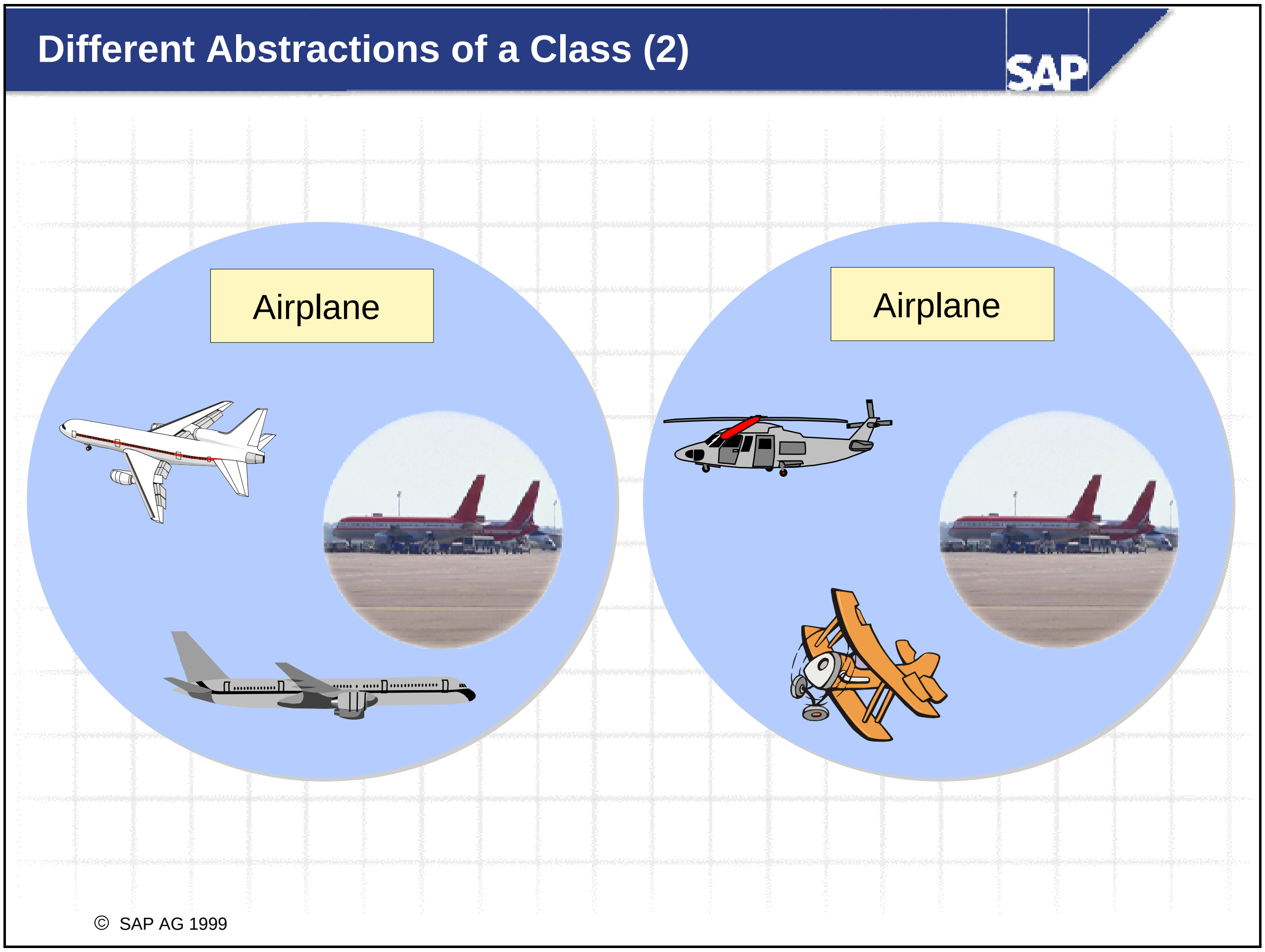
- Objects
- Classes
- Attributes
- Methods
- Instantiation, Garbage Collector
- Working with Objects
- Further Principles

Different Abstractions of a Class



In this context, abstractions are a simplified representations of complex relationships in the real world. An actually existing object is abstracted to the significant dimensions that are to be mapped. Insignificant details are left out in order to aid understanding of the overall system.

This example concerns airplanes. Software for airlines and software for an airport’s hangar management contain different abstractions (classes) for these objects.

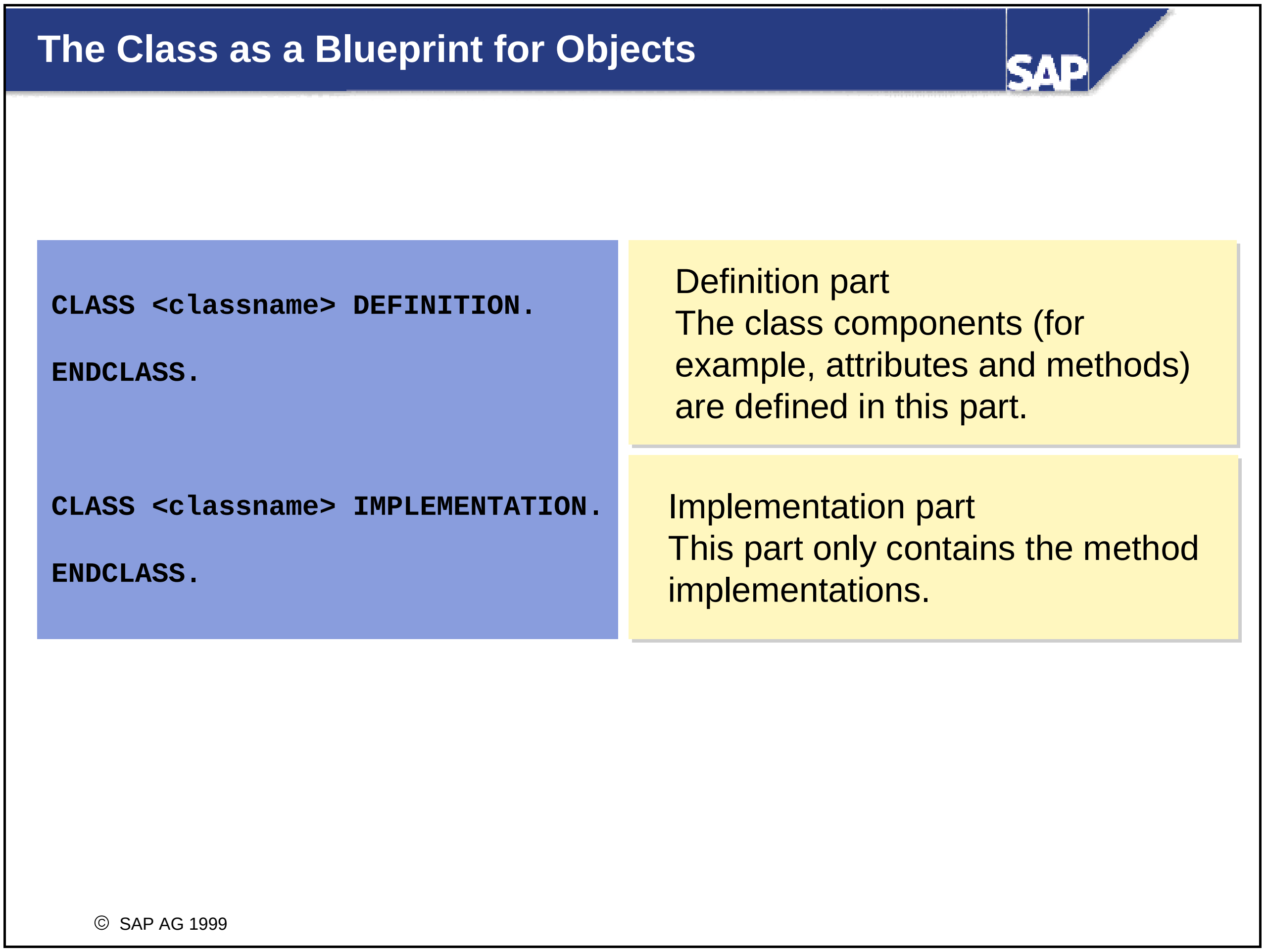


Different Abstractions of a Class (2)

A class can contain very different objects depending on the abstraction.

While in one software system the class ‘airplane’ only describes ‘actual’ airplanes, in the other system it it describes all aircraft.

Both classes have the same name but describe different objects.



The Class as a Blueprint for Objects

```
CLASS <classname> DEFINITION.

ENDCLASS.


CLASS <classname> IMPLEMENTATION.

ENDCLASS.
```

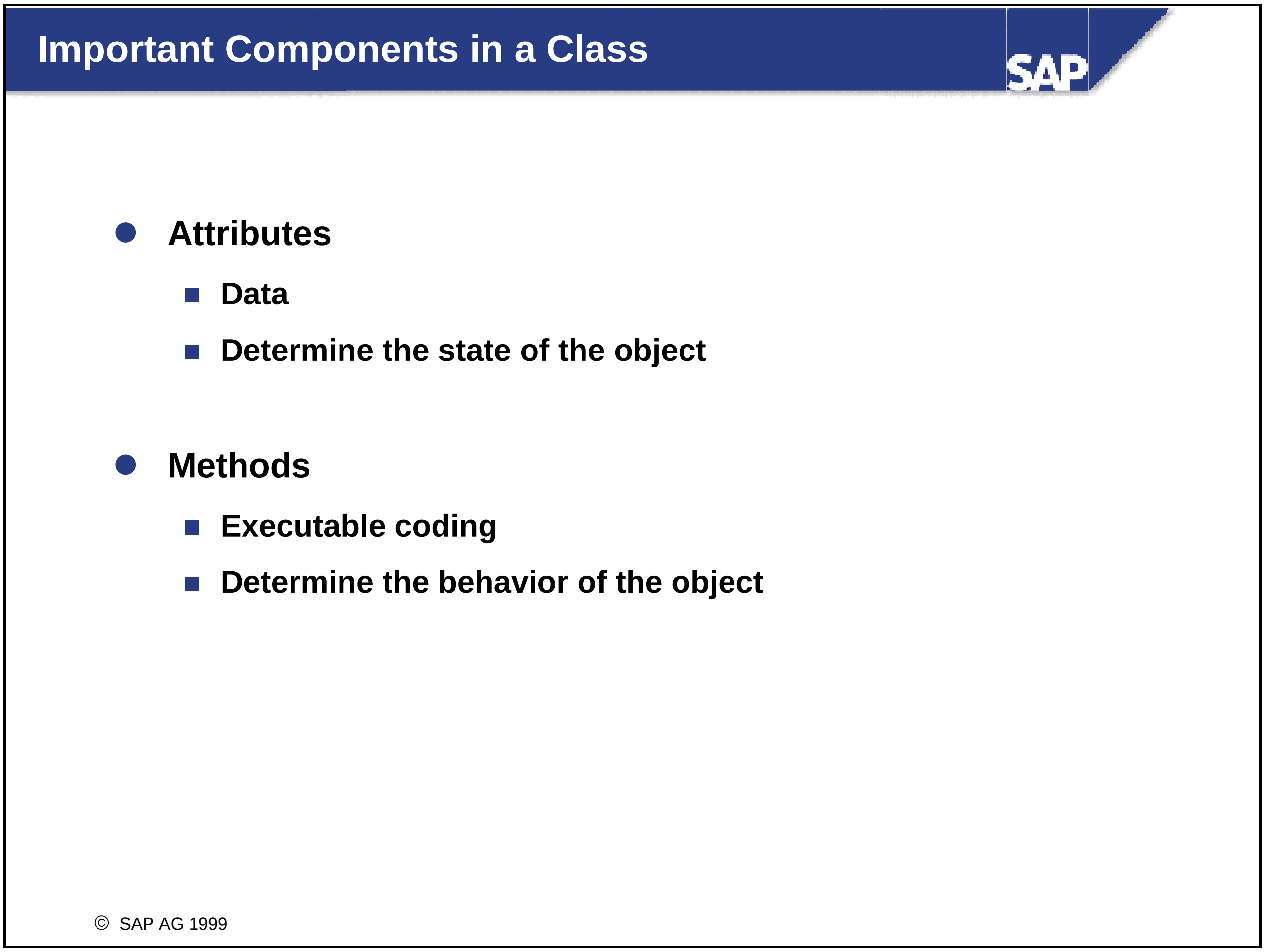
Definition part
The class components (for example, attributes and methods) are defined in this part.

Implementation part
This part only contains the method implementations.

A class is a description of a number of objects that have the same structure and the same behavior. A class is therefore like a blueprint, in accordance with which all objects in that class are created.

The 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.

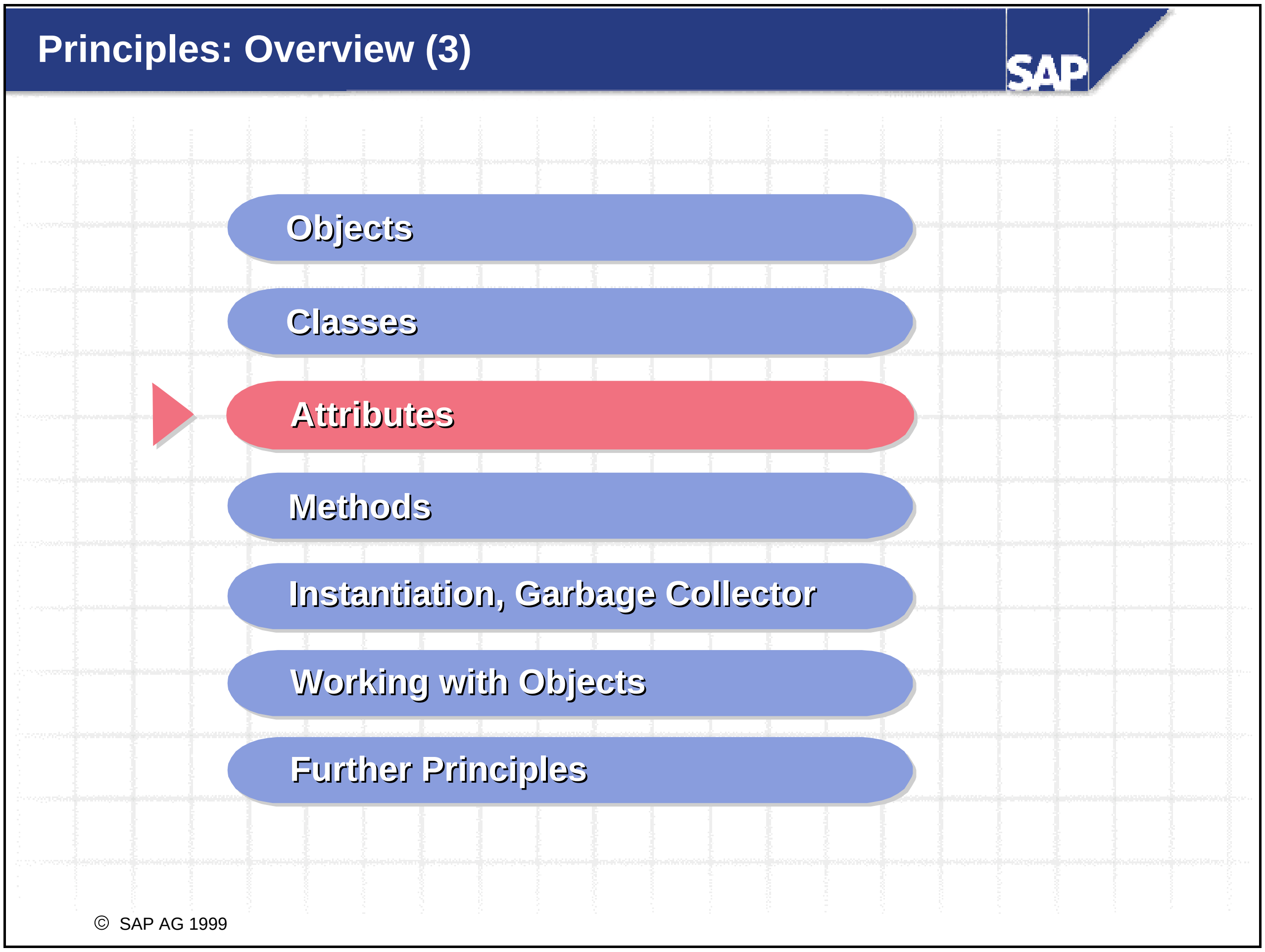
The CLASS statement cannot be nested, that is, you cannot define a class within a class.



Important Components in a Class

- Attributes
  - Data
  - Determine the state of the object
- Methods
  - Executable coding
  - Determine the behavior of the object

Further components in classes are events and interfaces, which will be explained later.

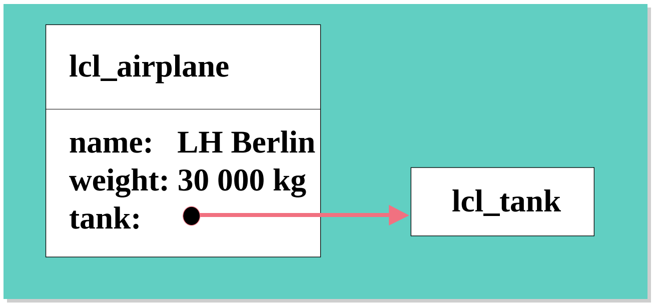


Principles: Overview (3)

- Objects
- Classes
- Attributes
- Methods
- Instantiation, Garbage Collector
- Working with Objects
- Further Principles

Attributes

- Attribute types can have any kind of data type:
  - Elementary types:
    - C, I, P, STRING
    - TYPE REF TO (References to objects/interfaces)
  - Define your own types



Attributes describe the data that can be stored in the objects in a class.

Class attributes can be of any type:

Data types: scalar (for example, data element), structured, in tables

ABAP elementary types (C, I, ...)

Object references

Interface references

Attributes of the airplane class are, for example:

Name

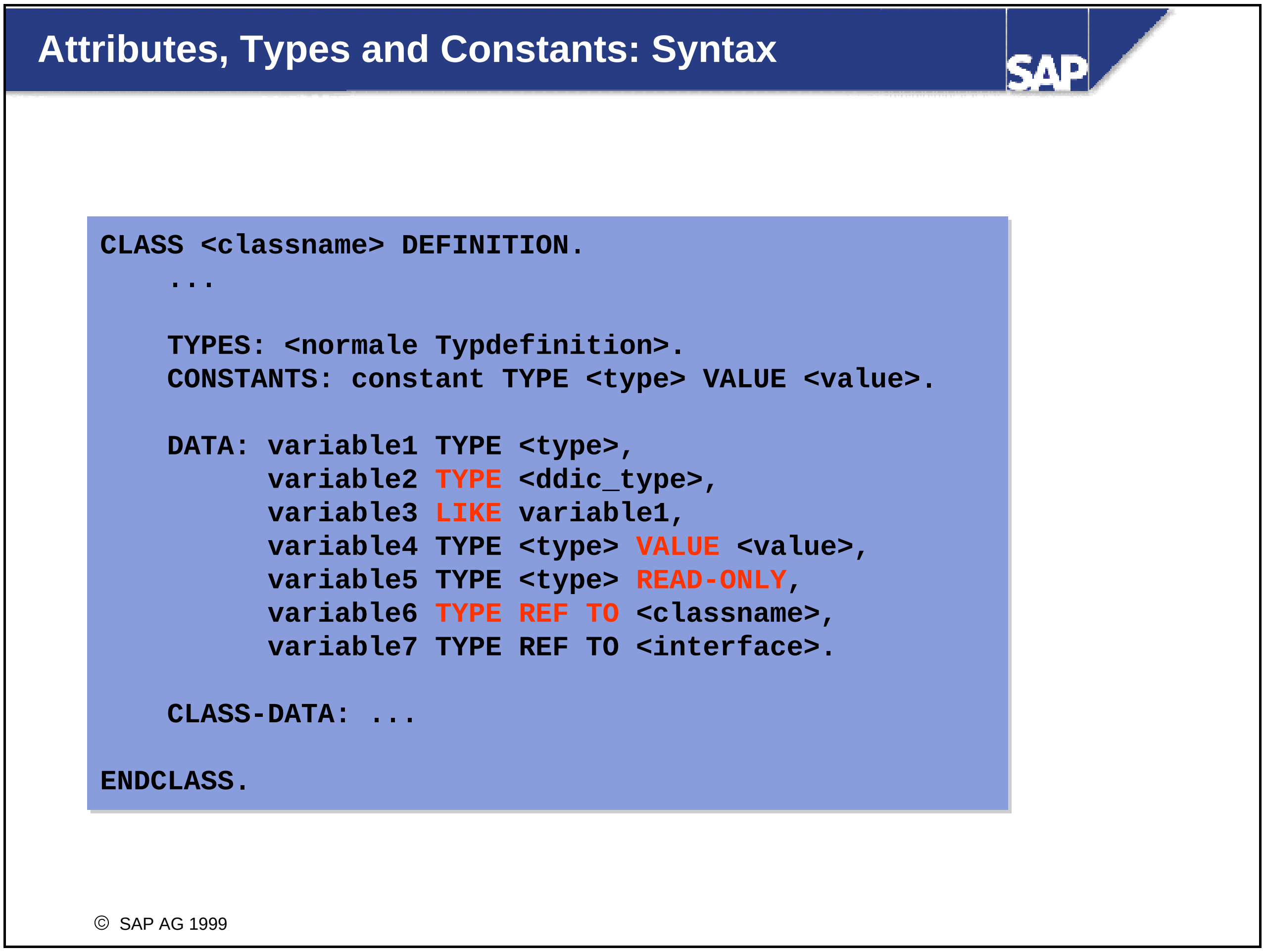
Seats

Weight

Length

Wings

Tank



Attributes, Types and Constants: Syntax

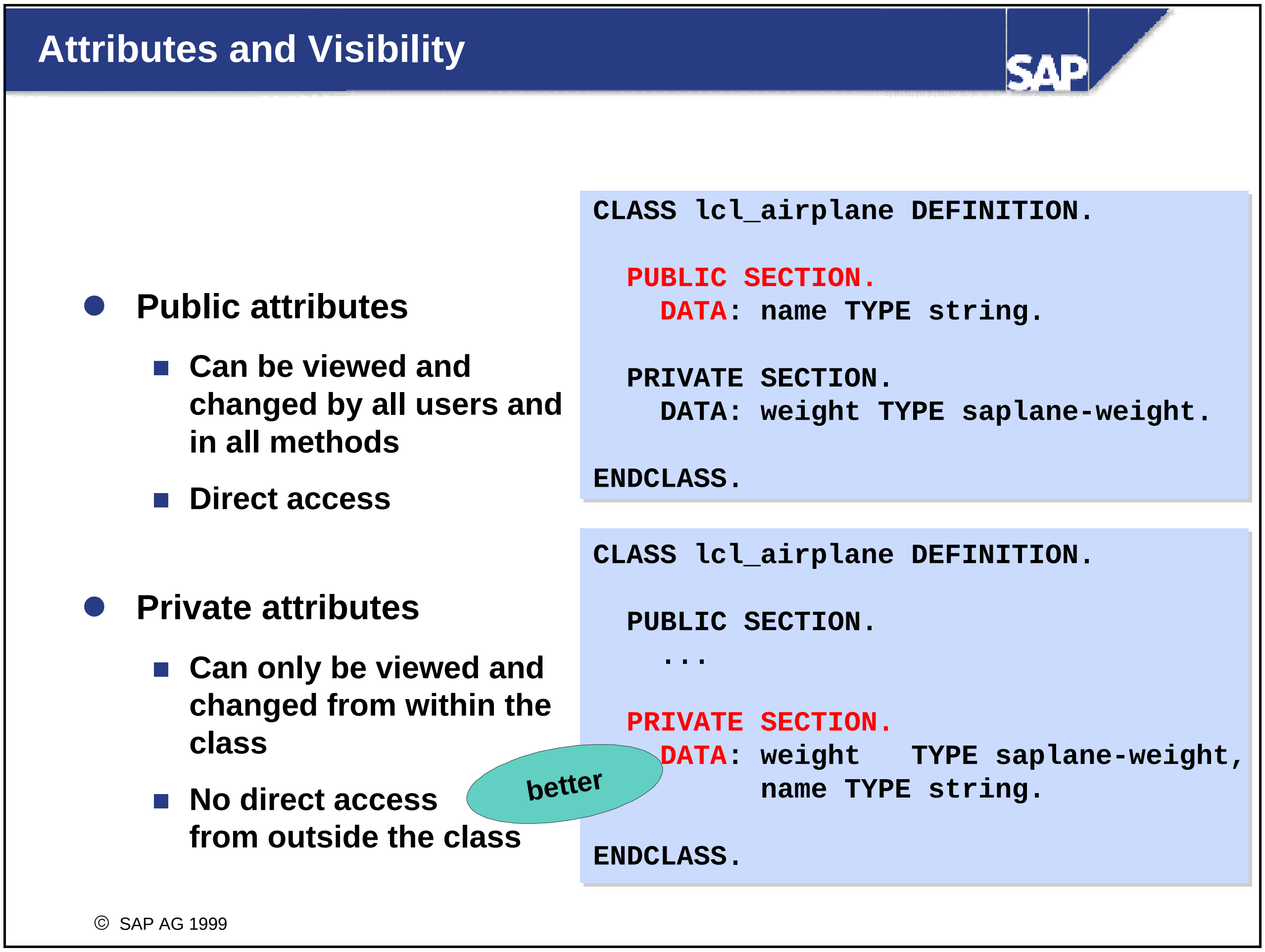
```
CLASS <classname> DEFINITION.
    ...

    TYPES: <normale Typdefinition>.
    CONSTANTS: constant TYPE <type> VALUE <value>.

    DATA: variable1 TYPE <type>,
          variable2 TYPE <ddic_type>,
          variable3 LIKE variable1,
          variable4 TYPE <type> VALUE <value>,
          variable5 TYPE <type> READ-ONLY,
          variable6 TYPE REF TO <classname>,
          variable7 TYPE REF TO <interface>.

    CLASS-DATA: ...

ENDCLASS.
```

In classes, you can only use the TYPE reference to refer to data types in the ABAP Dictionary.

You can only use the LIKE reference for local data objects.

The READ-ONLY addition means that a public attribute declared with DATA can be read from outside, but can only be changed by methods within the 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.



Attributes and Visibility

- **Public attributes**
  - **Can be viewed and changed by all users and in all methods**
  - **Direct access**

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    DATA: name TYPE string.

  PRIVATE SECTION.
    DATA: weight TYPE saplane-weight.

ENDCLASS.
```

- **Private attributes**
  - **Can only be viewed and changed from within the class**
  - **No direct access from outside the class**

better

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    ...

  PRIVATE SECTION.
    DATA: weight   TYPE saplane-weight,
          name TYPE string.

ENDCLASS.
```

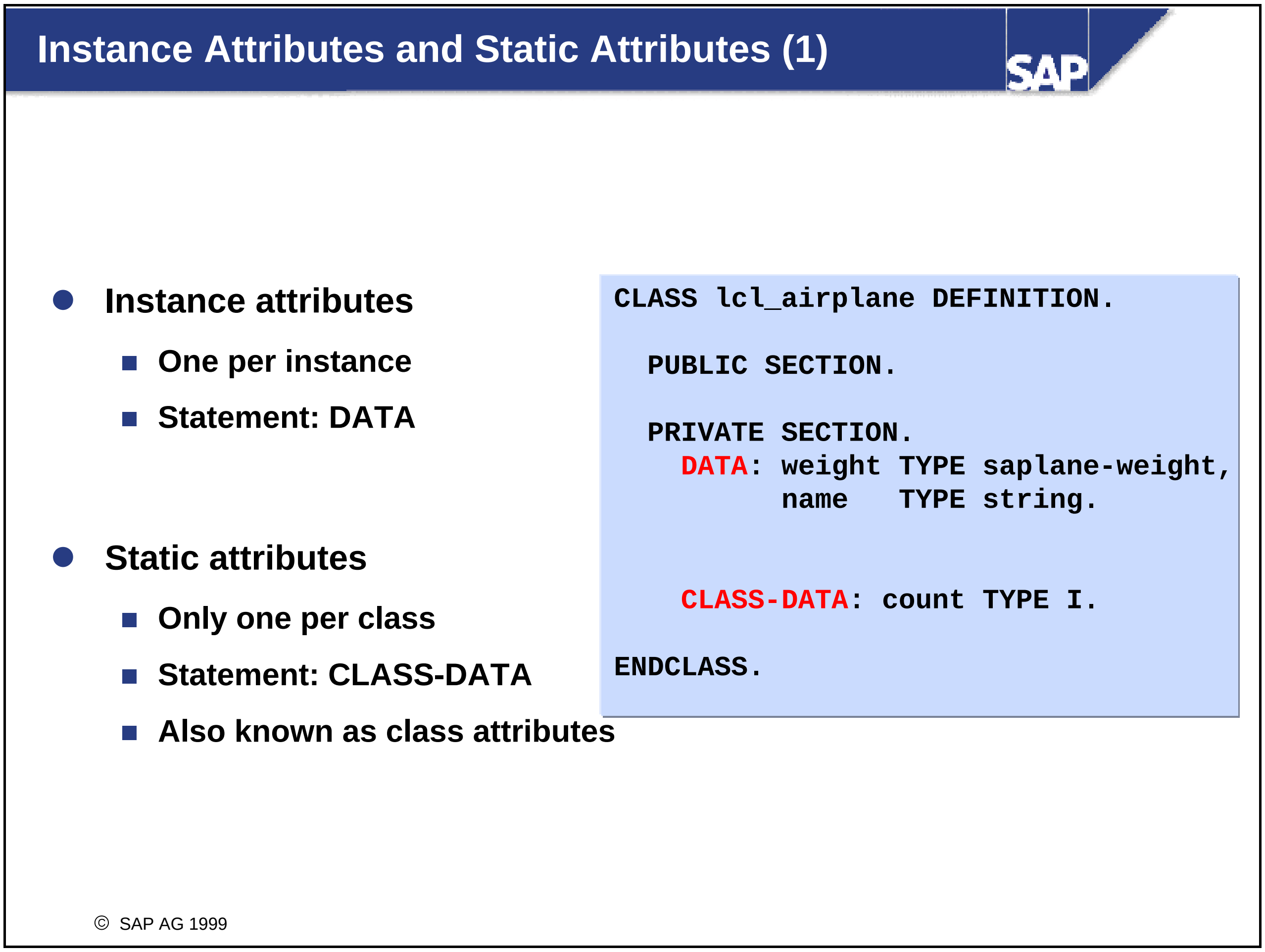
You can protect attributes against access from outside by characterizing them as private attributes (defined in the PRIVATE SECTION).

Attributes and their values that may be used directly by an external user are public attributes and are defined in the PUBLIC SECTION.

In the above example for class lcl_airplane, the name attribute is initially defined as a public attribute and the weight attribute is defined as a private attribute.

Public attributes belong to the class ‘external point of contact’ that is, their implementation is publicized. If you want to hide the internal implementation from users, you must define internal and external views of attributes.

As a general rule, you should define as few public attributes as possible.



Instance Attributes and Static Attributes (1)

- **Instance attributes**
  - **One per instance**
  - **Statement: DATA**
- **Static attributes**
  - **Only one per class**
  - **Statement: CLASS-DATA**
  - **Also known as class attributes**

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.

  PRIVATE SECTION.
    DATA: weight TYPE saplane-weight,
          name   TYPE string.


    CLASS-DATA: count TYPE I.

ENDCLASS.
```

There are two kinds of attributes

Static attributes

Instance attributes

Instance attributes are attributes that exist separately for each object.
Instance attributes are defined using the DATA keyword.

Static attributes exist once only for each class and are visible for all (runtime) instances in that class. Static attributes usually contain information that is common to all instances, such as:

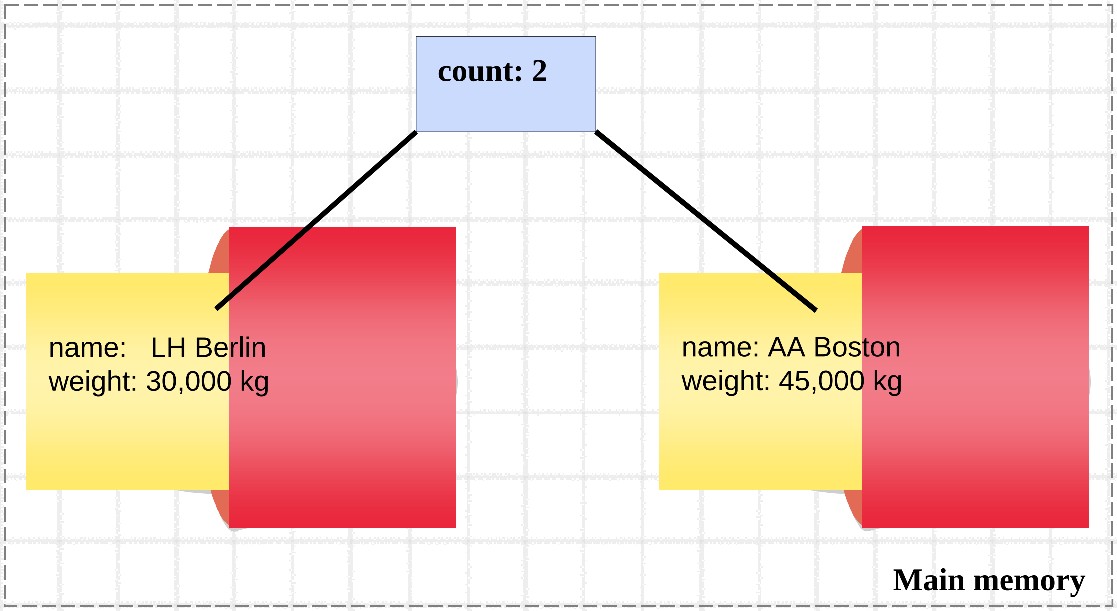
Data that is the same in all instances

Administrative information about the instances in that class (for example, counters and so on)

Static attributes are defined using the CLASS-DATA keyword.

You may come across the expression “class attributes” in documentation, however, the official term in ABAP Objects (as in C++, Java) is “static” attributes.

Instance Attributes and Static Attributes (2)



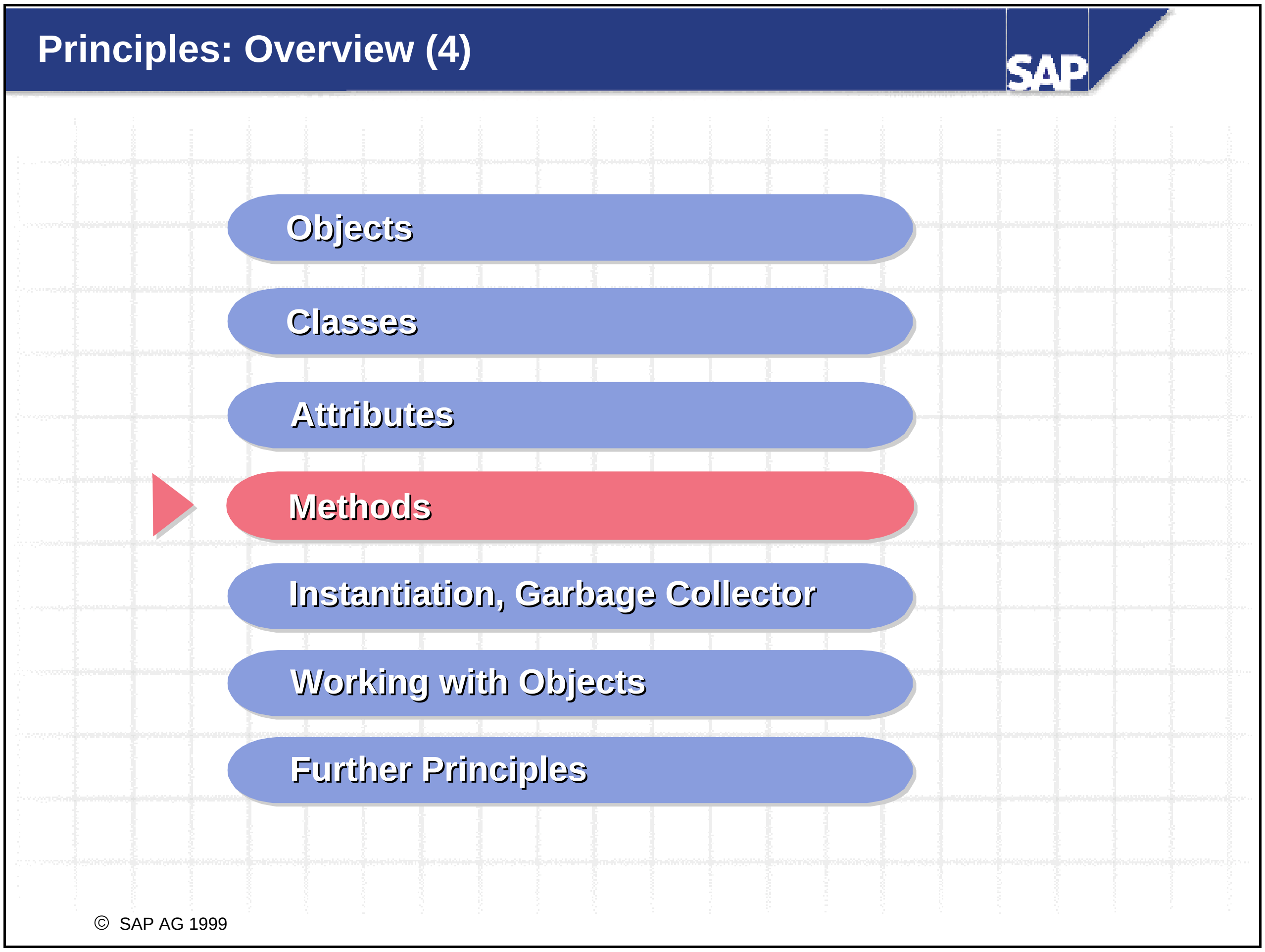
count: 2

name: LH Berlin
weight: 30,000 kg

name: AA Boston
weight: 45,000 kg

Main memory

🟦 **Static attributes for class LCL_AIRPLANE**

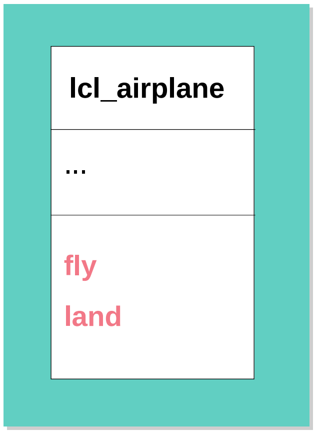


Principles: Overview (4)

- Objects
- Classes
- Attributes
- Methods
- Instantiation, Garbage Collector
- Working with Objects
- Further Principles

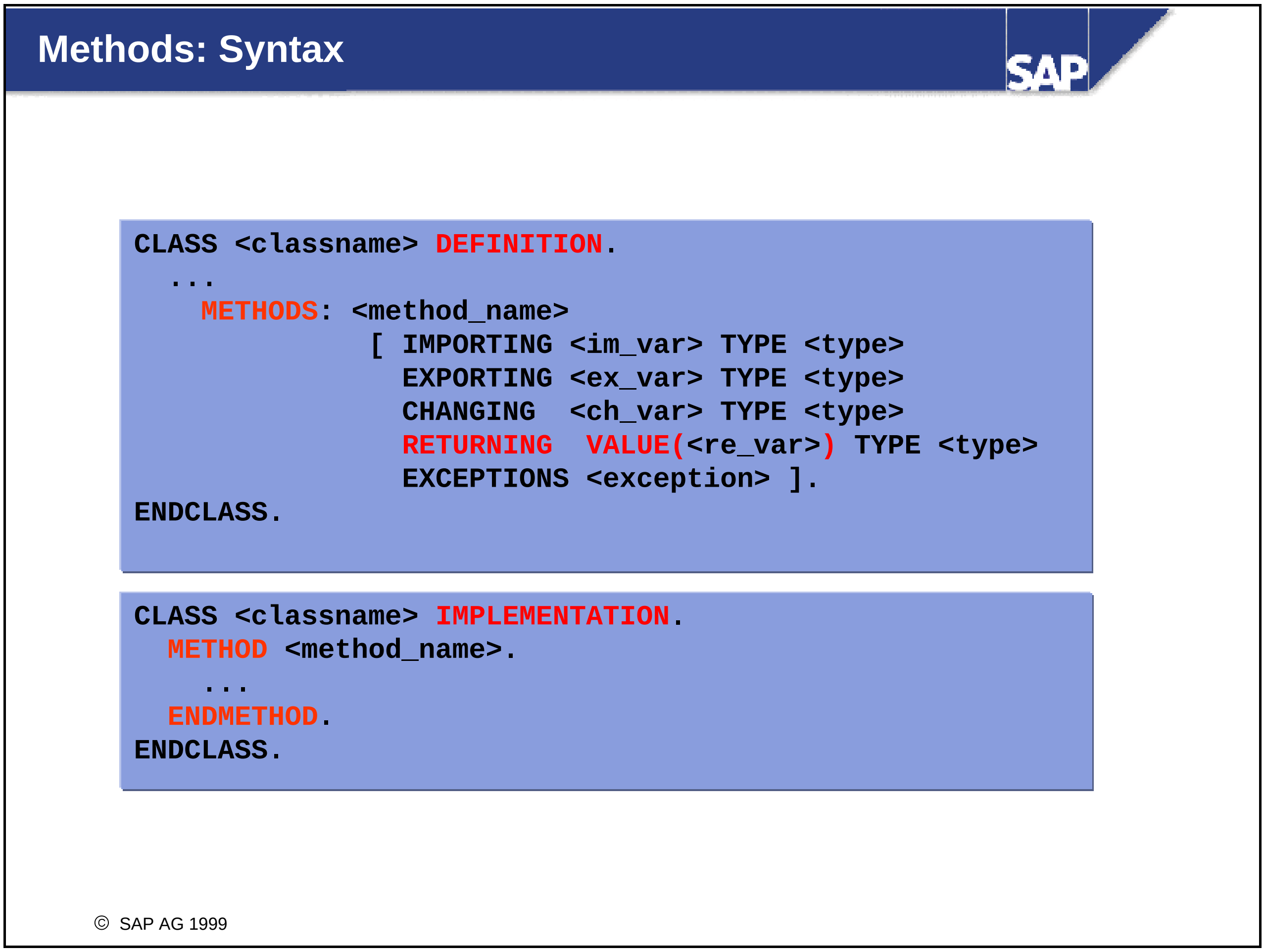
Methods

- Contain coding
- Have an interface



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

Methods have a parameter interface that enables them to receive values when they are called and pass values back to the calling program.



Methods: Syntax

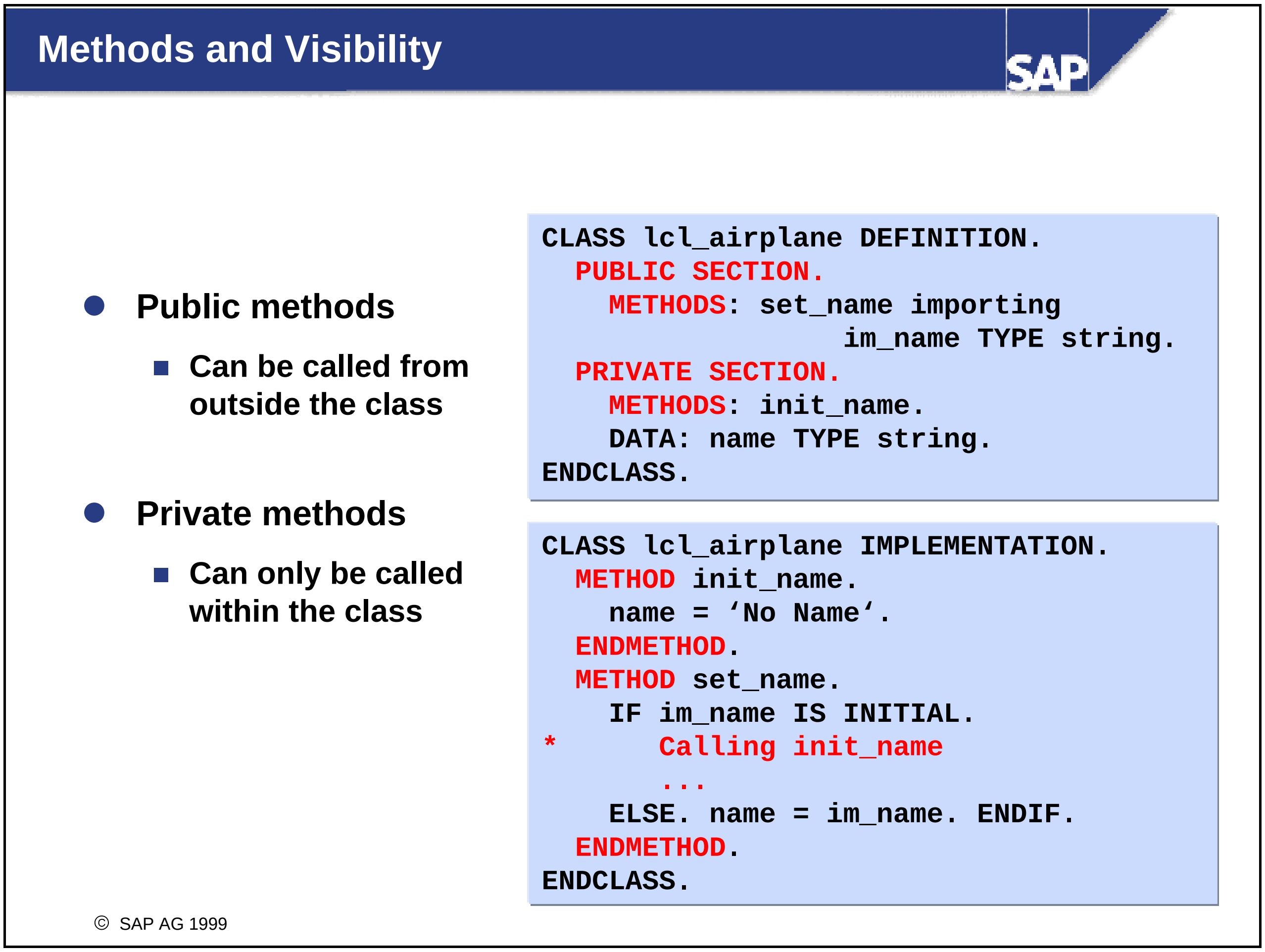
```
CLASS <classname> DEFINITION.
  ...
    METHODS: <method_name>
              [ IMPORTING <im_var> TYPE <type>
                EXPORTING <ex_var> TYPE <type>
                CHANGING  <ch_var> TYPE <type>
                RETURNING  VALUE(<re_var>) TYPE <type>
                EXCEPTIONS <exception> ].
ENDCLASS.
```

```
CLASS <classname> IMPLEMENTATION.
  METHOD <method_name>.
    ...
  ENDMETHOD.
ENDCLASS.
```

In ABAP Objects, methods can have IMPORTING, EXPORTING, CHANGING and RETURNING parameters as well as EXCEPTIONS. All parameters can be passed by value or reference.

You can define a return code for methods using RETURNING. You can only do this for a single parameter, which additionally must be passed as a value. Also, you cannot then define EXPORTING and CHANGING parameters. You can define functional methods using the RETURNING parameter (explained in more detail below).

All input parameters (IMPORTING, 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 passed 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.



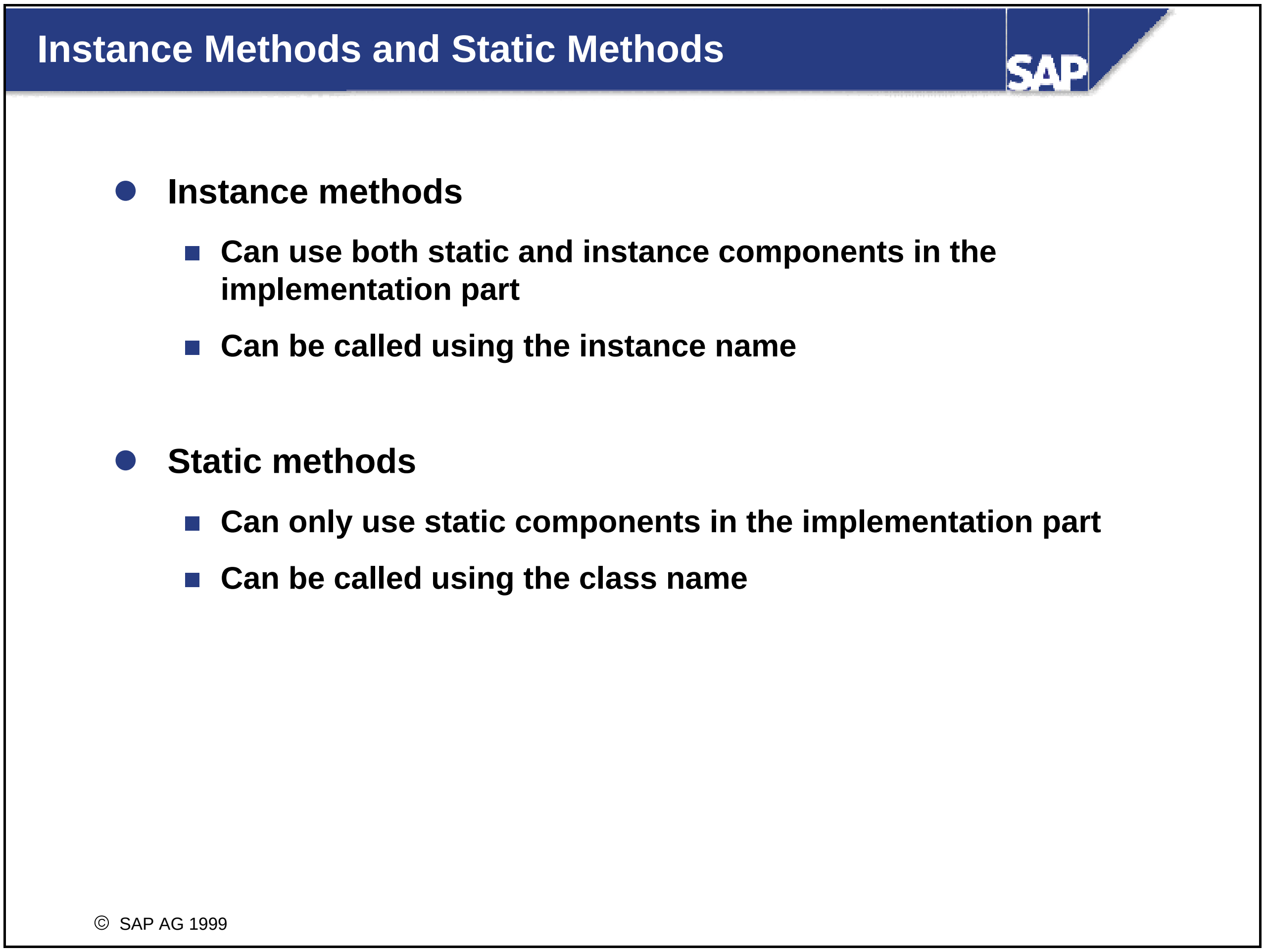
Methods and Visibility

- Public methods
  - Can be called from outside the class
- Private methods
  - Can only be called within the class

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS: set_name importing
                      im_name TYPE string.
  PRIVATE SECTION.
    METHODS: init_name.
    DATA: name TYPE string.
ENDCLASS.
```

```
CLASS lcl_airplane IMPLEMENTATION.
  METHOD init_name.
    name = ‘No Name‘.
  ENDMETHOD.
  METHOD set_name.
    IF im_name IS INITIAL.
*      Calling init_name
       ...
    ELSE. name = im_name. ENDIF.
  ENDMETHOD.
ENDCLASS.
```

Methods, like attributes, must be assigned to a visibility area. This determines whether the methods can be called from outside the class or only from within the class.

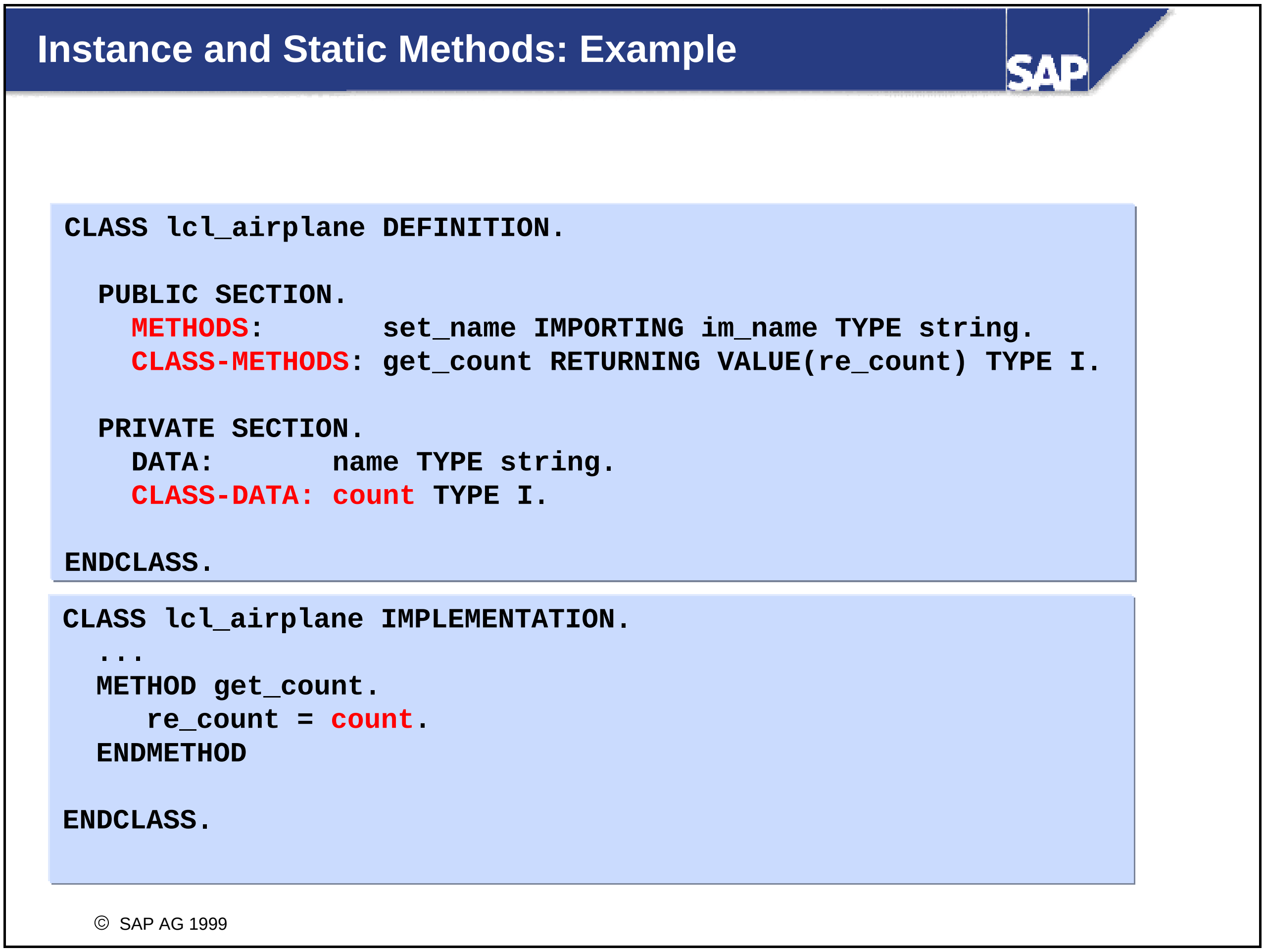


Instance Methods and Static Methods

- Instance methods
  - Can use both static and instance components in the implementation part
  - Can be called using the instance name
- Static methods
  - Can only use static components in the implementation part
  - Can be called using the class name

Static methods are defined on the class level. They are similar to instance methods, but with the restriction that they can only use static components (such as static attributes) in the implementation part. This means that static methods do not need instances and can therefore be called from anywhere. They are defined using the CLASS-METHODS statement, and they are bound by the same syntax and parameter rules as instance methods.

The term “class method” is common, but the official term in ABAP Objects (as in C++, Java) is “static method”. This course uses the term “static method”.

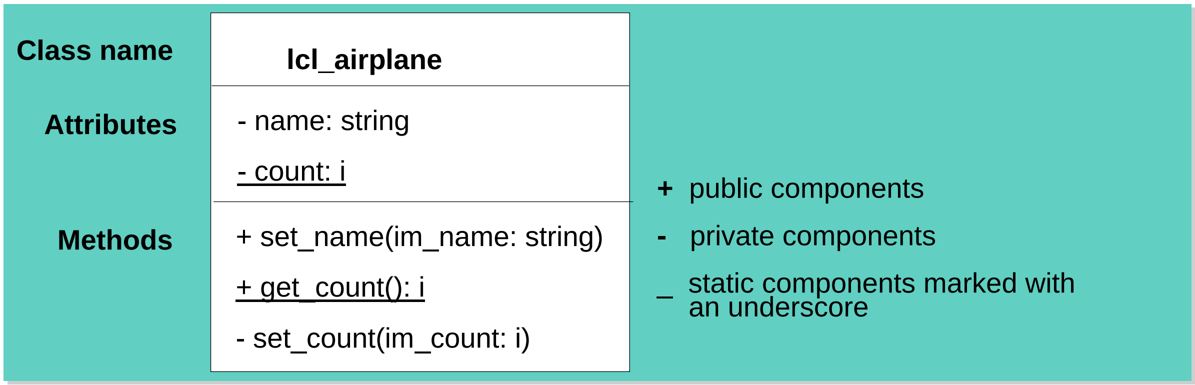


Instance and Static Methods: Example

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    METHODS:       set_name IMPORTING im_name TYPE string.
    CLASS-METHODS: get_count RETURNING VALUE(re_count) TYPE I.

  PRIVATE SECTION.
    DATA:       name TYPE string.
    CLASS-DATA: count TYPE I.

ENDCLASS.
```

```
CLASS lcl_airplane IMPLEMENTATION.
  ...
  METHOD get_count.
     re_count = count.
  ENDMETHOD

ENDCLASS.
```

Attributes and Methods in UML Notation



```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    METHODS:       set_name  IMPORTING im_name TYPE string.
    CLASS-METHODS: get_count RETURNING VALUE(re_count) TYPE I.

  PRIVATE SECTION.
    DATA:       name  TYPE string.
    CLASS-DATA: count TYPE I.
    METHODS: set_count IMPORTING im_count TYPE i.
ENDCLASS.
```

A UML class diagram shows firstly the class name and, underneath that, the class attributes and methods.

The visibility of components in a class is shown in UML using the characters “+” and “-”:

+ public components

- private components

Alternatively, public and private can be prefixed to the methods. The third option for providers of modeling tools in UML is to introduce their own symbols for visibility.

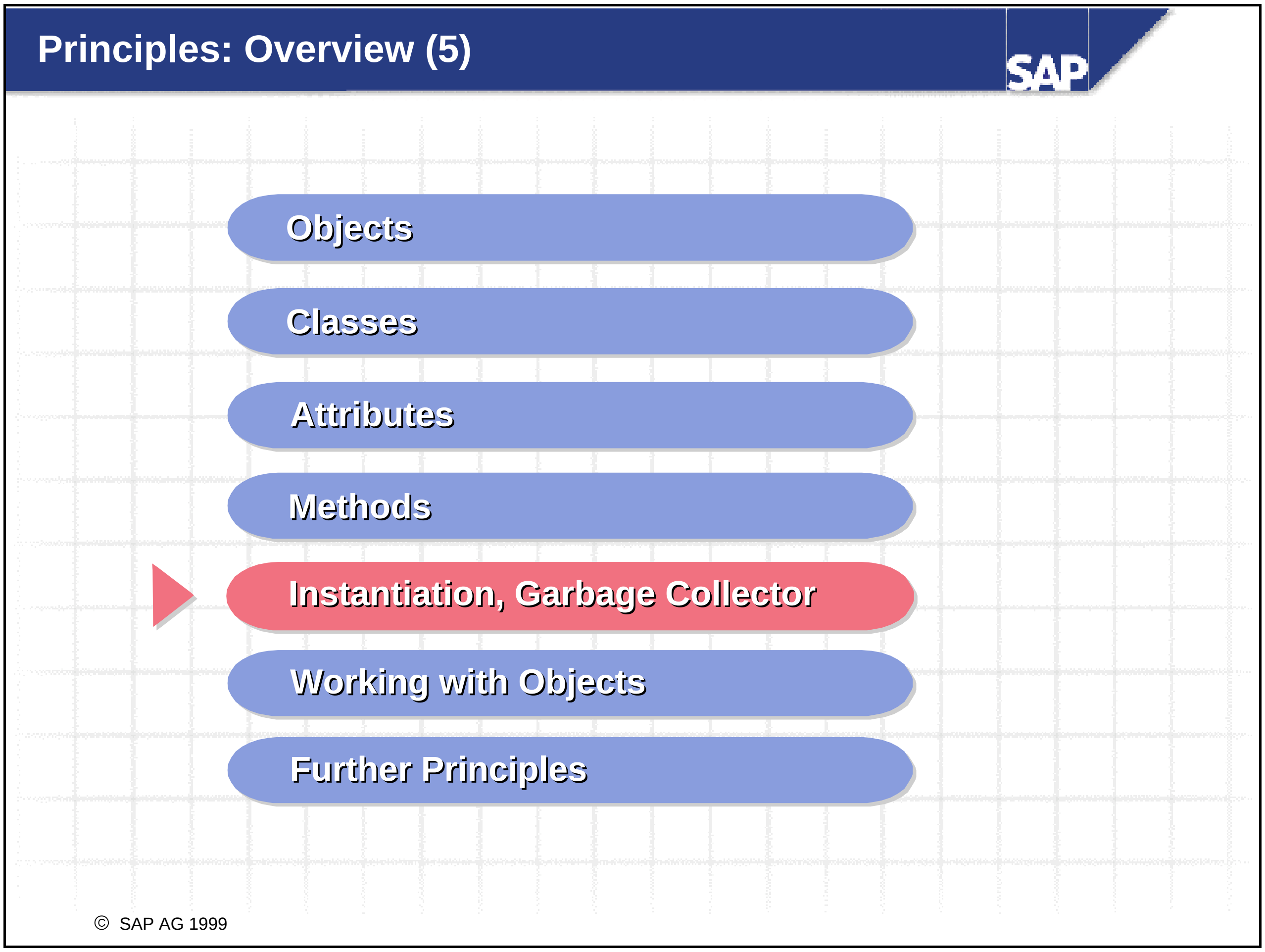
Representation of visibility characteristics is optional and is normally only used for models that are close to implementation.

Static components are marked with an underscore.

The method signature is represented as follows (optional):

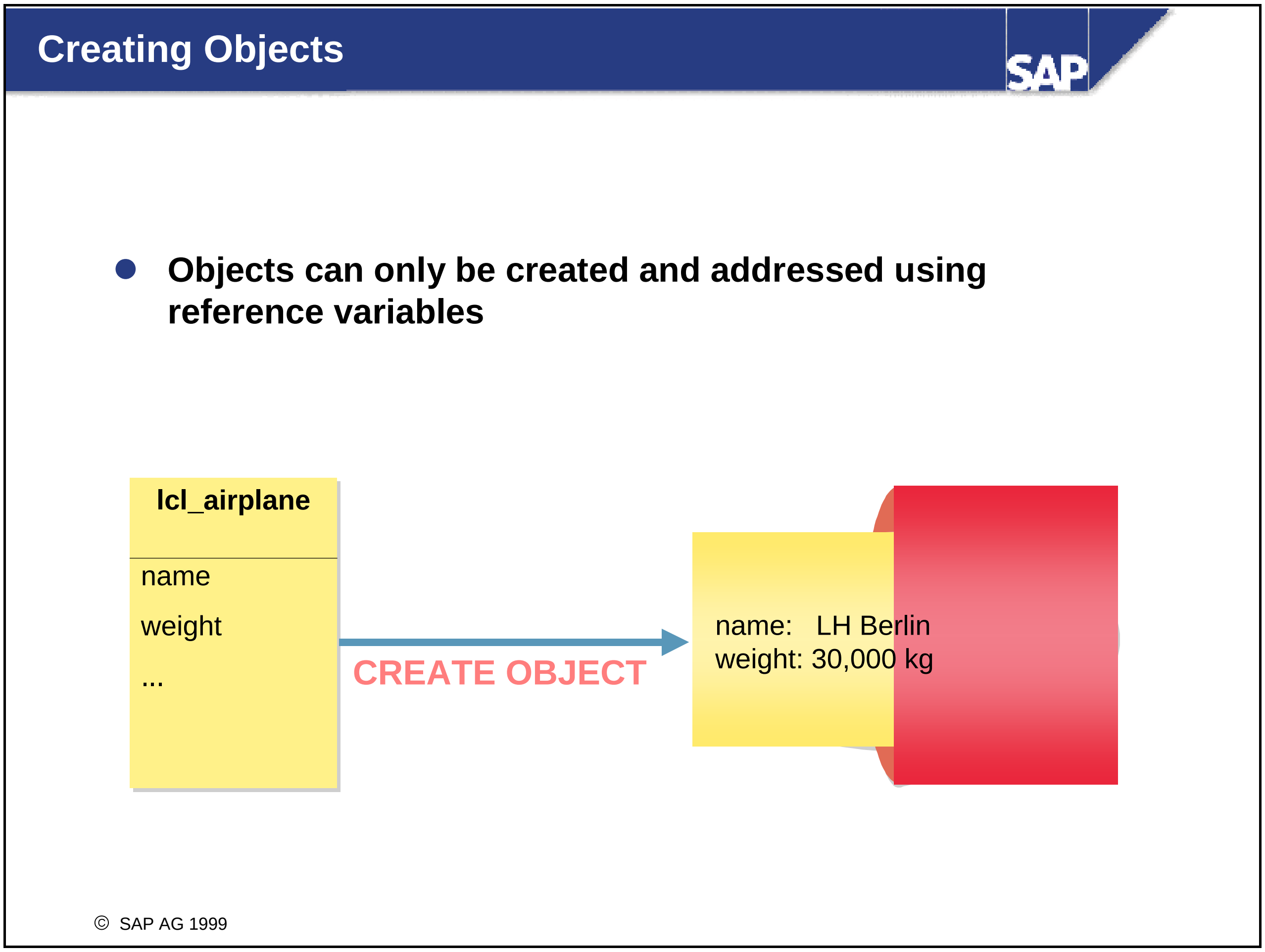
The input and output parameters and the parameters to be changed are shown in brackets.

The return code is separated from the type name by a colon.



Principles: Overview (5)

- Objects
- Classes
- Attributes
- Methods
- **Instantiation, Garbage Collector**
- Working with Objects
- Further Principles



Creating Objects

- **Objects can only be created and addressed using reference variables**

lcl_airplane

name

weight

...

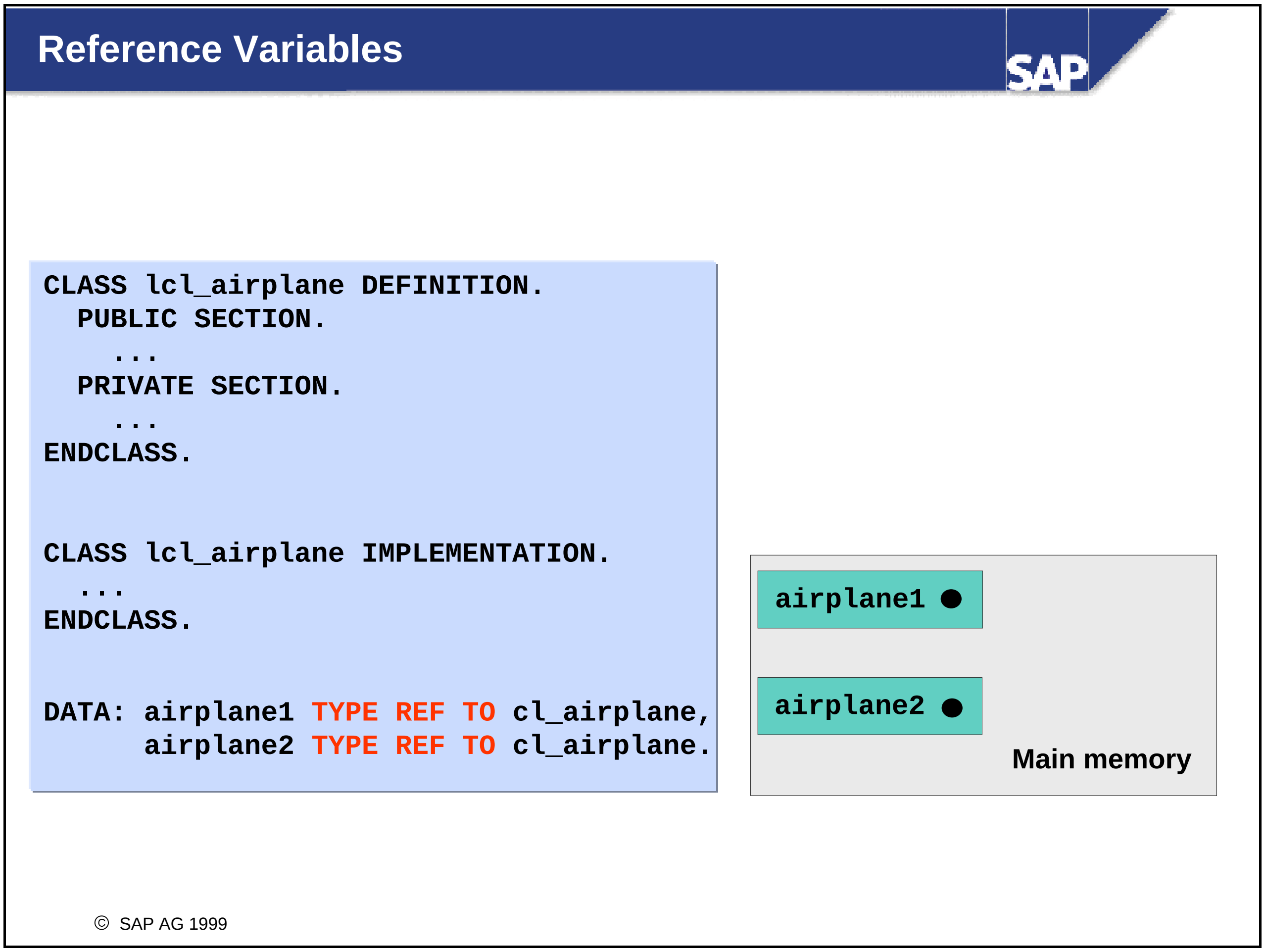
CREATE OBJECT

name: LH Berlin
weight: 30,000 kg

A class contains the generic description of an object. It describes all the characteristics that are common to all the objects in that class. During the program runtime, the class is used to create specific objects (instances). This process is called instantiation.

Example:
The object *LH Berlin* is created during runtime in the main memory by instantiation from the lcl_airplane class.
The lcl_airplane class itself does not exist as an independent runtime object in ABAP Objects.

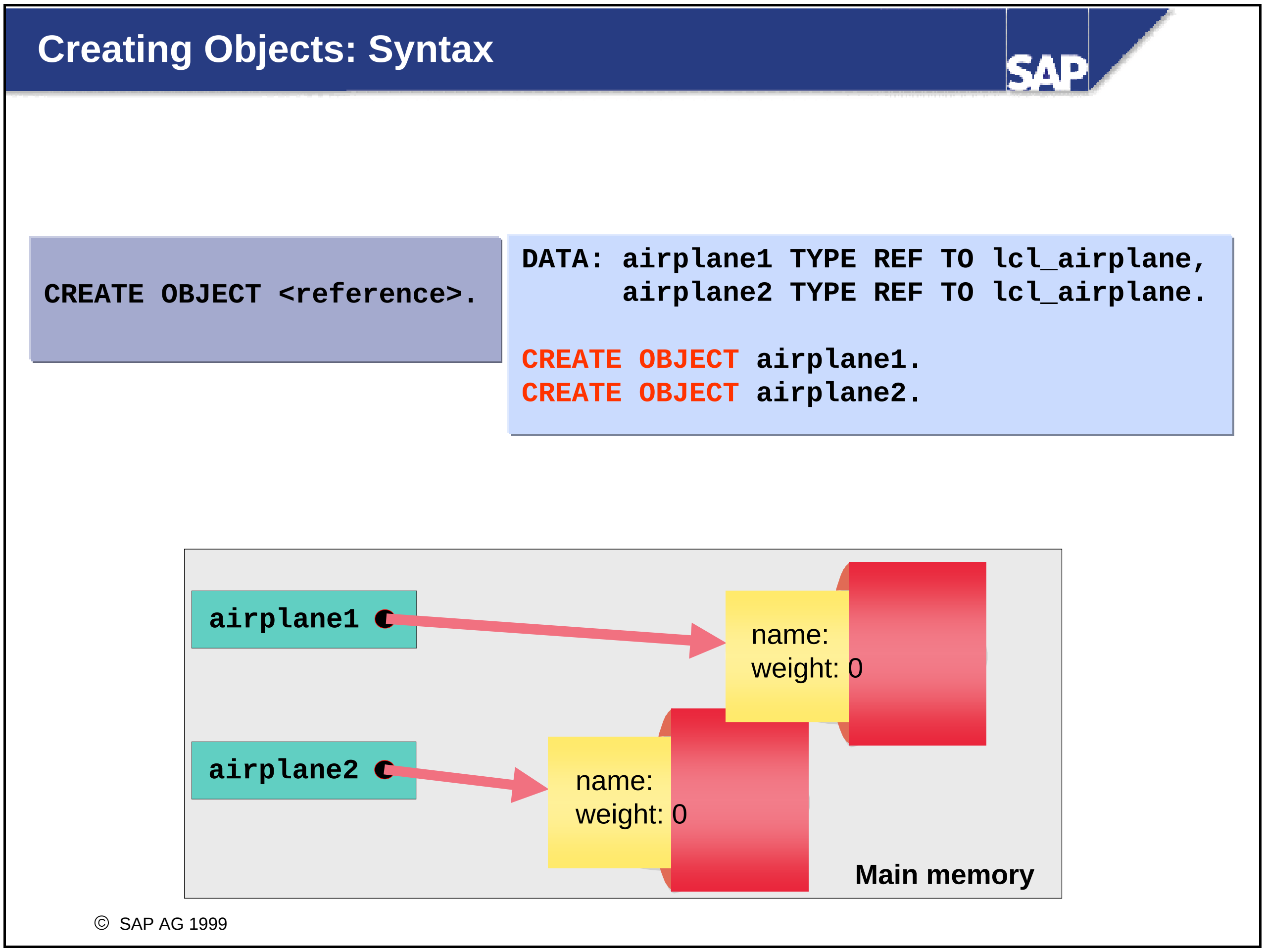
Realization:
Objects are instantiated using the statement: CREATE OBJECT.
During instantiation, the runtime environment dynamically requests main memory space and assigns it to the object.



Reference Variables

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    ...
  PRIVATE SECTION.
    ...
ENDCLASS.


CLASS lcl_airplane IMPLEMENTATION.
  ...
ENDCLASS.


DATA: airplane1 TYPE REF TO cl_airplane,
      airplane2 TYPE REF TO cl_airplane.
```

DATA: airplane1 TYPE REF TO lcl_airplane declares the reference variable airplane1. This acts as a pointer to an object.

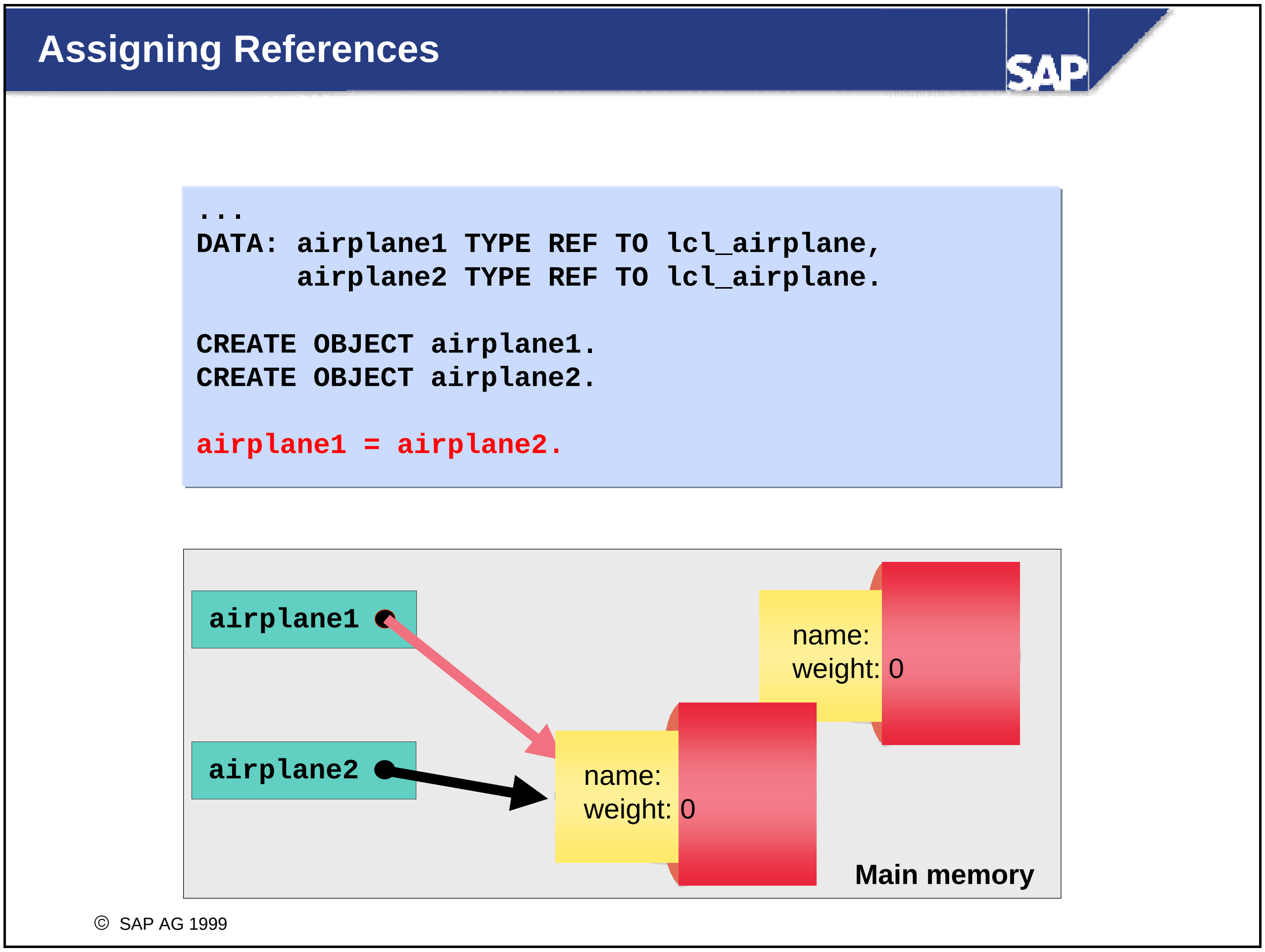


Creating Objects: Syntax

```
CREATE OBJECT <reference>.
```

```
DATA: airplane1 TYPE REF TO lcl_airplane,
      airplane2 TYPE REF TO lcl_airplane.

CREATE OBJECT airplane1.
CREATE OBJECT airplane2.
```

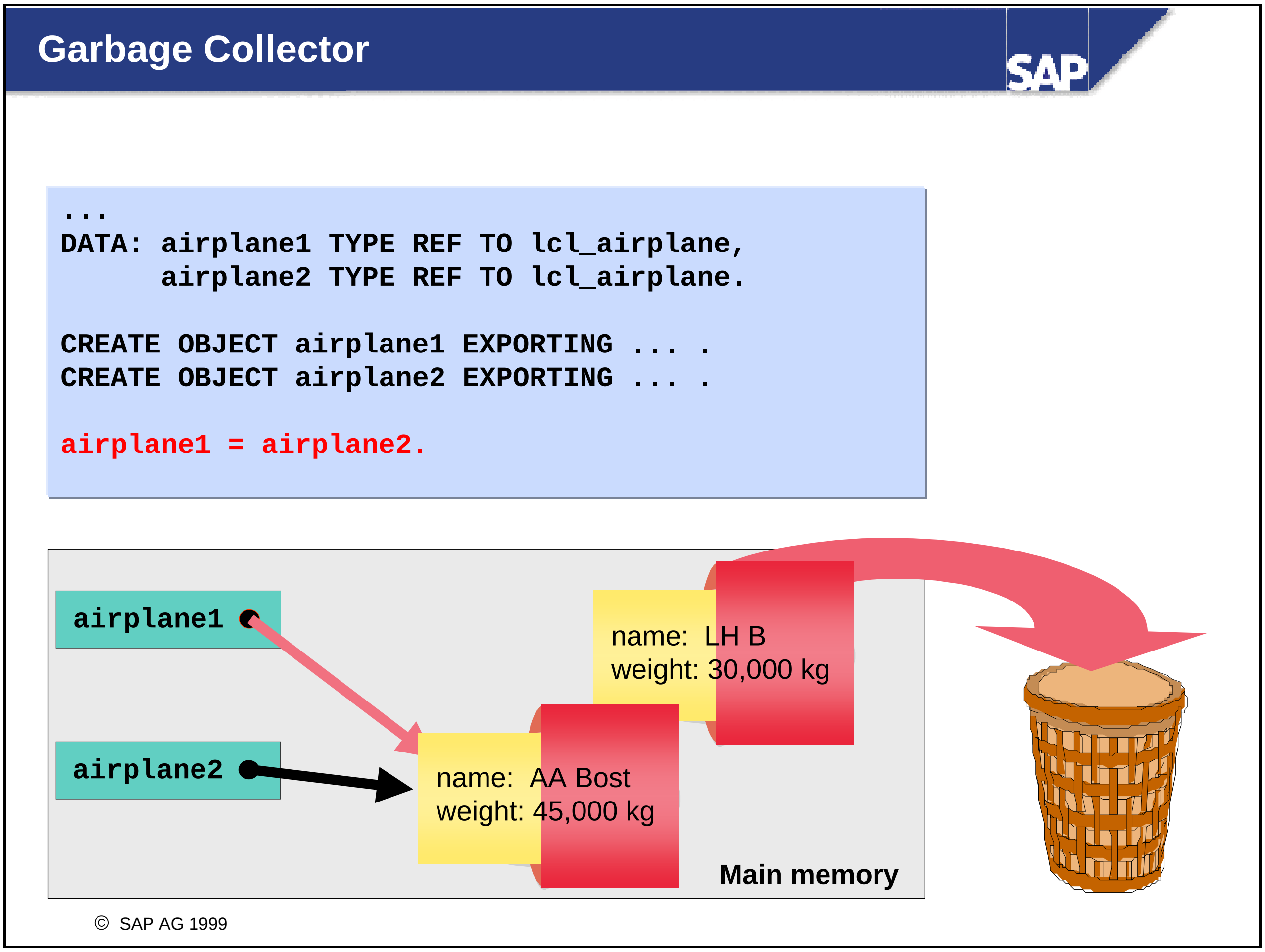
The CREATE OBJECT statement creates an object in the main memory. The attribute values of this object are either initial values or correspond to the VALUE entry.



Assigning References

```
...
DATA: airplane1 TYPE REF TO lcl_airplane,
      airplane2 TYPE REF TO lcl_airplane.

CREATE OBJECT airplane1.
CREATE OBJECT airplane2.

airplane1 = airplane2.
```

Reference variables can also be assigned to each other. The above example shows that once it has been assigned, airplane1 points to the same object as reference airplane2.



Garbage Collector

```
...
DATA: airplane1 TYPE REF TO lcl_airplane,
      airplane2 TYPE REF TO lcl_airplane.

CREATE OBJECT airplane1 EXPORTING ... .
CREATE OBJECT airplane2 EXPORTING ... .

airplane1 = airplane2.
```

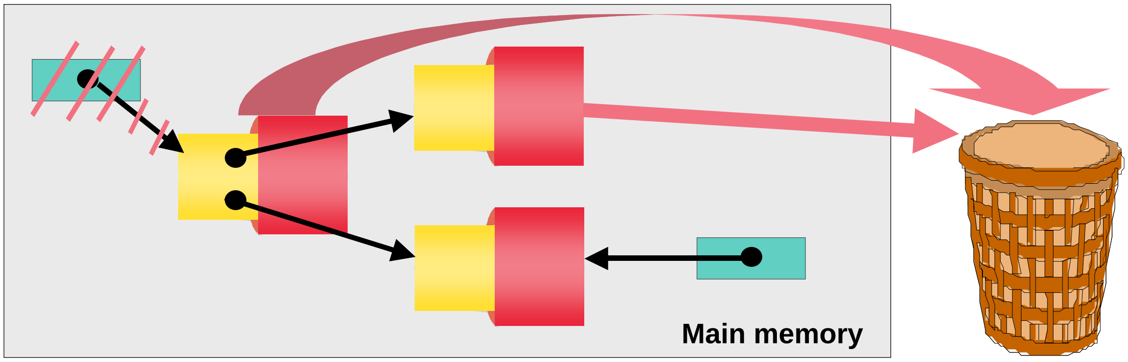
© SAP AG 1999

As soon as no more references point to an object, the Garbage Collector removes it from the memory.

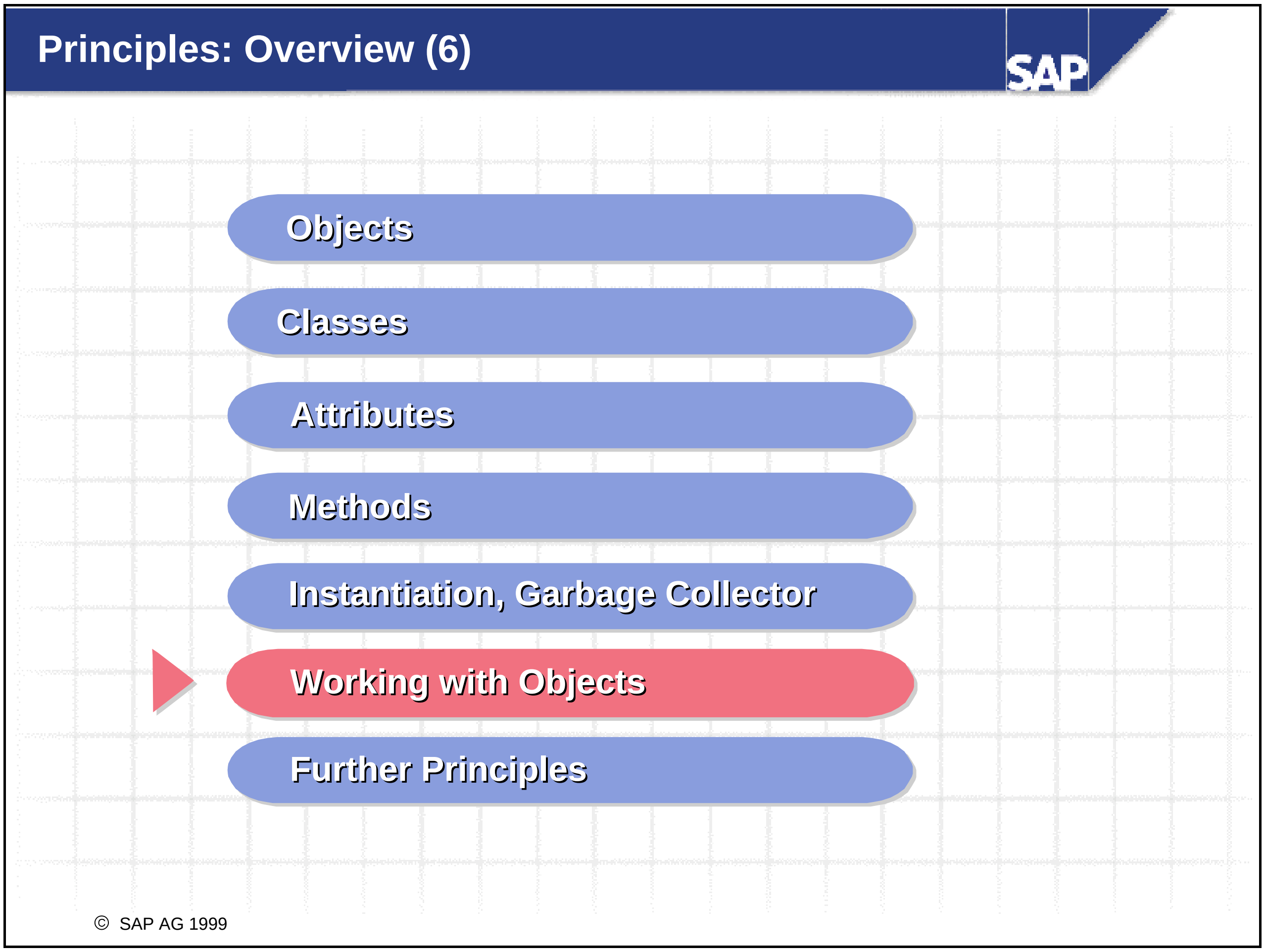
The Garbage Collector is a system routine that automatically deletes objects that can no longer be addressed from the main memory and releases the memory space they occupied.

Garbage Collector: Concept

- All independent references in the global main memory are checked. The references point to active objects, which are marked internally.
- If class or instance attribute references point to other objects, these are also marked.
- Objects that are not marked are deleted from the main memory.



Independent references are references that have not been defined within a class.



Principles: Overview (6)

Objects

Classes

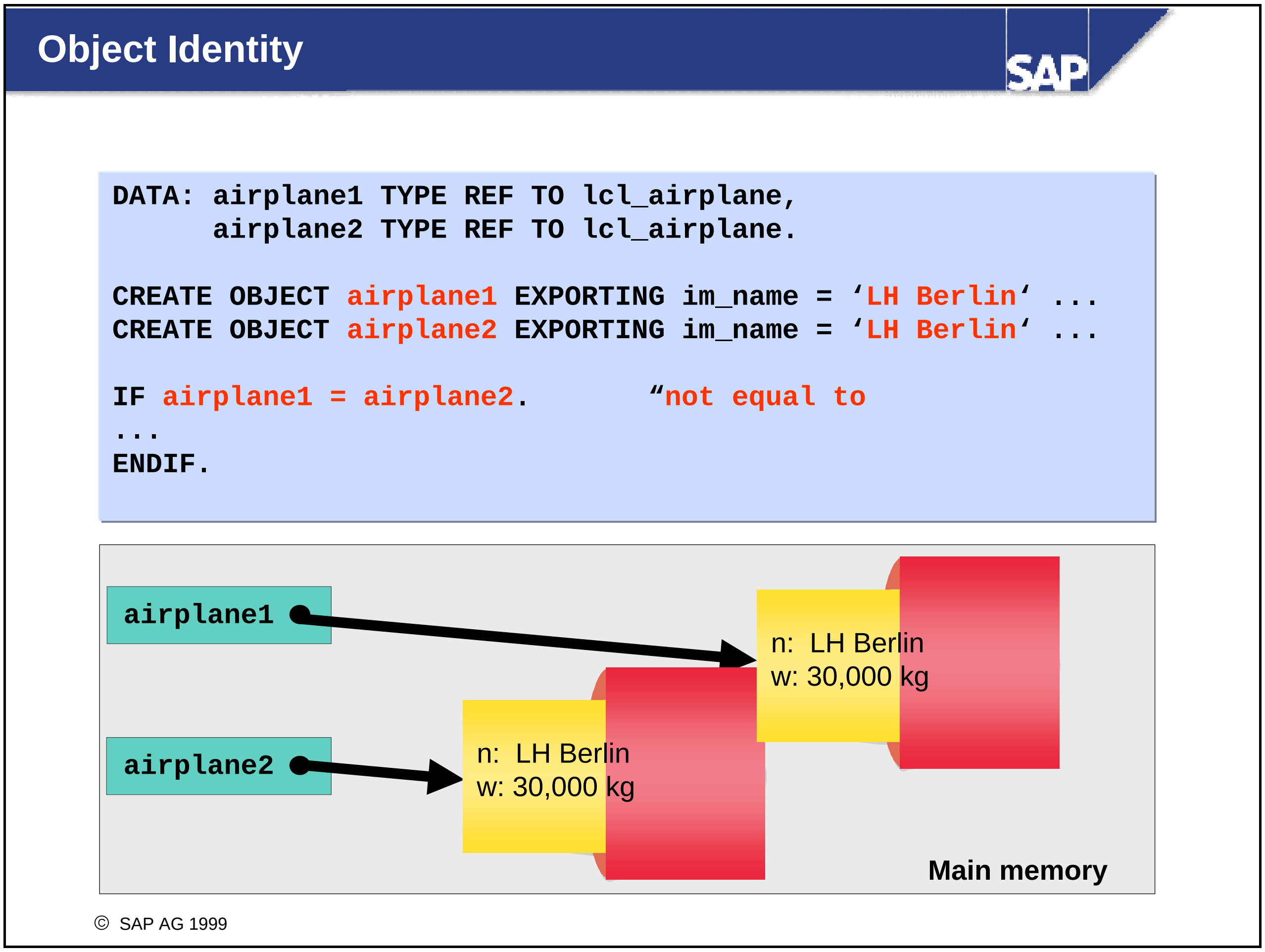
Attributes

Methods

Instantiation, Garbage Collector

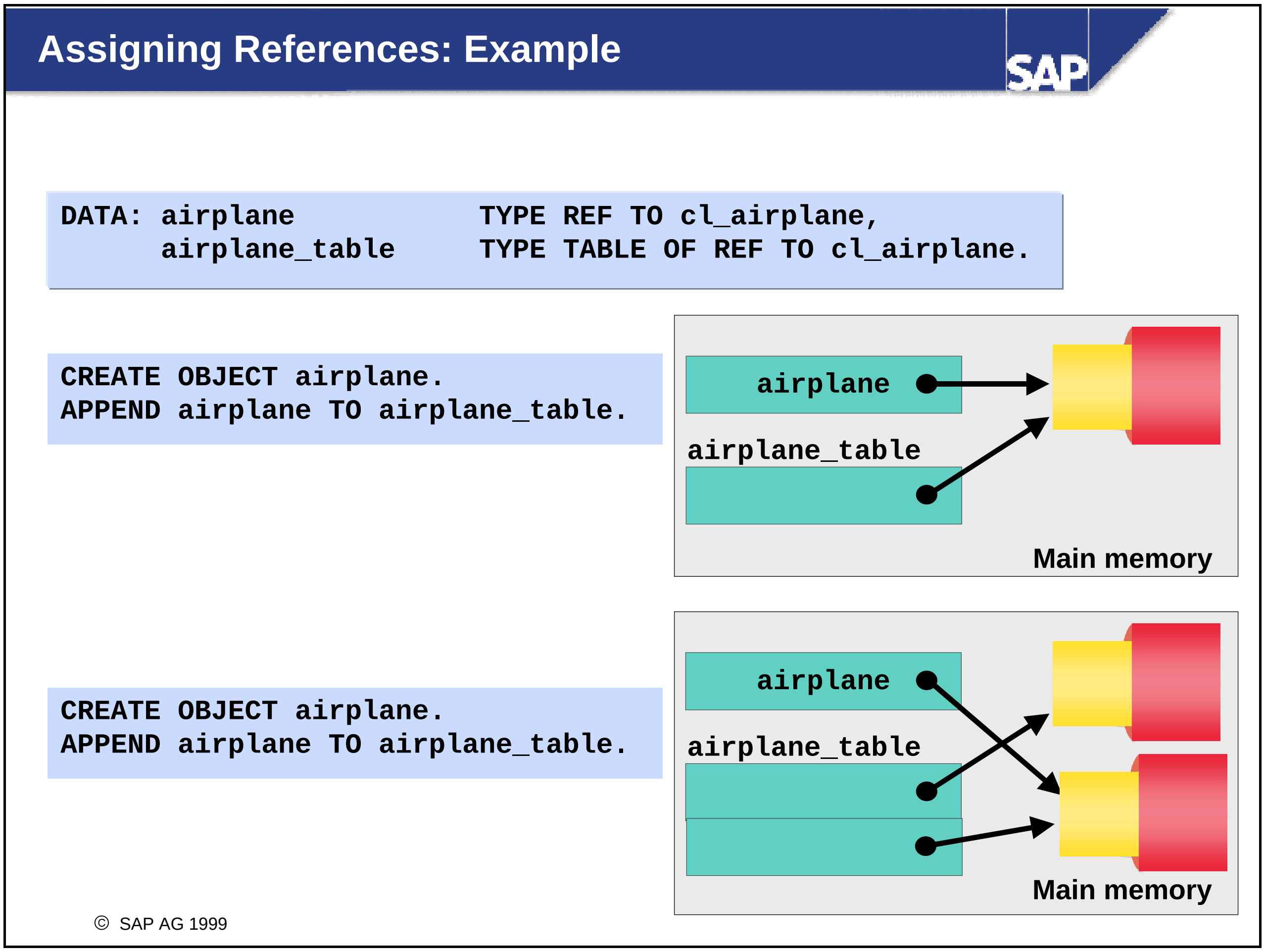
▶ Working with Objects

Further Principles



Object Identity

```
DATA: airplane1 TYPE REF TO lcl_airplane,
      airplane2 TYPE REF TO lcl_airplane.

CREATE OBJECT airplane1 EXPORTING im_name = ‘LH Berlin‘ ...
CREATE OBJECT airplane2 EXPORTING im_name = ‘LH Berlin‘ ...

IF airplane1 = airplane2.       “not equal to
...
ENDIF.
```



Assigning References: Example

```
DATA: airplane          TYPE REF TO cl_airplane,
      airplane_table    TYPE TABLE OF REF TO cl_airplane.
```

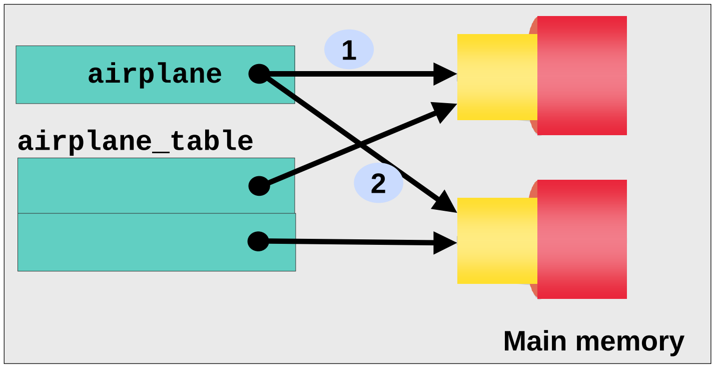
```
CREATE OBJECT airplane.
APPEND airplane TO airplane_table.
```

```
CREATE OBJECT airplane.
APPEND airplane TO airplane_table.
```

If you want to keep several objects from the same class in your program, you can define an internal table, which might, for example, only consist of one column with the object references for this class.

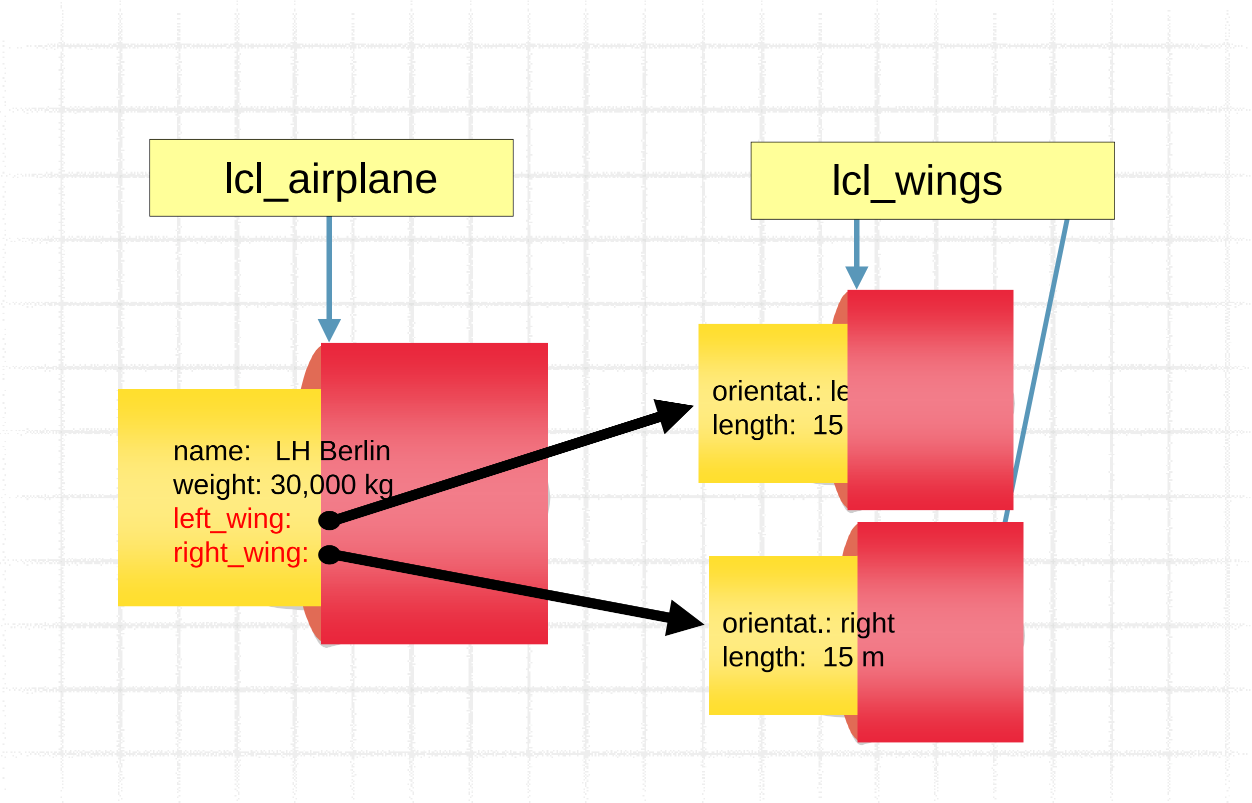
Assigning References: Example (2)

```
LOOP AT TO airplane_table INTO airplane.

*  work with the current instance

ENDLOOP.
```



You can work with the objects using the internal table within the loop.

Object References as Attributes



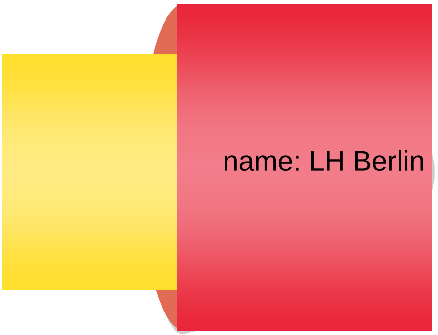
© SAP AG 1999

If a class defines object references to a second class as attributes (in the above example: left_wing, right_wing), then only these object references will be stored in the objects belonging to that class. The objects in the second class have their own identity.

External Access to Public Attributes

Instance attribute:
<reference>-><instance_attribute>

Class attribute:
<classname>=><class_attribute>

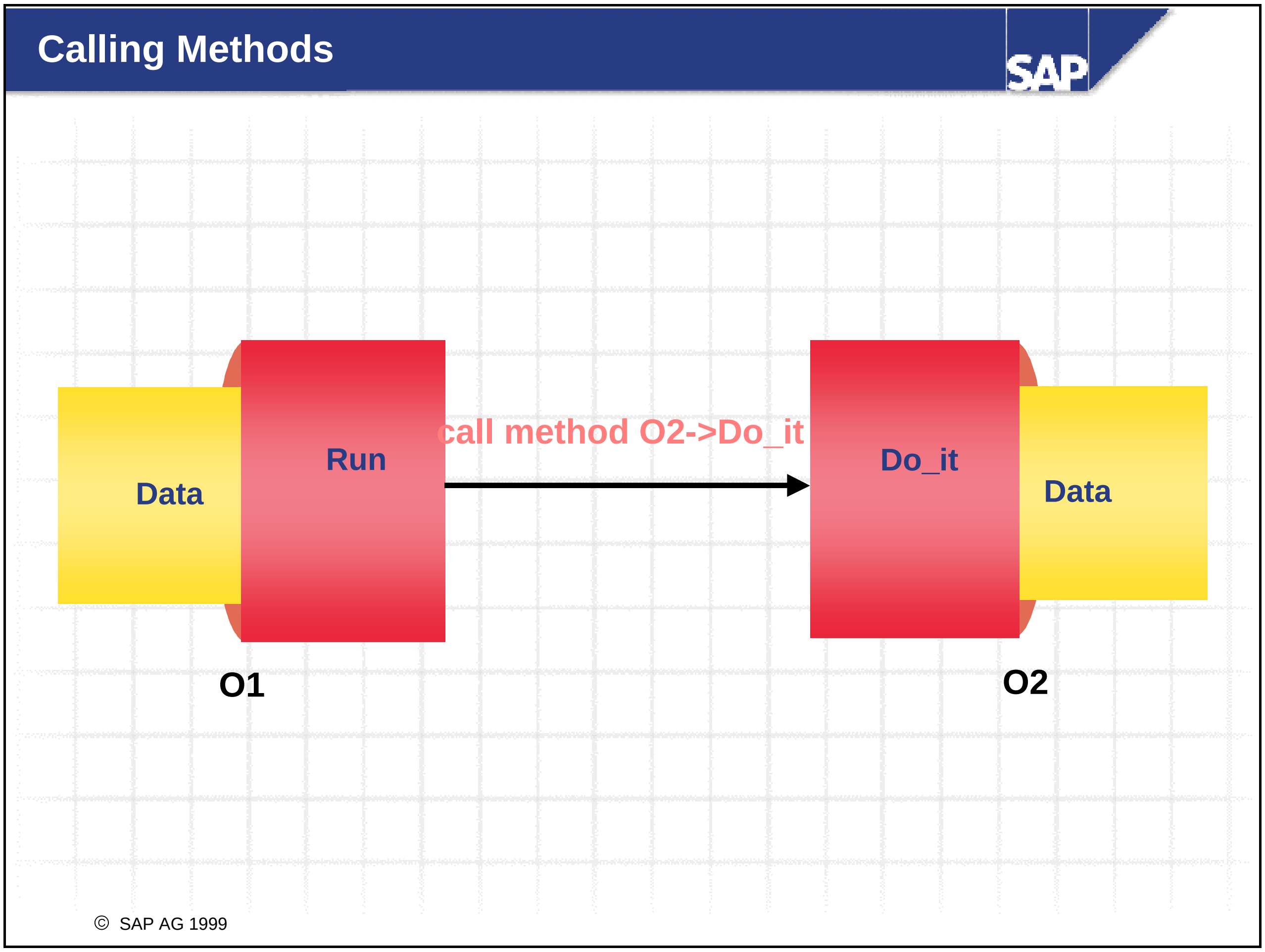


```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    DATA: name    TYPE string READ-ONLY.
    CLASS-DATA: count TYPE I READ-ONLY.
    ...
ENDCLASS.
...

DATA: airplane1 TYPE REF TO lcl_airplane.
DATA: airplane_name TYPE STRING,
      n_o_airplanes TYPE i.
...

airplane_name = airplane1->name.
n_o_airplanes = lcl_airplane=>count.
```

Public attributes can be accessed from outside the class in various ways:

Static attributes are accessed using <classname>=><class_attribute>.

Instance attributes are accessed using <instance>-><instance_attribute>.



Calling Methods

Data

Run

call method O2->Do_it

Do_it

Data

O1

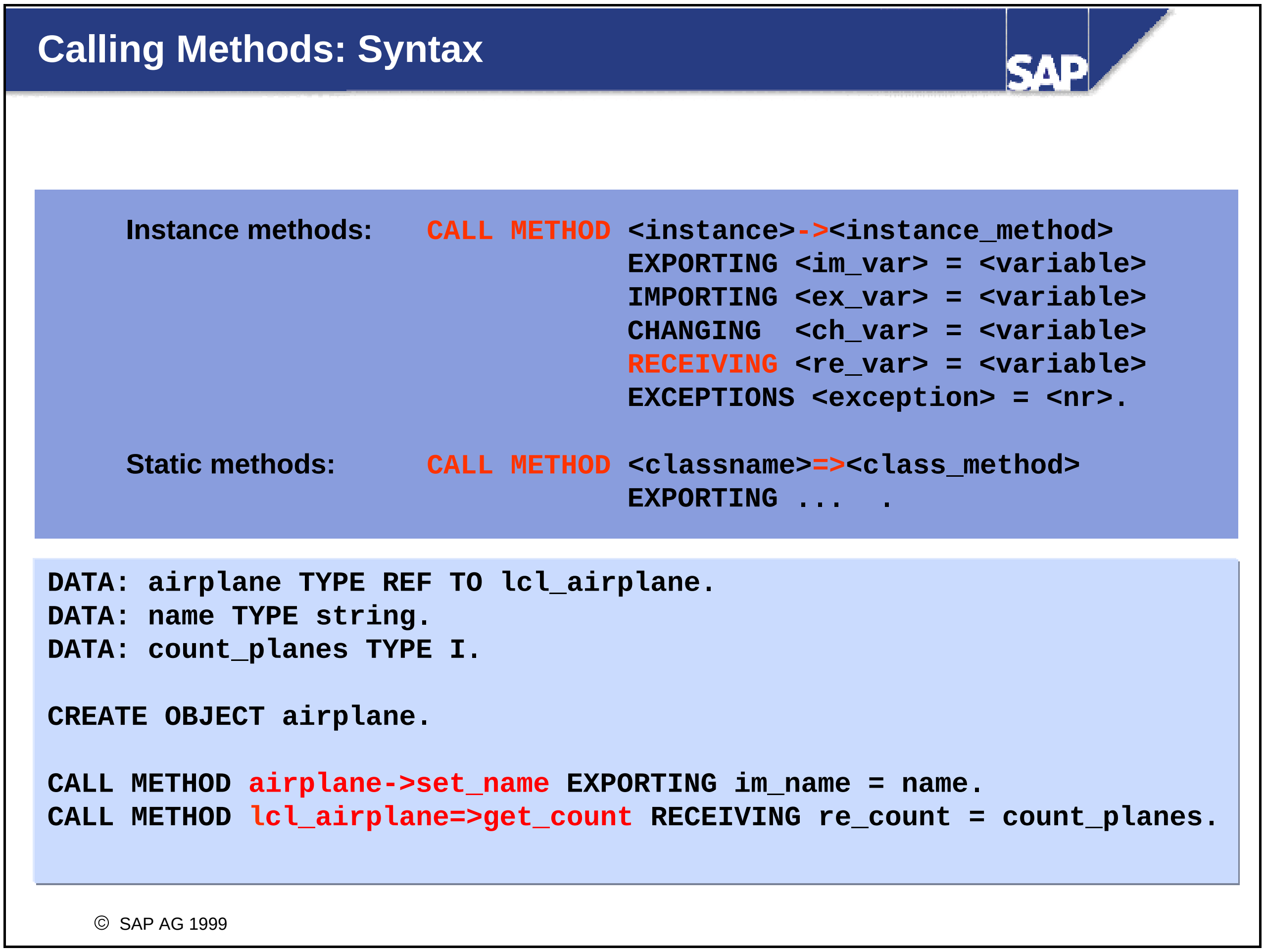
O2

Every object behaves in a certain way. This behavior is determined by its methods. There are three types of method:

1. Methods that cause behavior and do not pass values
2. Methods that pass a value
3. Methods that pass or change several values

An object that requires services from 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 is used below as a synonym for operation and message.



Calling Methods: Syntax

```
Instance methods:   CALL METHOD <instance>-><instance_method>
                                EXPORTING <im_var> = <variable>
                                IMPORTING <ex_var> = <variable>
                                CHANGING  <ch_var> = <variable>
                                RECEIVING <re_var> = <variable>
                                EXCEPTIONS <exception> = <nr>.

Static methods:     CALL METHOD <classname>=><class_method>
                                EXPORTING ...   .
```

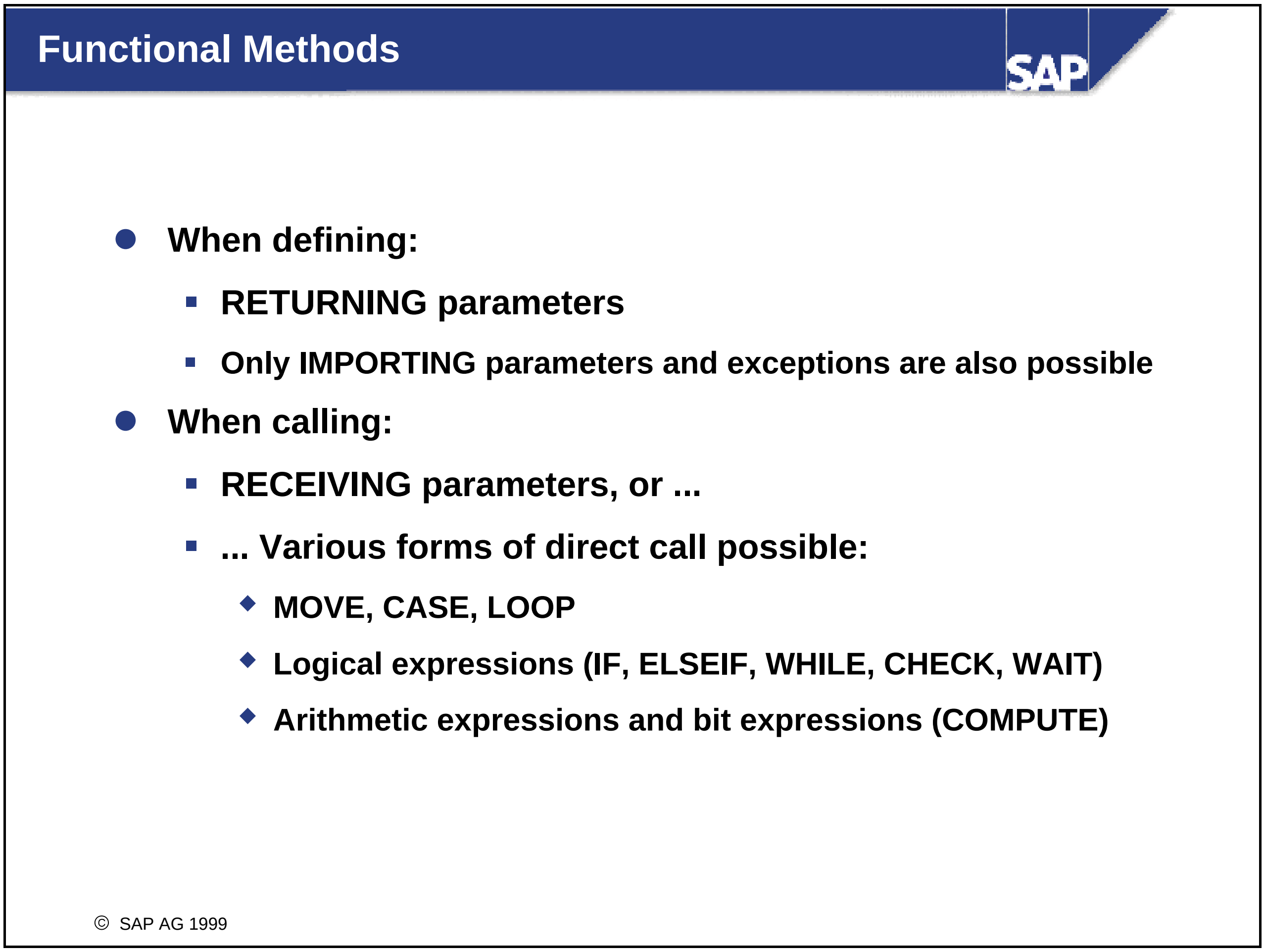
```
DATA: airplane TYPE REF TO lcl_airplane.
DATA: name TYPE string.
DATA: count_planes TYPE I.

CREATE OBJECT airplane.

CALL METHOD airplane->set_name EXPORTING im_name = name.
CALL METHOD lcl_airplane=>get_count RECEIVING re_count = count_planes.
```

Public methods can be called from outside the class in a number of ways:

Instance methods are called using CALL METHOD <reference>-><instance_method>.

Static methods are called using CALL METHOD <classname>=><class_method>.
Static methods are addressed by class name, since they do not need instances.

Note:
If you are calling a static method from within the class, you can omit the class name.
When calling an instance method from within another instance method, you can omit the instance name. The method is automatically executed for the current object.



Functional Methods

- When defining:
  - RETURNING parameters
  - Only IMPORTING parameters and exceptions are also possible
- When calling:
  - RECEIVING parameters, or ...
  - ... Various forms of direct call possible:
    - MOVE, CASE, LOOP
    - Logical expressions (IF, ELSEIF, WHILE, CHECK, WAIT)
    - Arithmetic expressions and bit expressions (COMPUTE)

Methods that have a RETURNING parameter are described as functional methods. These methods cannot have EXPORTING or CHANGING parameters, but has many (or as few) IMPORTING parameters and EXCEPTIONS as required.

Functional methods can be used directly in various expressions (although EXCEPTIONS are not catchable at the moment - you must use the long form of the method call):

in logical expressions (IF, ELSEIF, WHILE, CHECK, WAIT)

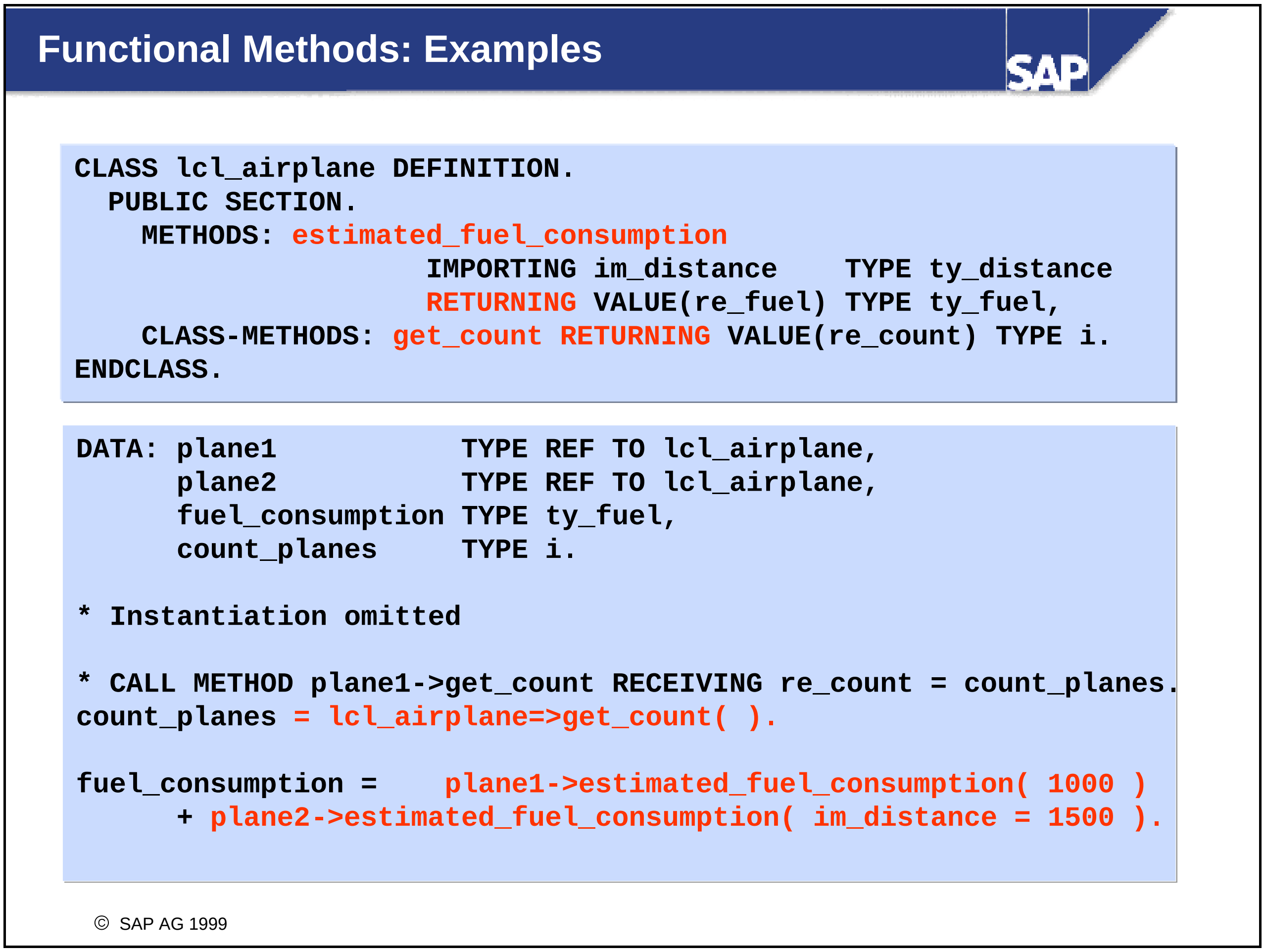
in the CASE statement (CASE, WHEN)

in the LOOP statement

in arithmetic expressions (COMPUTE)

in Bit expressions (COMPUTE)

in the MOVE statement.



Functional Methods: Examples

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS: estimated_fuel_consumption
                     IMPORTING im_distance    TYPE ty_distance
                     RETURNING VALUE(re_fuel) TYPE ty_fuel,
    CLASS-METHODS: get_count RETURNING VALUE(re_count) TYPE i.
ENDCLASS.
```

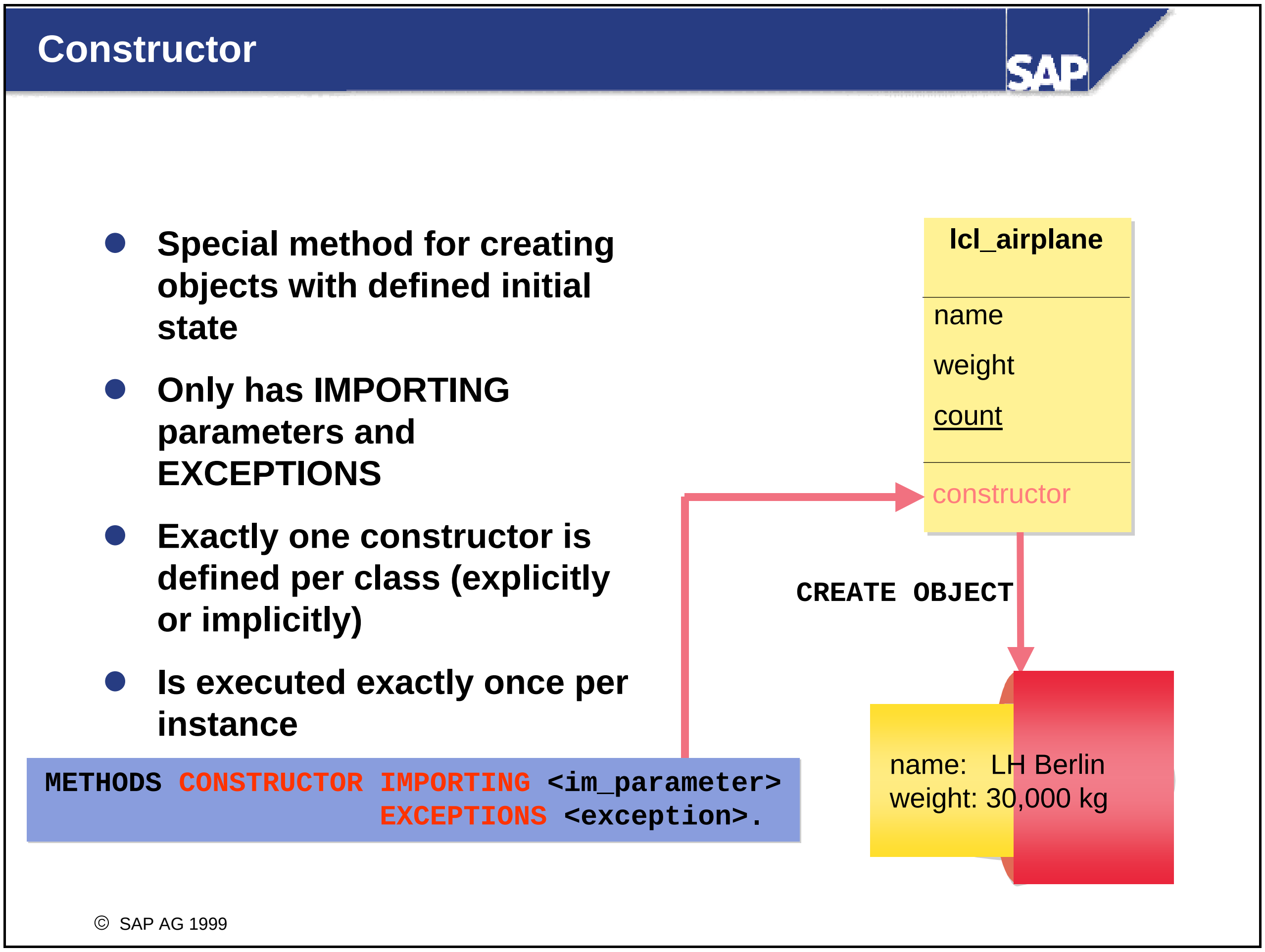
```
DATA: plane1           TYPE REF TO lcl_airplane,
      plane2           TYPE REF TO lcl_airplane,
      fuel_consumption TYPE ty_fuel,
      count_planes     TYPE i.

* Instantiation omitted

* CALL METHOD plane1->get_count RECEIVING re_count = count_planes.
count_planes = lcl_airplane=>get_count( ).

fuel_consumption =    plane1->estimated_fuel_consumption( 1000 )
      + plane2->estimated_fuel_consumption( im_distance = 1500 ).
```

The syntax for instance methods (analogous to static methods) is as follows, depending on the number of IMPORTING parameters :

no IMPORTING parameters: ref->func_method( )

exactly 1 IMPORTING parameter : ref->func_method(p1) oder
ref->func_method(im_1 = p1)

several IMPORTING parameters : ref->func_method(im_1 = p1 im_2 = p2)



Constructor

- Special method for creating objects with defined initial state
- Only has IMPORTING parameters and EXCEPTIONS
- Exactly one constructor is defined per class (explicitly or implicitly)
- Is executed exactly once per instance

```
METHODS CONSTRUCTOR IMPORTING <im_parameter>
                    EXCEPTIONS <exception>.
```

The constructor is a special (instance) method in a class and is always named CONSTRUCTOR. The following rules apply:

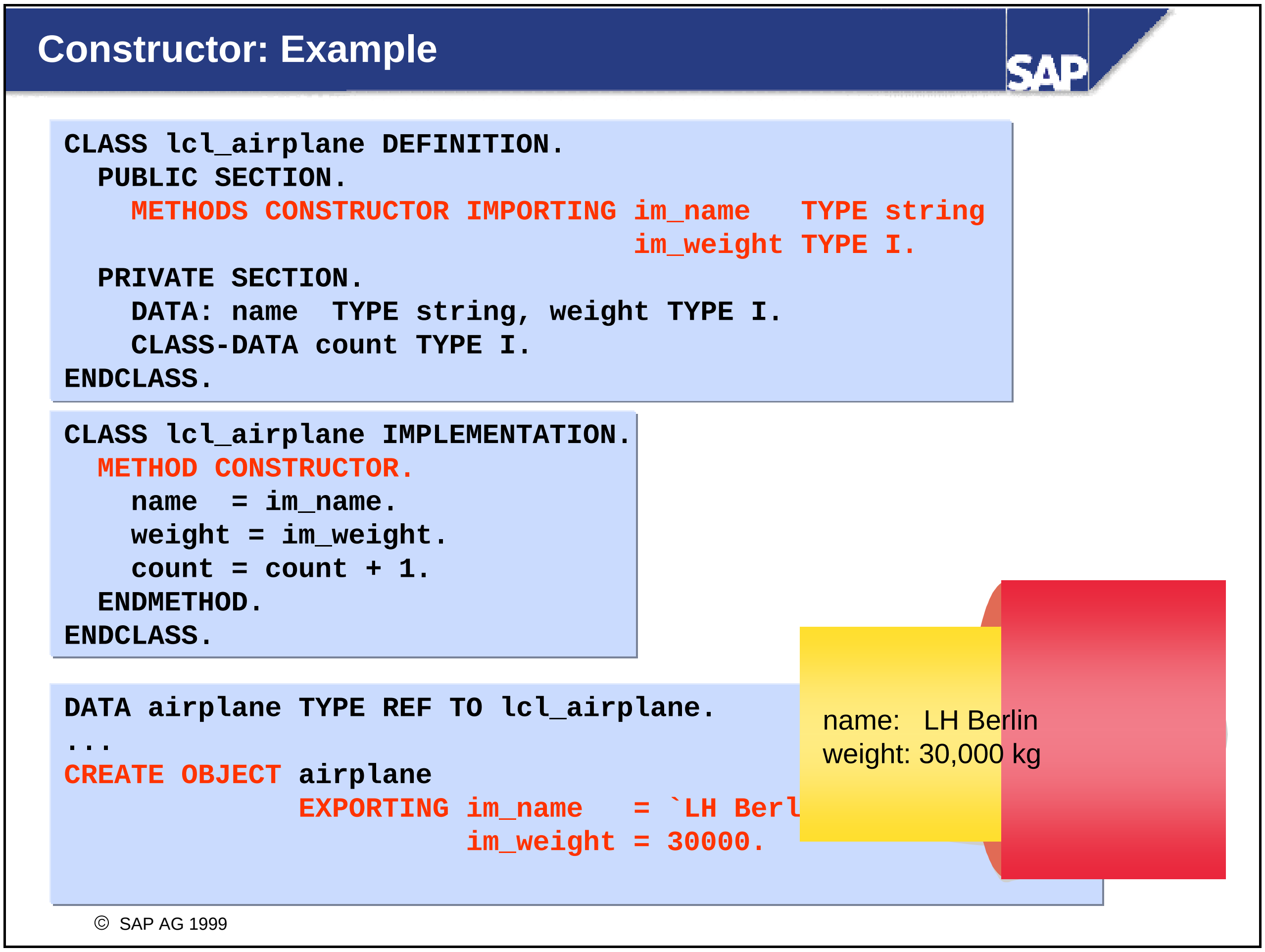
Each class has exactly one constructor.

The constructor does not need to be defined if no implementation is defined.

The constructor is automatically called during runtime within the CREATE OBJECT statement.

If you need to implement the constructor, then you must define and implement it in the PUBLIC SECTION.

When EXCEPTIONS are triggered in the constructor, instances are **not** created (as of 4.6a), so no main memory space is taken up.



Constructor: Example

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS CONSTRUCTOR IMPORTING im_name   TYPE string
                                  im_weight TYPE I.
  PRIVATE SECTION.
    DATA: name  TYPE string, weight TYPE I.
    CLASS-DATA count TYPE I.
ENDCLASS.
```

```
CLASS lcl_airplane IMPLEMENTATION.
  METHOD CONSTRUCTOR.
    name   = im_name.
    weight = im_weight.
    count = count + 1.
  ENDMETHOD.
ENDCLASS.
```

```
DATA airplane TYPE REF TO lcl_airplane.
...
CREATE OBJECT airplane
              EXPORTING im_name   = `LH Berl
                        im_weight = 30000.
```

name: LH Berlin
weight: 30,000 kg

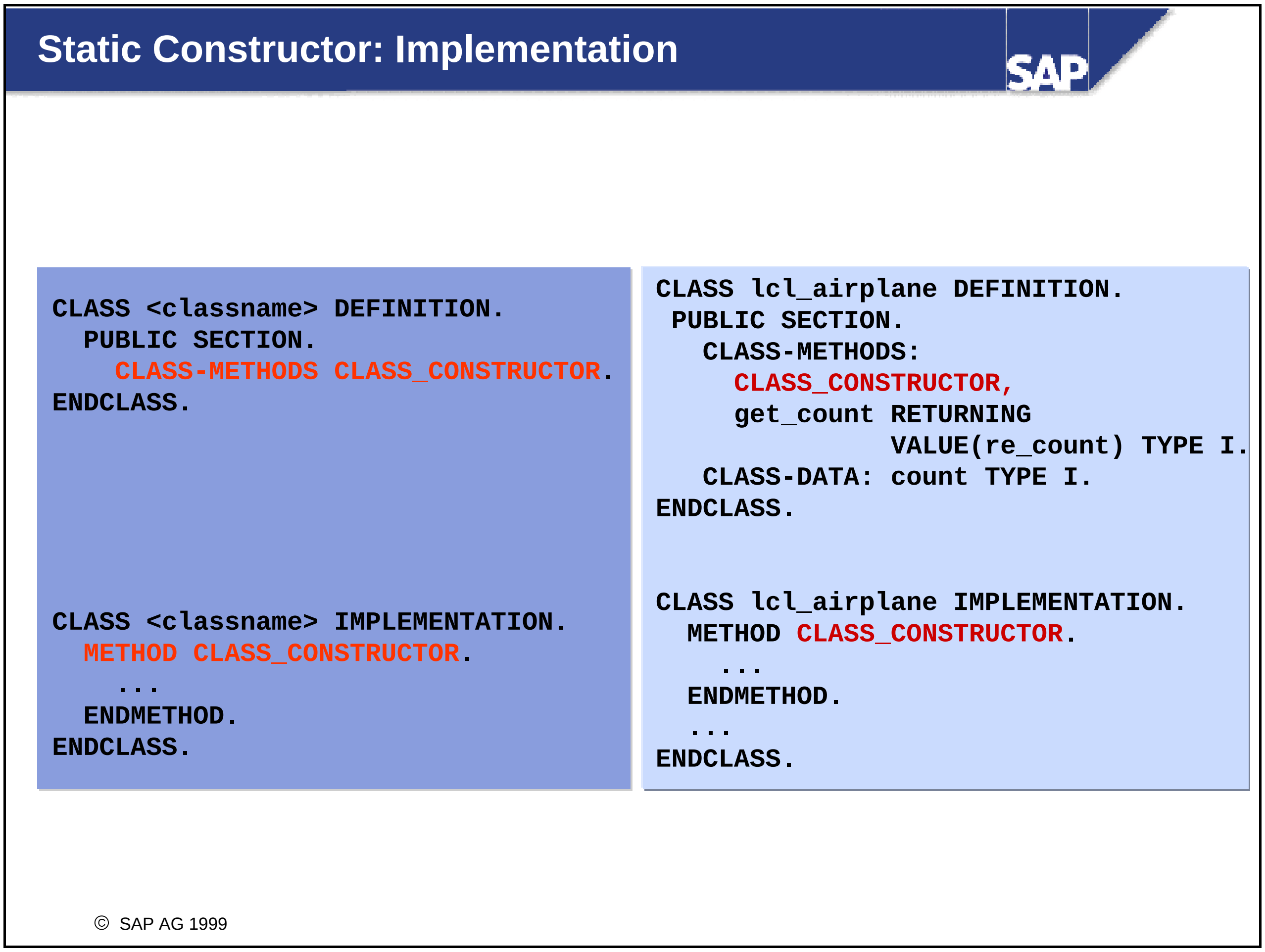
You need to implement the constructor when, for example

You need to allocate (external) resources

You need to initialize attributes that cannot be covered by the VALUE supplement to the DATA statement

You need to modify static attributes

You cannot normally call the constructor explicitly.



Static Constructor: Implementation

```
CLASS <classname> DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS CLASS_CONSTRUCTOR.
ENDCLASS.



CLASS <classname> IMPLEMENTATION.
  METHOD CLASS_CONSTRUCTOR.
    ...
  ENDMETHOD.
ENDCLASS.
```

```
CLASS lcl_airplane DEFINITION.
 PUBLIC SECTION.
   CLASS-METHODS:
     CLASS_CONSTRUCTOR,
     get_count RETURNING
               VALUE(re_count) TYPE I.
   CLASS-DATA: count TYPE I.
ENDCLASS.


CLASS lcl_airplane IMPLEMENTATION.
  METHOD CLASS_CONSTRUCTOR.
    ...
  ENDMETHOD.
  ...
ENDCLASS.
```

The static constructor is a special static method in a class and is always named CLASS_CONSTRUCTOR. It is executed precisely once per program. The static constructor of class <classname> is called automatically before the class is first accessed, that is, before any of the following actions are executed:

Creating an instance in the class using CREATE OBJECT obj, where obj has the data type REF TO <classname>.

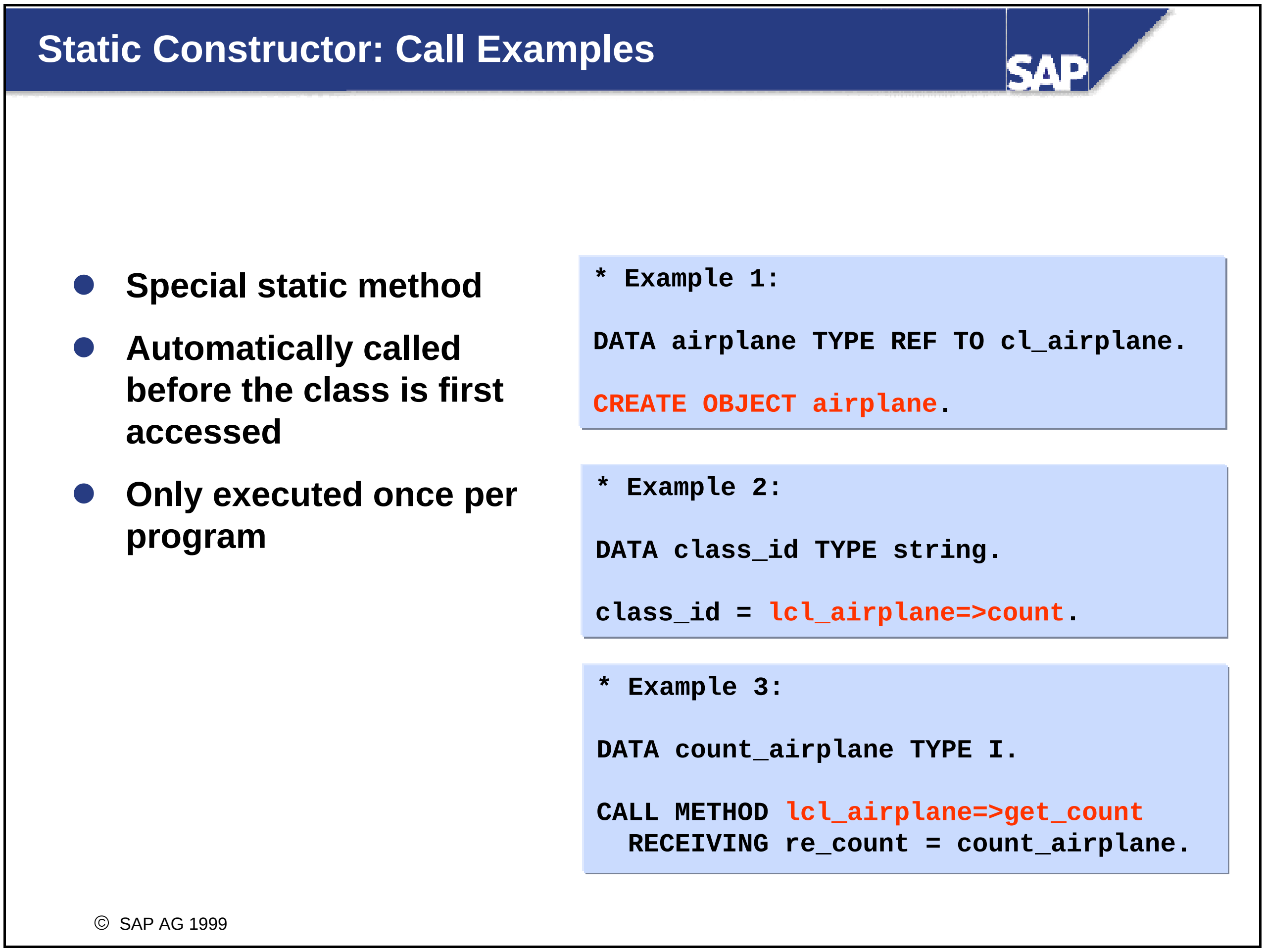
Addressing a static attribute using <classname>=><an_attribute>.

Calling a static attribute using CALL METHOD <classname>=><a_classmethod>.

Registering a static event handler method using SET HANDLER <classname>=><handler_method> for obj.

Registering an event handler method for a static event in class <classname>.

The static constructor cannot be called explicitly.



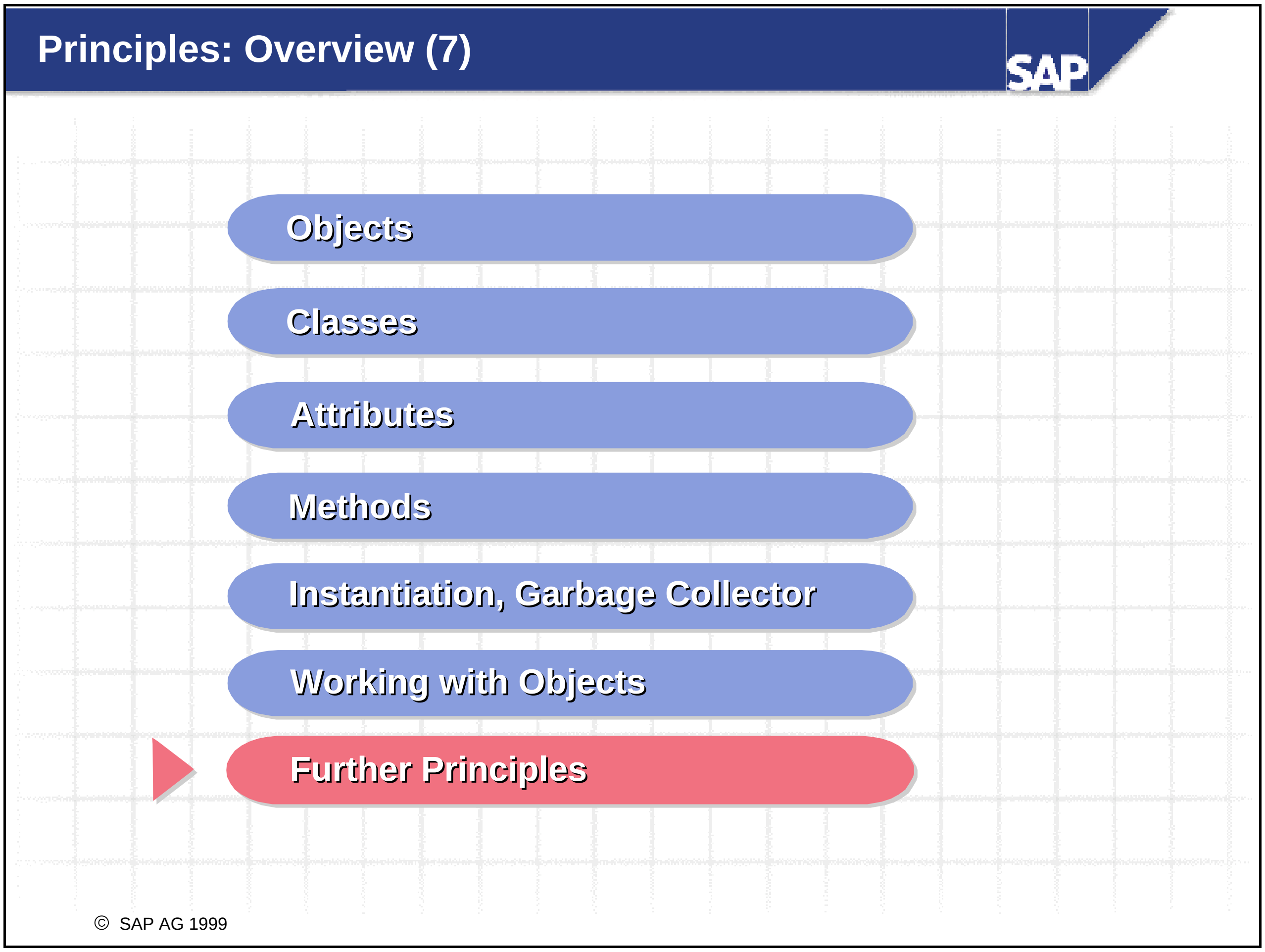
Static Constructor: Call Examples

- Special static method
- Automatically called before the class is first accessed
- Only executed once per program

```
* Example 1:

DATA airplane TYPE REF TO cl_airplane.

CREATE OBJECT airplane.
```

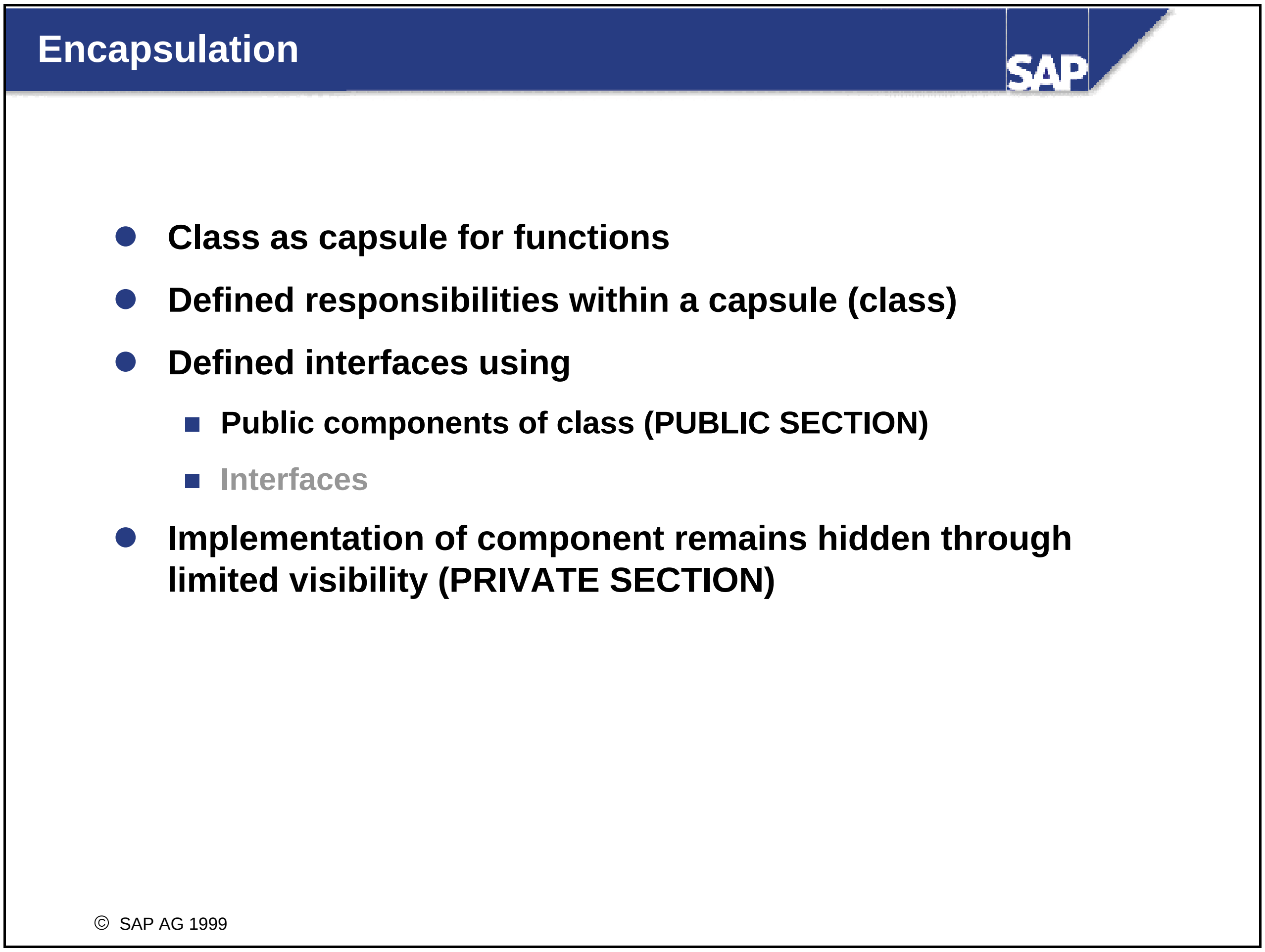
```
* Example 2:

DATA class_id TYPE string.

class_id = lcl_airplane=>count.
```

```
* Example 3:

DATA count_airplane TYPE I.

CALL METHOD lcl_airplane=>get_count
  RECEIVING re_count = count_airplane.
```



Principles: Overview (7)

- Objects
- Classes
- Attributes
- Methods
- Instantiation, Garbage Collector
- Working with Objects
- Further Principles



Encapsulation

- Class as capsule for functions
- Defined responsibilities within a capsule (class)
- Defined interfaces using
  - Public components of class (PUBLIC SECTION)
  - Interfaces
- Implementation of component remains hidden through limited visibility (PRIVATE SECTION)

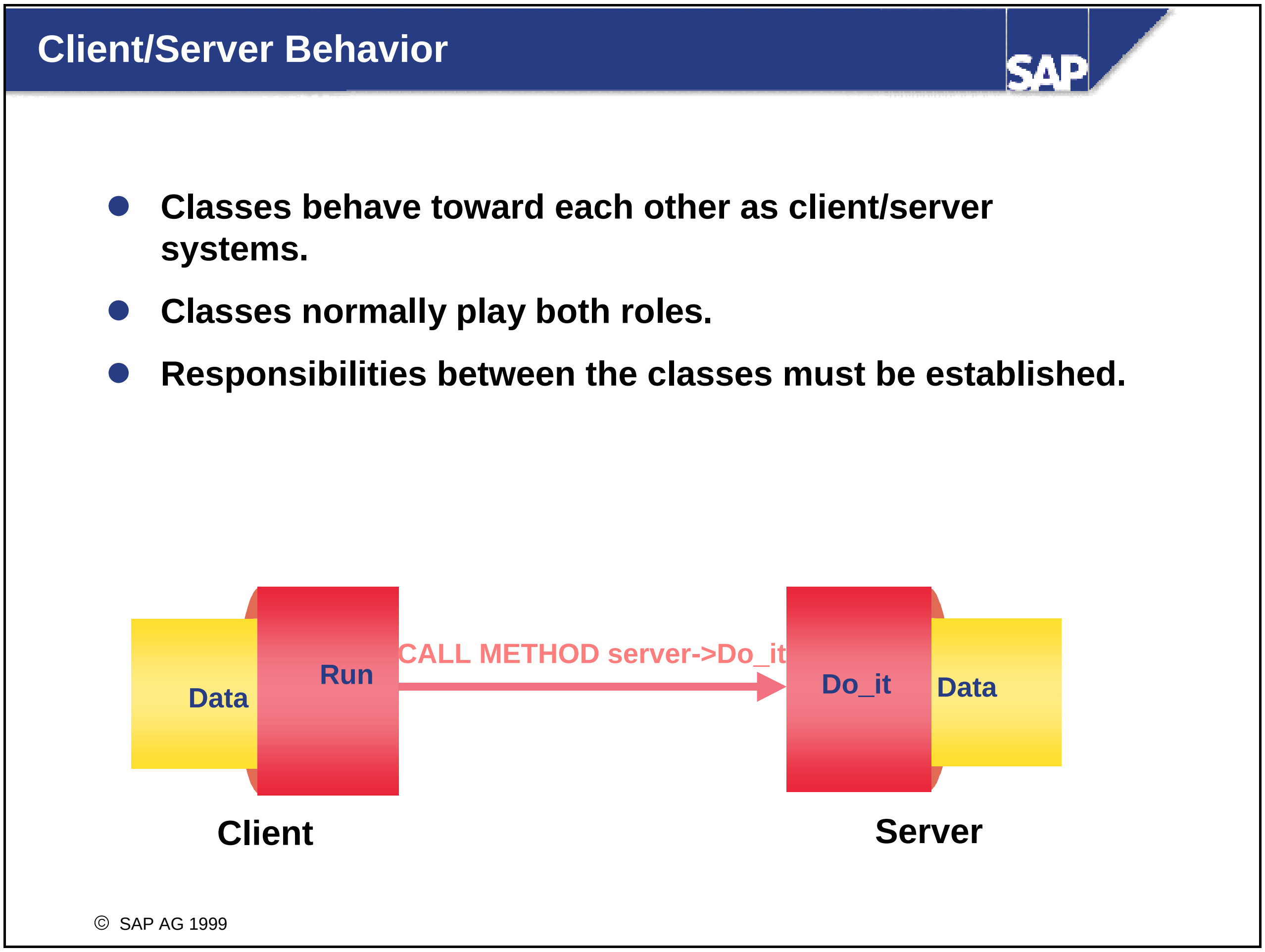
© SAP AG 1999

Encapsulation

The principle of encapsulation is to hide the implementation of a class from other components in the system, so that these components cannot make assumptions about the internal state of the objects in that class or of the class itself. This prevents dependencies on specific implementations from arising.

The class is the capsule surrounding related functions.

The principle of visibility ensures that the implementation of the functions and the information administered within a class is hidden.



Client/Server Behavior

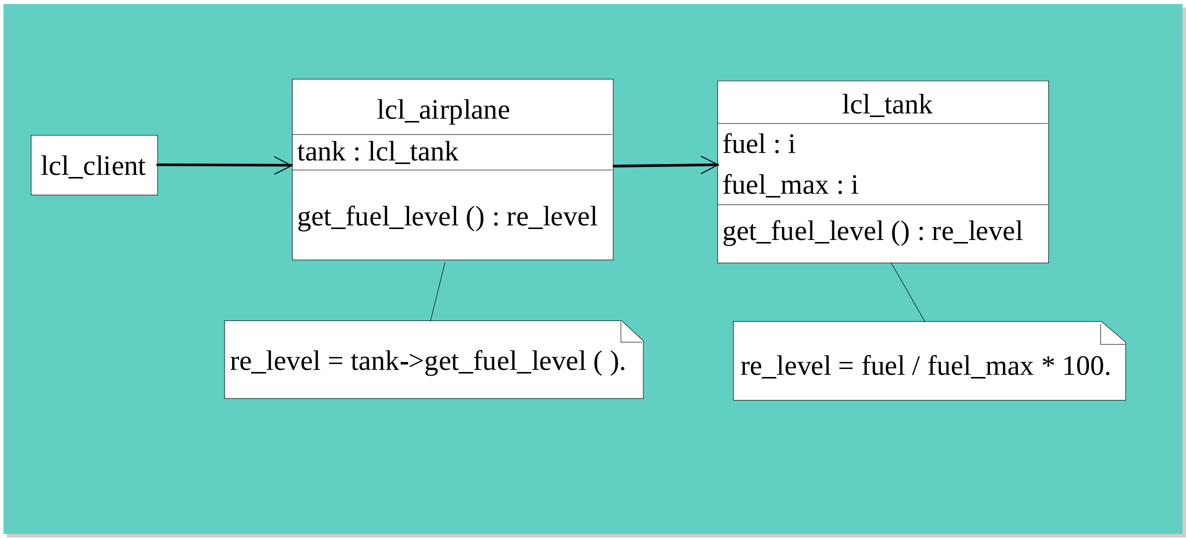
- Classes behave toward each other as client/server systems.
- Classes normally play both roles.
- Responsibilities between the classes must be established.

Classes behave like client/server systems: When a class is called by a method of another class, it automatically becomes the client of the other (server) class. This creates two requirements :
- The client class must observe the protocol of the server class.
- The server class protocol must be clear and detailed enough that a potential client has no trouble in orienting by it.

Classes normally play both roles. Every class is a potential server class, and when it is called by a method of another class it then becomes a client class too.

Establishing logical business and software responsibilities between classes results in a true client/server software system in which redundancy is avoided.

The Delegation Principle



lcl_client

lcl_airplane
tank : lcl_tank
get_fuel_level () : re_level

lcl_tank
fuel : i
fuel_max : i
get_fuel_level () : re_level

re_level = tank->get_fuel_level ( ).

re_level = fuel / fuel_max * 100.

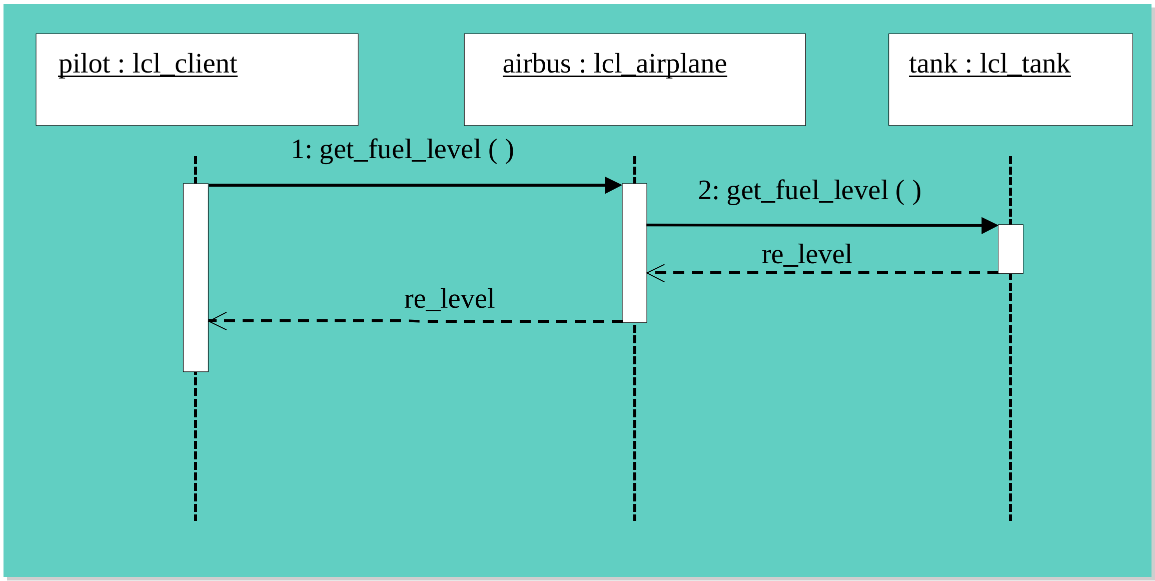
In delegation, two objects are involved in handling a request: the recipient of the request delegates the execution of the request to a delegate.

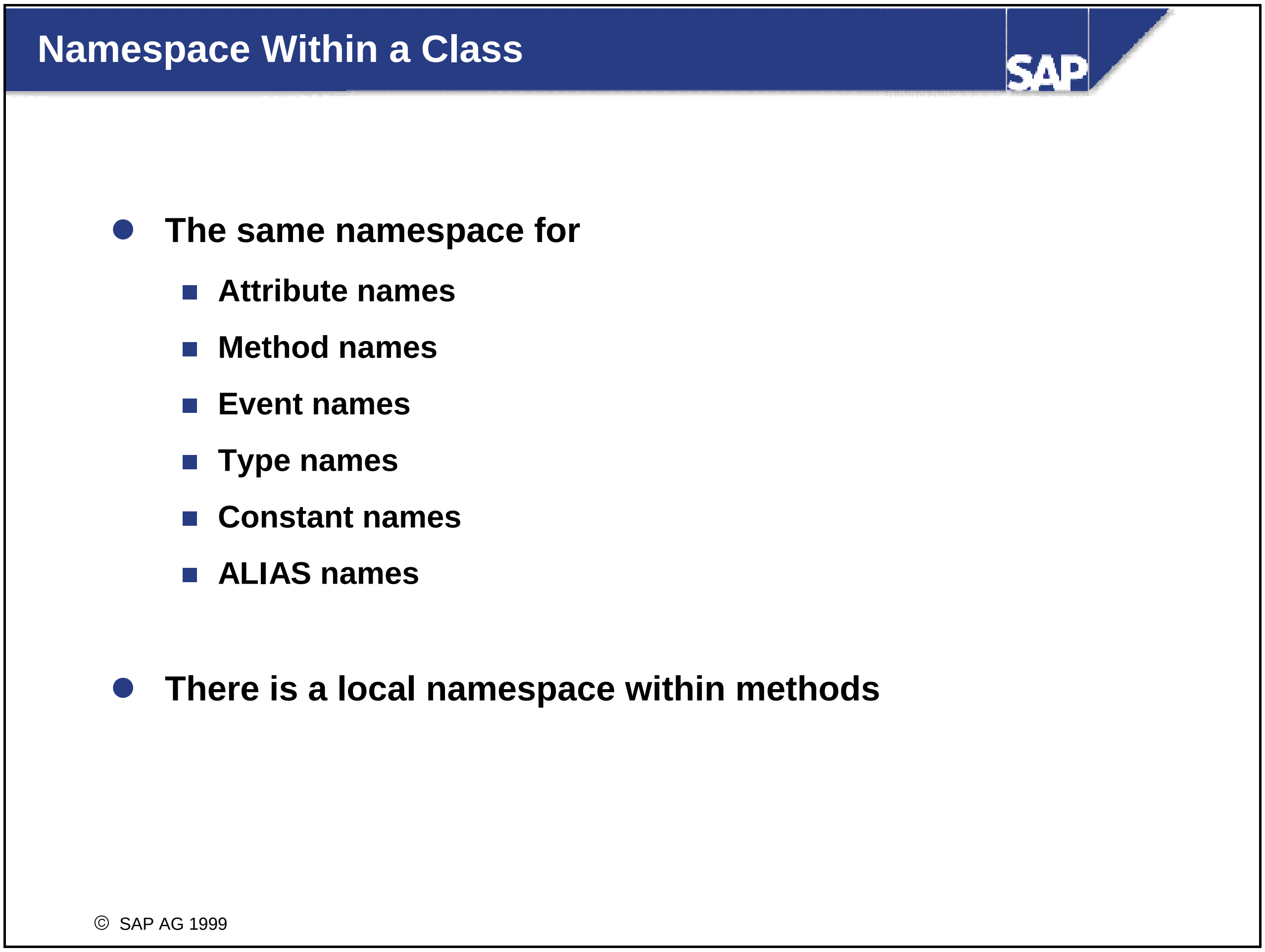
Example:
The pilot (lcl_client) calls the method get_fuel_level from the airplane class (lcl_airplane). The airplane cannot carry out this task itself. Therefore the airplane calls the get_fuel_level method from the tank class (lcl_tank), that is, the airplane delegates the execution of the method to the tank.

The main advantage of delegation (as a re-use mechanism) lies in the option of changing the behavior of the recipient by substituting the delegate (at runtime). For example, delegation enables the airplane to be equipped with a new tank, without the call changing for the client or for the airplane class.

Good capsulation often forces you to use delegation: if tank in the above example were a private attribute in class lcl_airplane, then the user cannot address the tank directly, but only through the airplane!

Sequence Diagram: Delegation



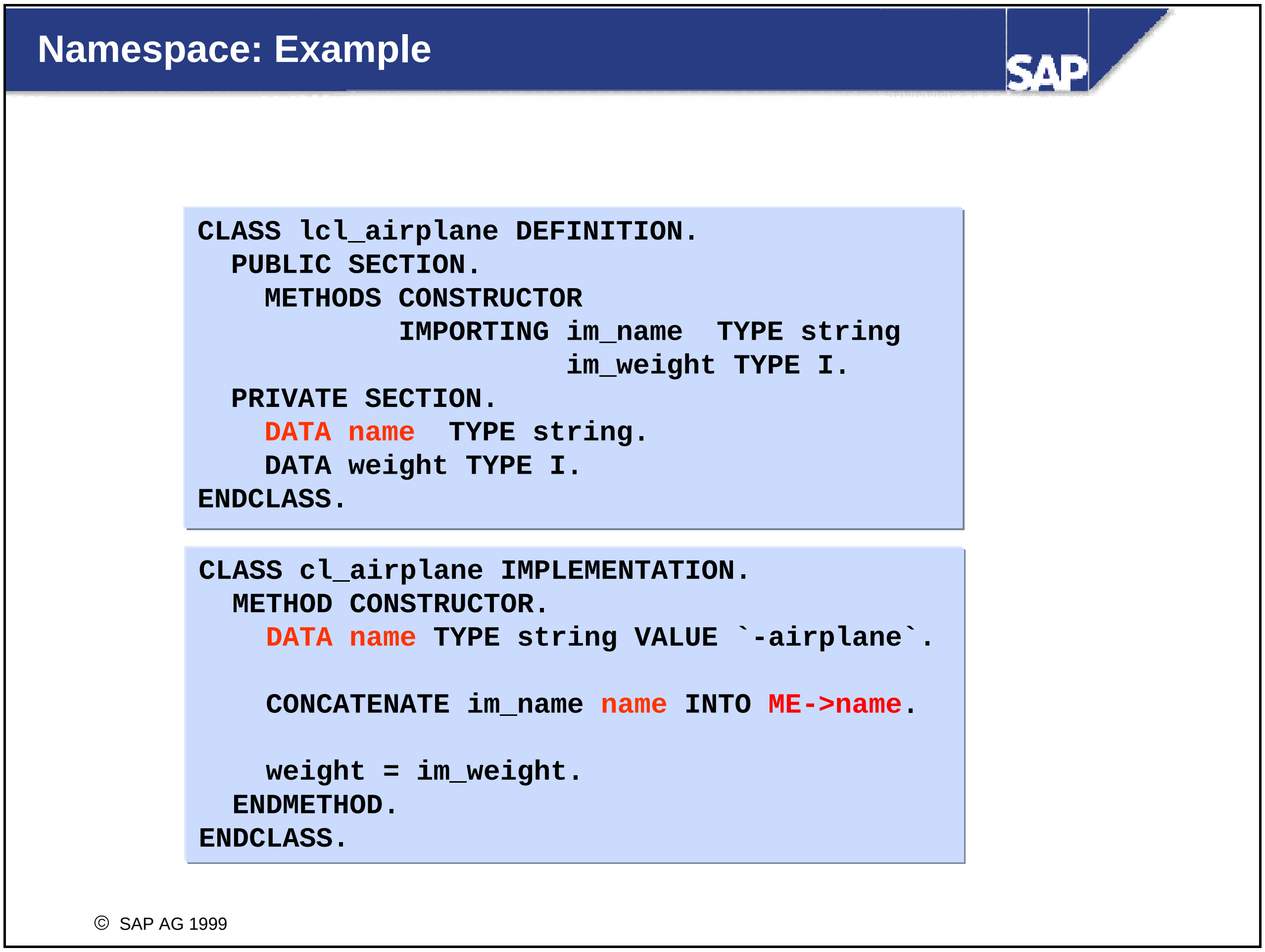


Namespace Within a Class

- The same namespace for
  - Attribute names
  - Method names
  - Event names
  - Type names
  - Constant names
  - ALIAS names
- There is a local namespace within methods

Within a class, attribute names, method names, event names, constant names, type names and alias names all share the same namespace.

There is a local namespace within methods. Definitions of variables can cover components in one class.



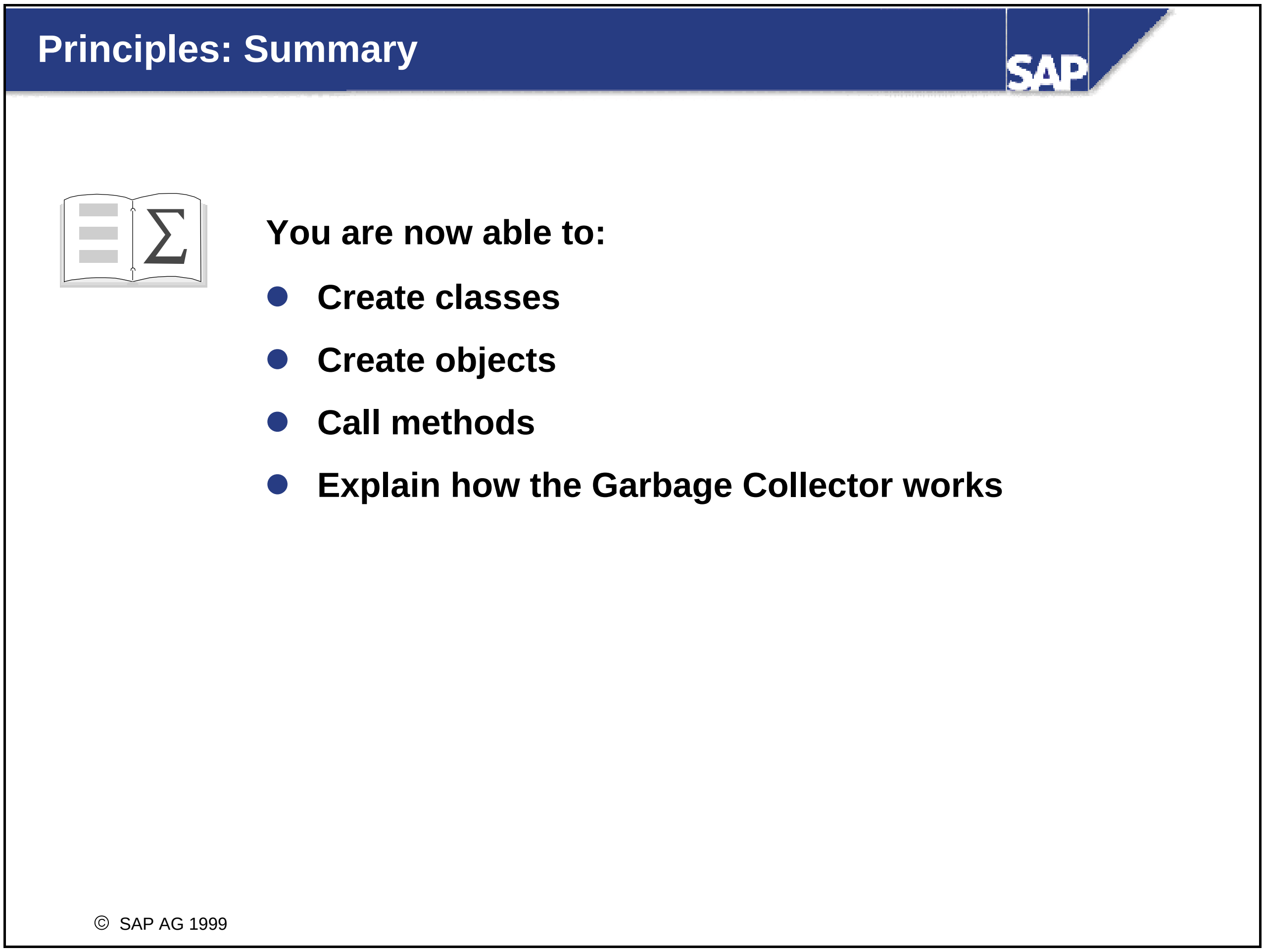
Namespace: Example

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS CONSTRUCTOR
            IMPORTING im_name  TYPE string
                      im_weight TYPE I.
  PRIVATE SECTION.
    DATA name  TYPE string.
    DATA weight TYPE I.
ENDCLASS.
```

```
CLASS cl_airplane IMPLEMENTATION.
  METHOD CONSTRUCTOR.
    DATA name TYPE string VALUE `-airplane`.

    CONCATENATE im_name name INTO ME->name.

    weight = im_weight.
  ENDMETHOD.
ENDCLASS.
```

You can address the object itself within object methods using the implicitly available reference variable ME.

Description of the example:
In the CONSTRUCTOR, the instance attribute name is covered by the locally defined variable name. In order to still be able to address the instance attribute, you need to use ME.



Principles: Summary

You are now able to:

- Create classes
- Create objects
- Call methods
- Explain how the Garbage Collector works

Exercises

Unit: Principles

Topic: Creating Classes

At the end of this exercise you will be able to:

- Create a class

An airline needs to manage its airplanes.

1-1 Create development class for your group **ZBC404_##** (##: group number) and save all the repository objects you have created during the course in this development class.

1-2 Create include program **ZBC404_##_LCL_AIRPLANE**
(##: group number) .

1-3 Create class *lcl_airplane* in the above include program.

1-3-1 This class has two **private instance attributes**:
- *name*

- *planetype*.

The attribute for the name of the airplane should be type C, length 25. Define the type in the PUBLIC SECTION. Define the attribute for the plane type using the table field *saplane-planetyp*.

1-3-2 The class has a **private static attribute**:
- *n_o_airplanes*.

This attribute should be type I.

1-3-3 The class has a **public instance method** *set_attributes* to set the private instance attributes name and plane type. Enter two corresponding importing parameters for the declaration of the method in the definition part. The definition of these parameters should be analogous to the two attributes.
Implement the method in the implementation part. Each time the method is called, the static attribute *n_o_airplanes* should increase by one.

1-3-4 The class has another **public instance method** *display_attributes* to display the instance attributes. Declare this method and output the attribute in the implementation part using the WRITE statement.

1-3-5 Declare and implement a **public static method** *display_n_o_airplanes* to display the static attribute *n_o_airplanes*.

Unit: Principles

Topic: Instantiating Objects

At the end of this exercise you will be able to:

- Instantiate objects
- Call methods

An airline needs to manage its airplanes.

2-1 Create program **ZBC404_##_MAINTAIN_AIRPLANES** (##: group number)**.**

2-2 Use the INCLUDE statement to include program **ZBC404_##_LCL_AIRPLANE** (##: group number) from the previous exercise .

2-3 Create a reference to class *lcl_airplane*.

2-4 Call the static method *display_n_o_airplanes* (before instantiating an object in class *lcl_airplane*).

2-5 Create an object in class *lcl_airplane*.

2-6 Set the object attributes using the *set_attributes* method.

2-6-1 Invent an airplane name and airplane type and pass them as text literal.

2-7 Display the object attributes using the *display_attributes* method.

2-8 Call the static method *display_n_o_airplanes* for a second time .

Unit: Principles

Topic: Constructor

At the end of this exercise you will be able to:

- Create a constructor for a class
- Create an object using the constructor
-

An airline needs to manage its airplanes.

3-1 Create a constructor for class *lcl_airplane* (in the include program **ZBC404_##_LCL_AIRPLANE**)

1-1-1 The constructor must have two importing parameters that fill the instance attributes *name* and *planetype*.

1-1-2 The static attribute *n_o_airplanes* should have an ascending sequence of one in the constructor.

3-2 In the method *set_attributes*, comment out the line in which the static attribute *n_o_airplanes* is increased by one.

3-3 In the main program **ZBC404_##_MAINTAIN_AIRPLANES**, extend the creation of the object with the constructor interface.

1-3-1 Fill the constructor interface parameters with the same values you used when calling the *set_attributes* method.

3-4 Comment out the *set_attributes* method call.

Principles Solutions

Unit: Principles

Topic: Creating Classes

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

* Public section
 PUBLIC SECTION.

  TYPES: name_type(25) TYPE c.
  CONSTANTS: pos_1 TYPE i VALUE 30.

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

  CLASS-METHODS: display_n_o_airplanes.

* Private section
 PRIVATE SECTION.

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

  CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.
```

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

* Method set_attributes
  METHOD set_attributes.
    name          = im_name.
    planetype     = im_planetype.
    n_o_airplanes = n_o_airplanes + 1.
  ENDMETHOD.

* Method display_attributes
  METHOD display_attributes.
    WRITE: / 'Name of the airplane: '(001), AT pos_1 name,
           / 'Plane type:           '(002), AT pos_1 planetype.
  ENDMETHOD.

* Method display_n_o_airplanes
  METHOD display_n_o_airplanes.
    WRITE: /, / 'Total number of airplanes: '(ca1),
           AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
  ENDMETHOD.

ENDCLASS.
```

Unit: Principles

Topic: Instantiating Objects

```
REPORT  sapbc404bass_create_object    .

* Including class lcl_airplane
INCLUDE sapbc404bass_lcl_airplane_1.

DATA: airplane TYPE REF TO lcl_airplane.


START-OF-SELECTION.

  CALL METHOD lcl_airplane=>display_n_o_airplanes.

  CREATE OBJECT airplane.

  CALL METHOD airplane->set_attributes EXPORTING
                                 im_name      = 'LH Berlin'
                                 im_planetype = '747-400'.

  CALL METHOD airplane->display_attributes.

  CALL METHOD lcl_airplane=>display_n_o_airplanes.
```

Include program SAPBC404BASS_LCL_AIRPLANE_1

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

* Public section
  PUBLIC SECTION.

    TYPES: name_type(25) TYPE c.
    CONSTANTS: pos_1 TYPE i VALUE 30.

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

    CLASS-METHODS: display_n_o_airplanes.

* Private section
  PRIVATE SECTION.

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

    CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.


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

* Method set_attributes
  METHOD set_attributes.
    name          = im_name.
    planetype     = im_planetype.
    n_o_airplanes = n_o_airplanes + 1.
  ENDMETHOD.

* Method display_attributes
  METHOD display_attributes.
    WRITE: / 'Name of the airplane: '(001), AT pos_1 name,
           / 'Plane type:           '(002), AT pos_1 planetype.
  ENDMETHOD.

* Method display_n_o_airplanes
  METHOD display_n_o_airplanes.
    WRITE: /, / 'Total number of airplanes: '(ca1),
           AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
  ENDMETHOD.

ENDCLASS.
```

Unit: Principles

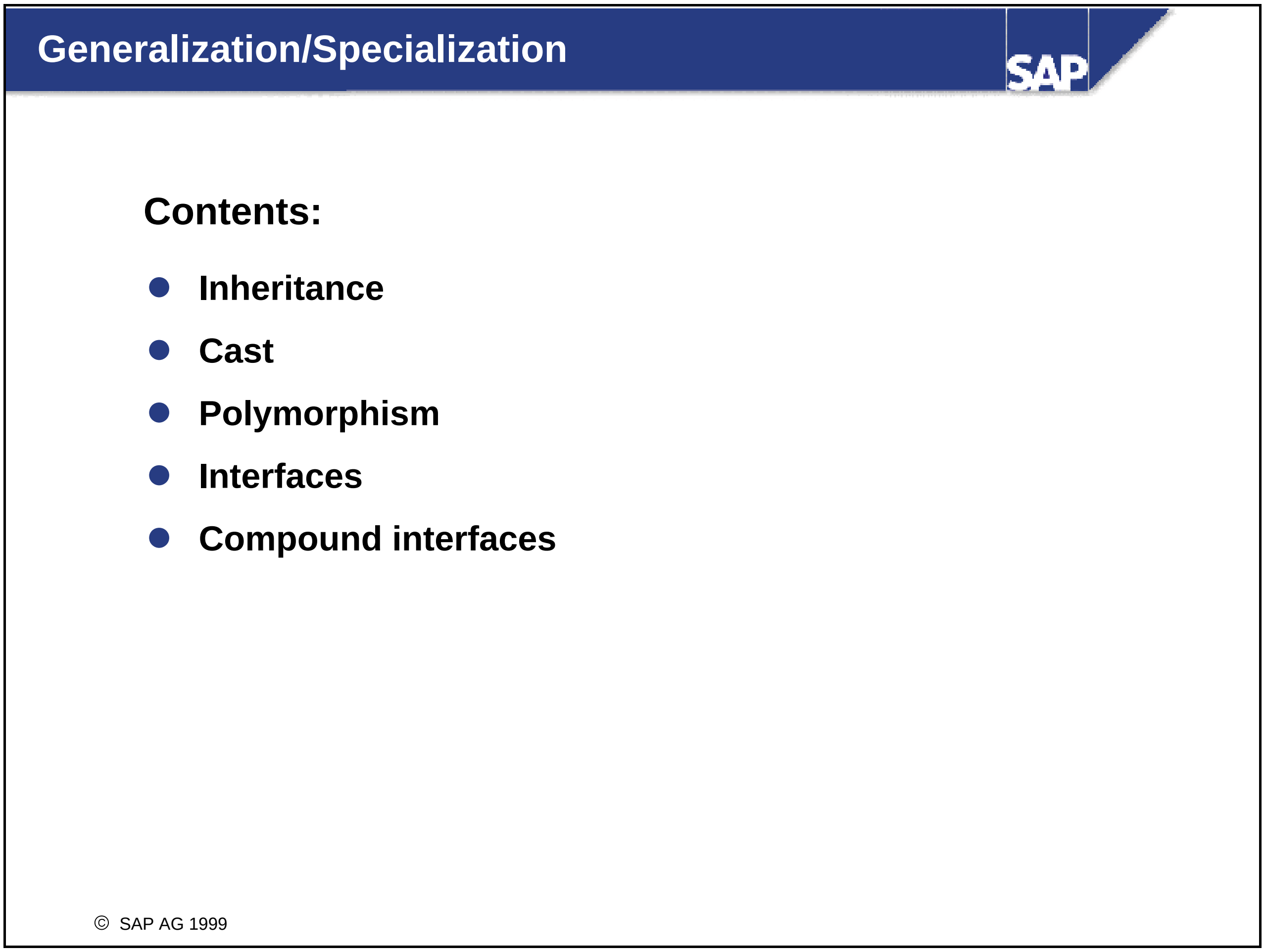
Topic: Constructor

```
REPORT  sapbc404bass_constructor     .

include sapbc404bass_lcl_airplane_2.

DATA: airplane TYPE REF TO lcl_airplane.


START-OF-SELECTION.

  CALL METHOD lcl_airplane=>display_n_o_airplanes.

* Create object with constructor
  CREATE OBJECT airplane EXPORTING im_name      = 'LH Berlin'
                                   im_planetype = '747-400'.

*  CALL METHOD airplane->set_attributes EXPORTING
*                                     im_name      = 'LH Berlin'
*                                     im_planetype = '747-400'.

  CALL METHOD airplane->display_attributes.

  CALL METHOD lcl_airplane=>display_n_o_airplanes.
```

Include program SAPBC404BASS_LCL_AIRPLANE_2

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

  PUBLIC SECTION.

    TYPES: name_type(25) TYPE c.
    CONSTANTS: pos_1 TYPE i VALUE 30.

*   NEW: constructor
    METHODS: constructor importing
                    im_name      TYPE name_type
```

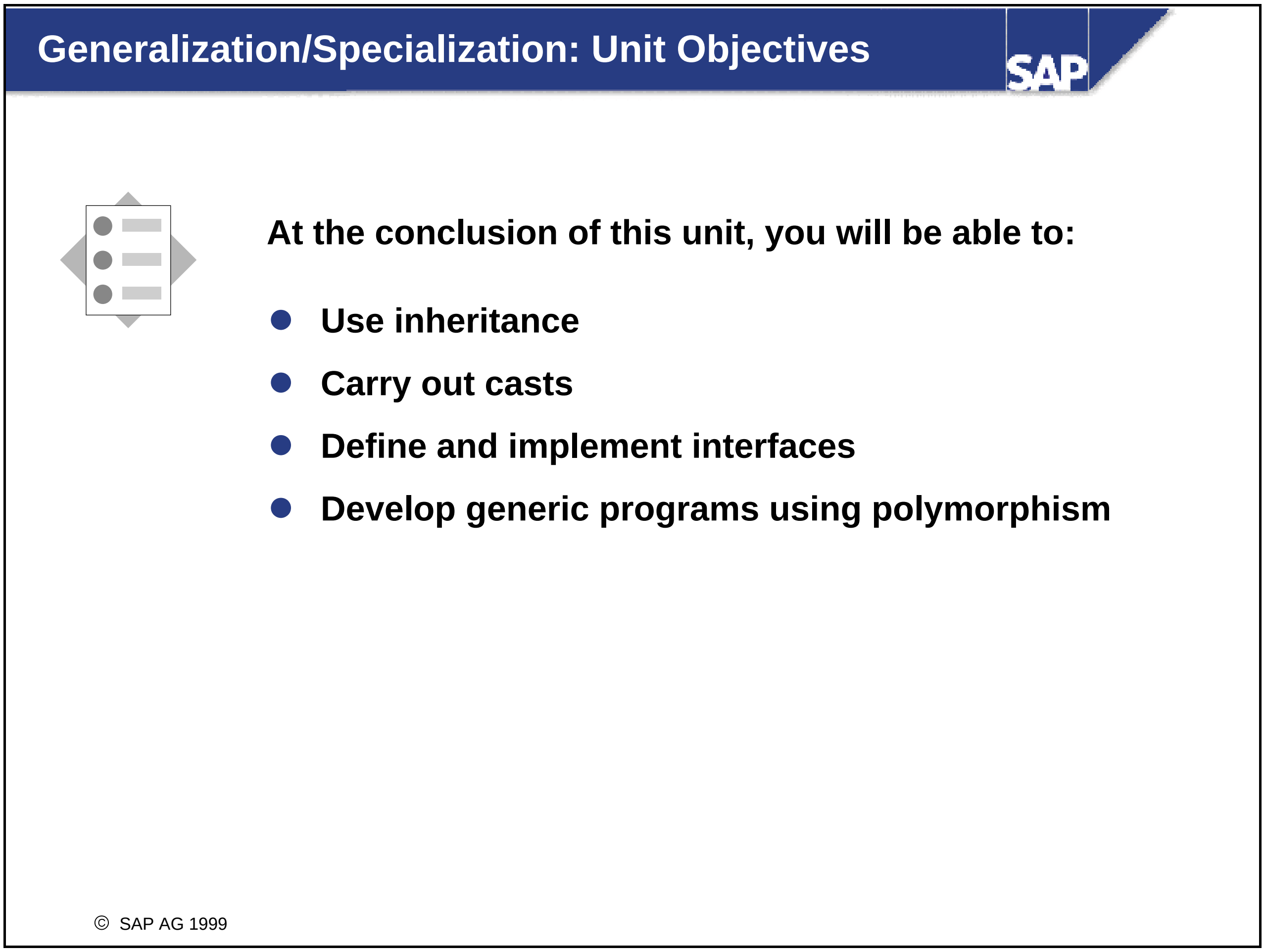
```
            im_planetype TYPE saplane-planetype,
      set_attributes IMPORTING
                                 im_name      TYPE name_type
              im_planetype TYPE saplane-planetype,
      display_attributes.

  CLASS-METHODS: display_n_o_airplanes.

 PRIVATE SECTION.

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

  CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.


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

* NEW: constructor
 METHOD constructor.
  name          = im_name.
  planetype     = im_planetype.
  n_o_airplanes = n_o_airplanes + 1.
 ENDMETHOD.

 METHOD set_attributes.
  name      = im_name.
  planetype = im_planetype.
*   n_o_airplanes = n_o_airplanes + 1.
 ENDMETHOD.

 METHOD display_attributes.
  WRITE: / 'Name of the airplane: '(001), at pos_1 name,
      / 'Plane type:           '(002), at pos_1 planetype.
 ENDMETHOD.

 METHOD display_n_o_airplanes.
  WRITE: /, / 'Total number of airplanes: '(ca1),
      AT pos_1 n_o_airplanes left-justified, /.
 ENDMETHOD.

ENDCLASS.
```



Generalization/Specialization

Contents:

- Inheritance
- Cast
- Polymorphism
- Interfaces
- Compound interfaces

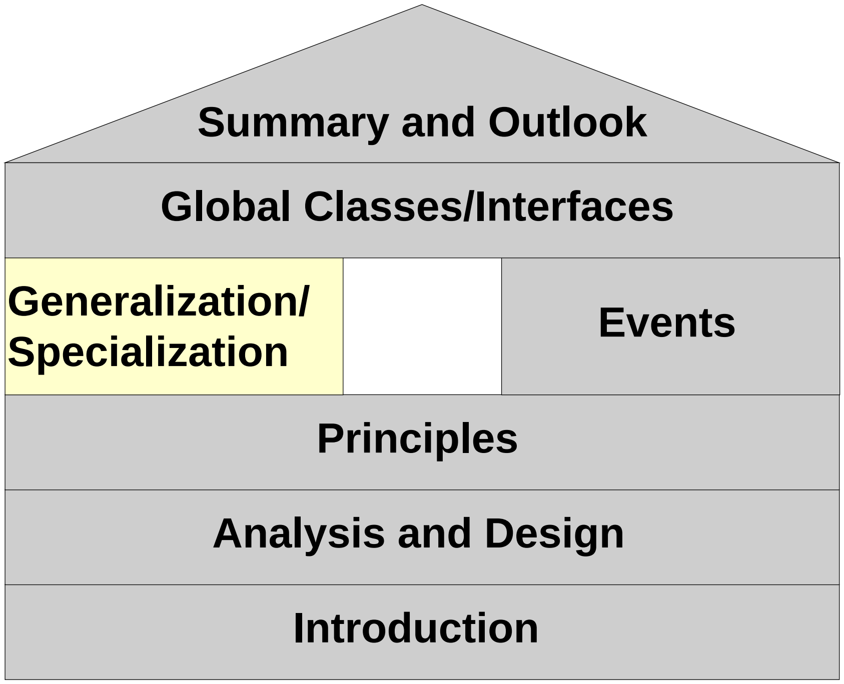


Generalization/Specialization: Unit Objectives

At the conclusion of this unit, you will be able to:

- Use inheritance
- Carry out casts
- Define and implement interfaces
- Develop generic programs using polymorphism

Generalization/Specialization: Course Overview Diagram



Summary and Outlook

Global Classes/Interfaces

Generalization/ Specialization

Events

Principles

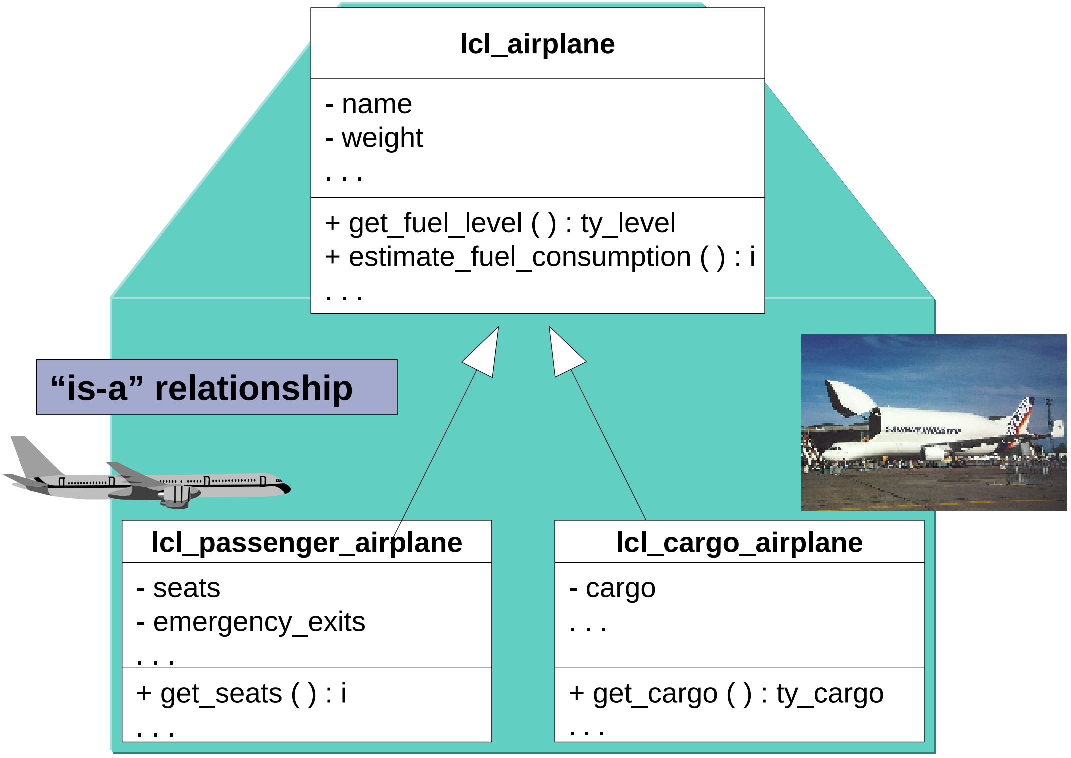
Analysis and Design

Introduction

Generalization/Specialization: Overview (1)

- ▶ Inheritance
- Cast
- Polymorphism
- Further Characteristics of Inheritance
- Interfaces
- Compound Interfaces

Inheritance (1)



lcl_airplane

- name
- weight
. . .

+ get_fuel_level ( ) : ty_level
+ estimate_fuel_consumption ( ) : i
. . .

"is-a" relationship

lcl_passenger_airplane

- seats
- emergency_exits
. . .

+ get_seats ( ) : i
. . .

lcl_cargo_airplane

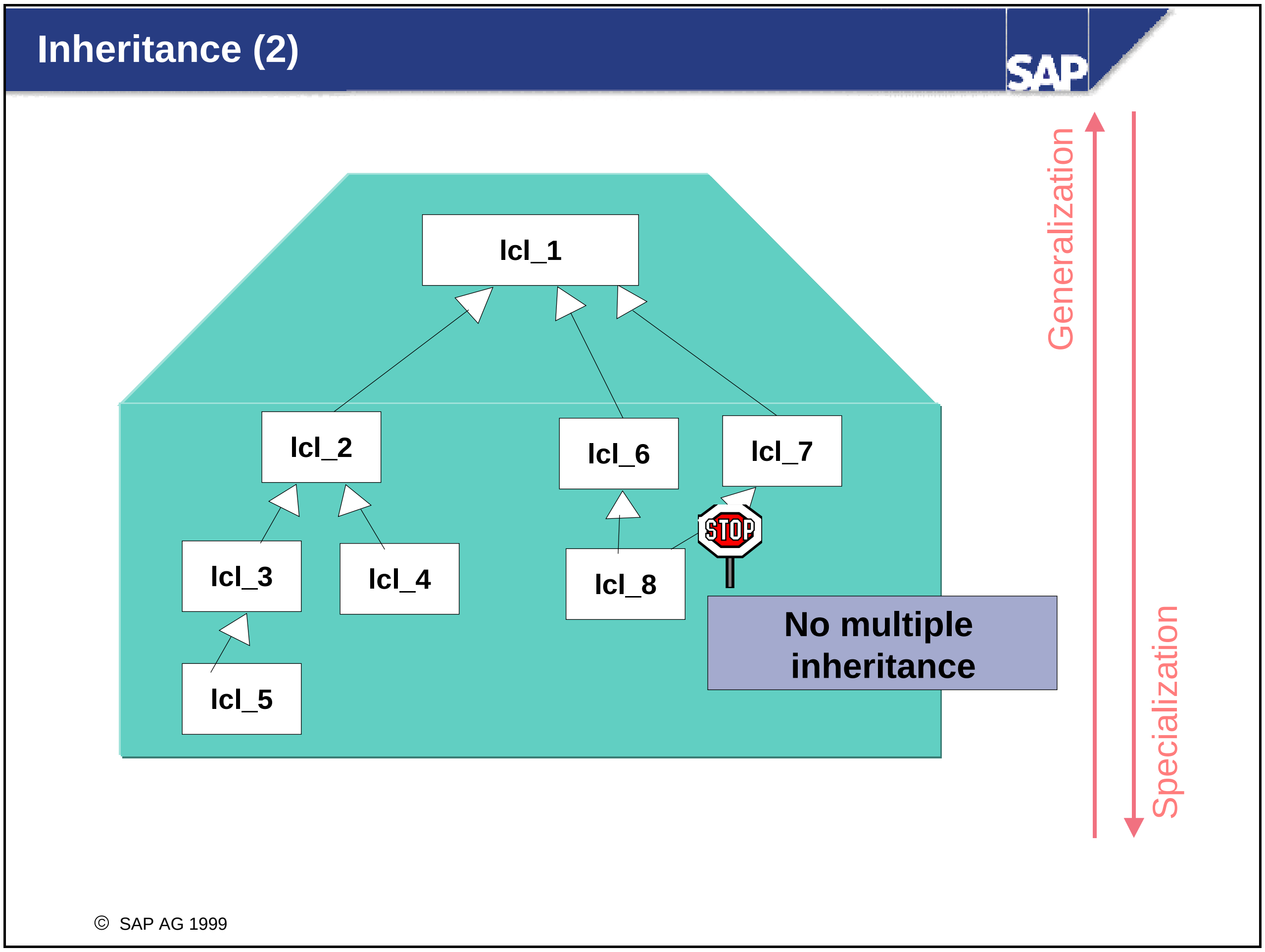
- cargo
. . .

+ get_cargo ( ) : ty_cargo
. . .

Inheritance 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 inherited methods with its own implementations.

Inheritance is an implementation relationship that emphasizes similarities between classes. In the example above, the similarities between the passenger plane and cargo plane classes are extracted to the airplane superclass. This means that common components are only defined/implemented in the superclass and are automatically present in the subclasses.

The inheritance relationship is often described as an "is-a" relationship: a passenger plan **is an** airplane.



Inheritance (2)

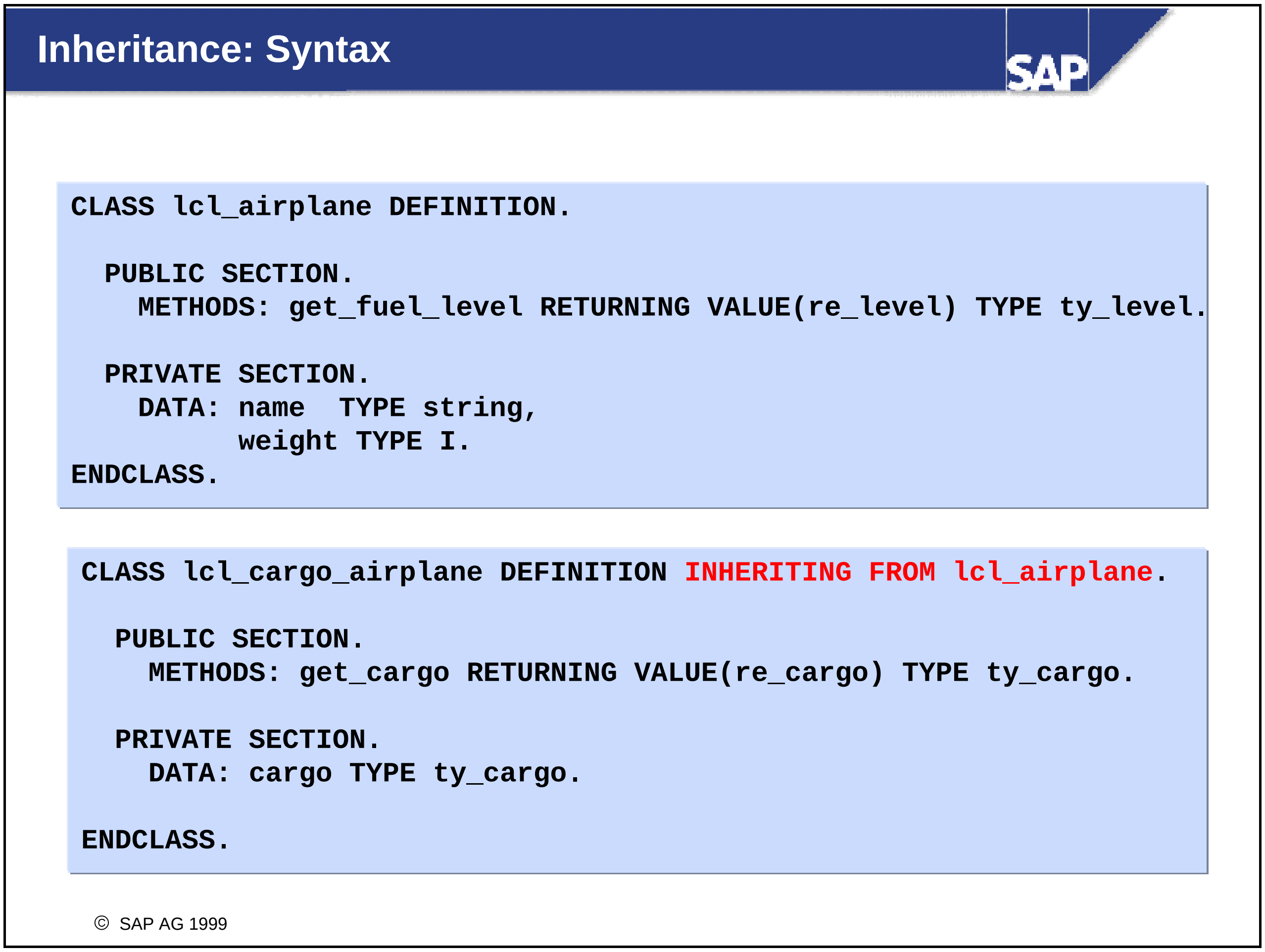
No multiple inheritance

Inheritance should be used to implement generalization and specialization relationships. A superclass is a **generalization** of its subclasses. The subclass in turn is a **specialization** of its superclasses.

The situation in which a class, for example lcl_8, inherits from two classes (lcl_6 and lcl_7) simultaneously, is known as multiple inheritance. This is **not** realized in ABAP Objects. ABAP Objects only has single inheritance.
However, you can simulate multiple inheritance in ABAP Objects using interfaces (see the section on interfaces).

Single inheritance does not mean that the inheritance tree only has one level. On the contrary, the direct superclass of one class can in turn be the subclass of a further superclass. In other words: the inheritance tree can have any number of levels, so that a class can inherit from several superclasses, as long as it only has one direct superclass.

Inheritance is a "one-sided relationship": subclasses know their direct superclasses, but (super)classes do not know their subclasses.

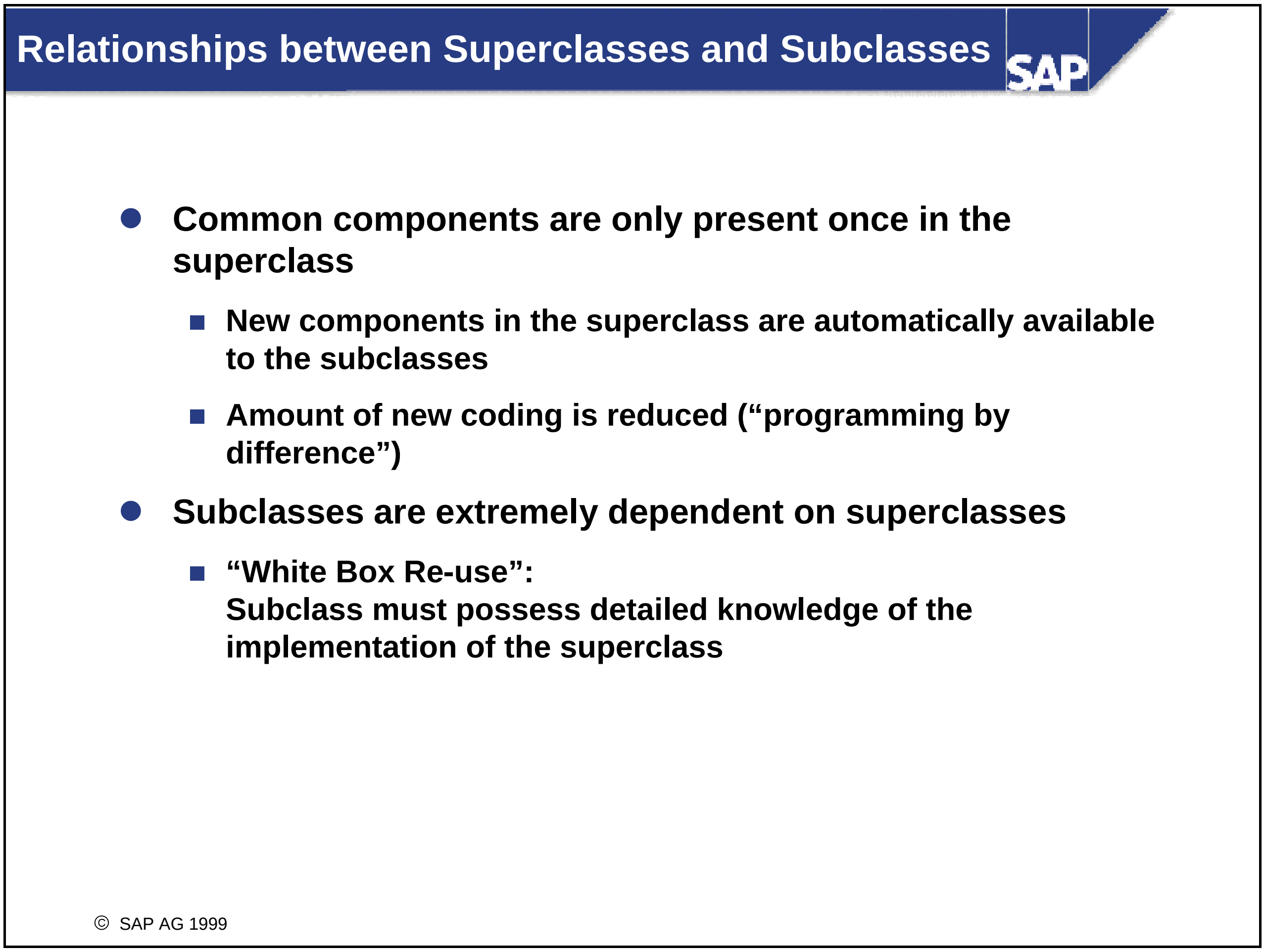


Inheritance: Syntax

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    METHODS: get_fuel_level RETURNING VALUE(re_level) TYPE ty_level.

  PRIVATE SECTION.
    DATA: name   TYPE string,
          weight TYPE I.
ENDCLASS.
```

```
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.

  PUBLIC SECTION.
    METHODS: get_cargo RETURNING VALUE(re_cargo) TYPE ty_cargo.

  PRIVATE SECTION.
    DATA: cargo TYPE ty_cargo.

ENDCLASS.
```

Normally the only other entry required for subclasses is what has changed in relation to the direct superclass. Only additions are permitted in ABAP Objects, that is, in a subclass you can “never take something away from a superclass”. All components from the superclass are automatically present in the subclass.

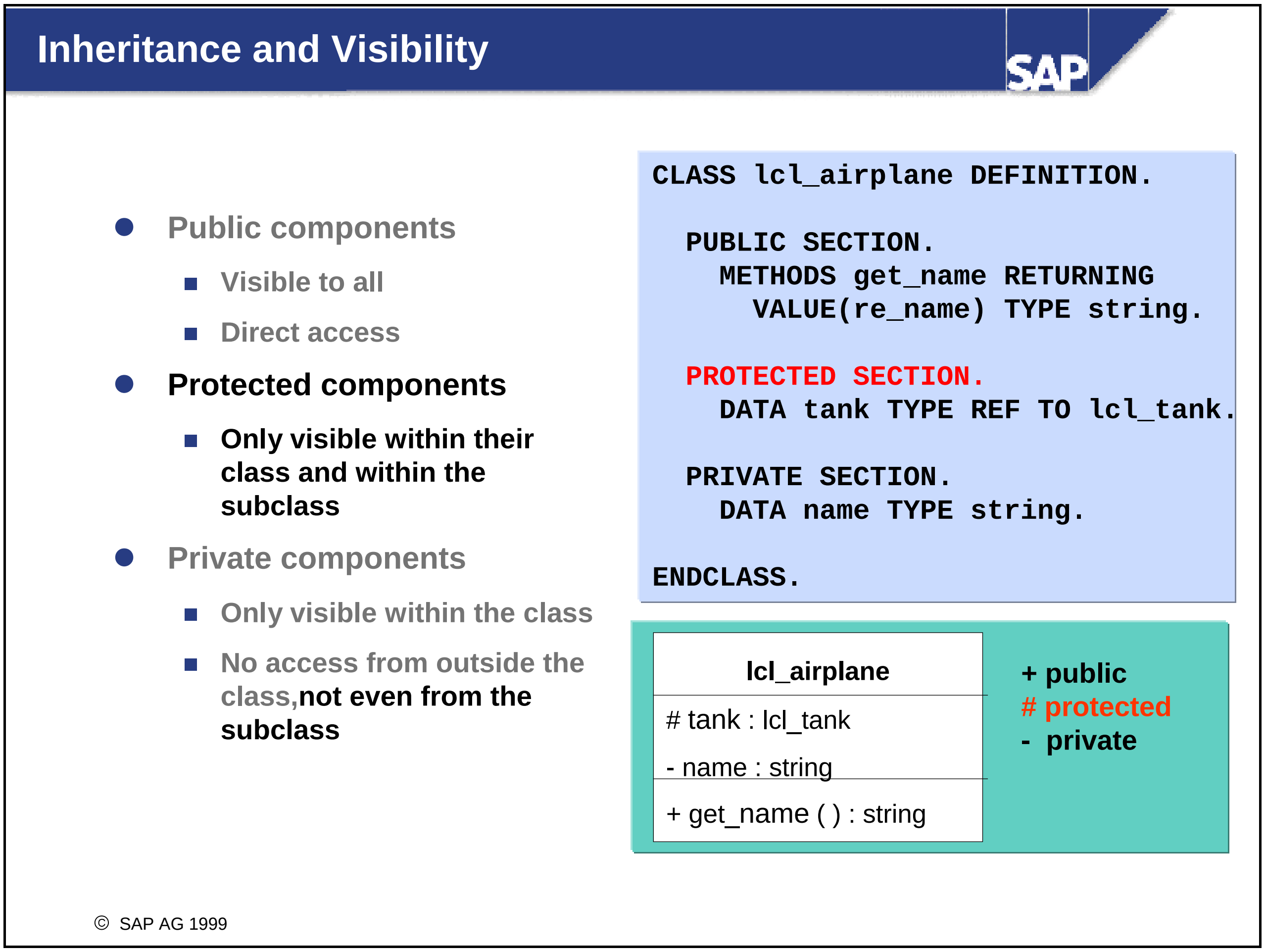


Relationships between Superclasses and Subclasses

- **Common components are only present once in the superclass**
  - **New components in the superclass are automatically available to the subclasses**
  - **Amount of new coding is reduced (“programming by difference”)**
- **Subclasses are extremely dependent on superclasses**
  - **“White Box Re-use”:**
Subclass must possess detailed knowledge of the implementation of the superclass

If inheritance is used properly, it provides a significantly better structure, as common components only need to be stored once centrally (in the superclass) and are then automatically available to subclasses. Subclasses also profit immediately from changes (although the changes can also render them invalid!).

Inheritance provides very strong links between the superclass and the subclass. The subclass must possess detailed knowledge of the implementation of the superclass, particularly for redefinition, but also in order to use inherited components. Even if, technically, the superclass does not know its subclasses, the subclass often makes additional requirements of the superclass, for example, because a subclass needs certain protected components or because implementation details in the superclass need to be changed in the subclass in order to redefine methods. The basic reason is that the developer of a (super)class cannot normally predict all the requirements that subclasses will later need to make of the superclass.



Inheritance and Visibility

- Public components
  - Visible to all
  - Direct access
- **Protected components**
  - **Only visible within their class and within the subclass**
- Private components
  - Only visible within the class
  - No access from outside the class,**not even from the subclass**

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    METHODS get_name RETURNING
      VALUE(re_name) TYPE string.

  PROTECTED SECTION.
    DATA tank TYPE REF TO lcl_tank.

  PRIVATE SECTION.
    DATA name TYPE string.

ENDCLASS.
```

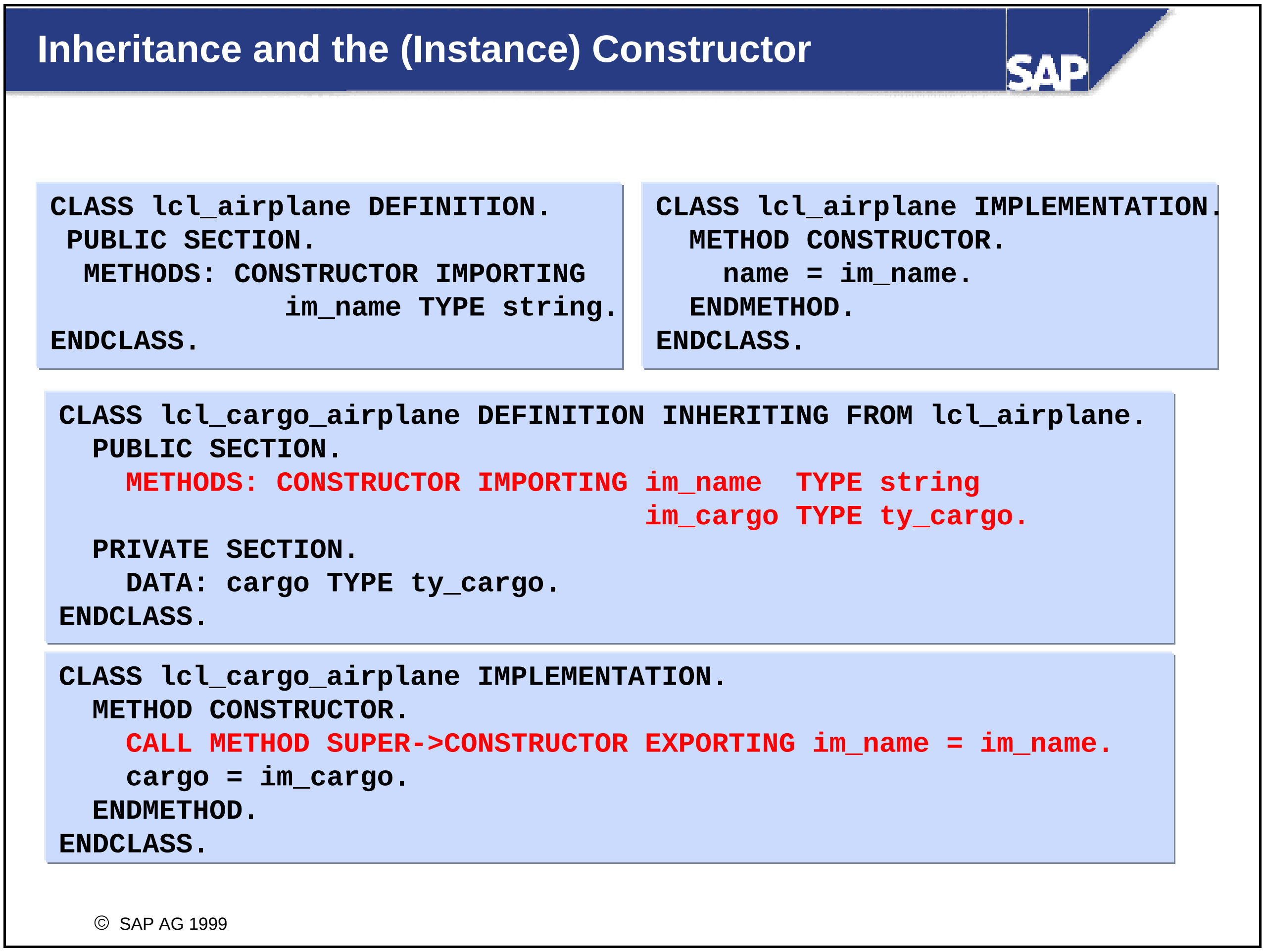
lcl_airplane
tank : lcl_tank - name : string
+ get_name ( ) : string

+ public
# protected
- private

Inheritance provides an extension of the visibility concept: there are protected components. The visibility of these components lies between that of the public components (visible to all users, all subclasses, and the class itself), and private (visible only to the class itself). Protected components are visible to and can be used by all subclasses and the class itself.

Subclasses cannot access the private components (particularly attributes) of the superclass. Private components are genuinely private. This is particularly important if a (super)class needs to make local enhancements to handle errors: it can use private components to do this without knowing or invalidating subclasses.

In ABAP Objects, you must keep to the section sequence PUBLIC, PROTECTED, PRIVATE.



Inheritance and the (Instance) Constructor

```
CLASS lcl_airplane DEFINITION.
 PUBLIC SECTION.
  METHODS: CONSTRUCTOR IMPORTING
                 im_name TYPE string.
ENDCLASS.
```

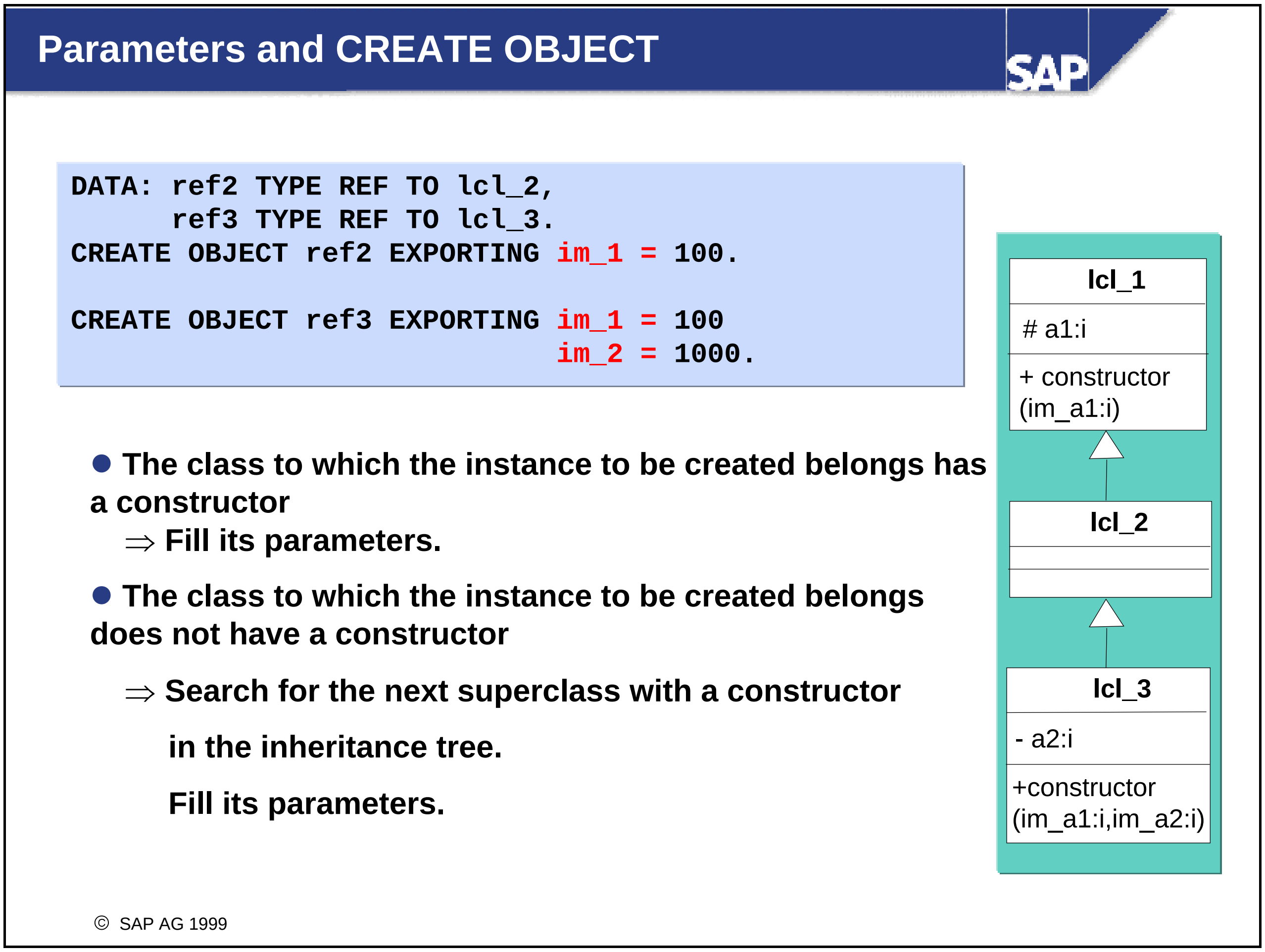
```
CLASS lcl_airplane IMPLEMENTATION.
  METHOD CONSTRUCTOR.
    name = im_name.
  ENDMETHOD.
ENDCLASS.
```

```
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.
  PUBLIC SECTION.
    METHODS: CONSTRUCTOR IMPORTING im_name  TYPE string
                                   im_cargo TYPE ty_cargo.
  PRIVATE SECTION.
    DATA: cargo TYPE ty_cargo.
ENDCLASS.
```

```
CLASS lcl_cargo_airplane IMPLEMENTATION.
  METHOD CONSTRUCTOR.
    CALL METHOD SUPER->CONSTRUCTOR EXPORTING im_name = im_name.
    cargo = im_cargo.
  ENDMETHOD.
ENDCLASS.
```

The constructor of the superclass **must** be called within the constructor of the subclass. The reason for this is the special task of the constructor: to ensure that objects are initialized correctly. Only the class itself, however, can initialize its own (private) components correctly; this task cannot be carried out by the subclass. Therefore it is essential that all (instance) constructors are called in an inheritance hierarchy (in the correct sequence).

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



Parameters and CREATE OBJECT

```
DATA: ref2 TYPE REF TO lcl_2,
      ref3 TYPE REF TO lcl_3.
CREATE OBJECT ref2 EXPORTING im_1 = 100.

CREATE OBJECT ref3 EXPORTING im_1 = 100
                             im_2 = 1000.
```

- **The class to which the instance to be created belongs has a constructor**
  ⇒ Fill its parameters.
- **The class to which the instance to be created belongs does not have a constructor**
  ⇒ Search for the next superclass with a constructor
  in the inheritance tree.
  Fill its parameters.

You must also consider the model described for instance constructors when using CREATE OBJECT. In this model, the constructor of the immediate superclass must be called before the non-inherited instance attributes can be initialized.

There are basically two methods of creating an instance in a class using CREATE OBJECT:

1. The instance constructor for that class has been defined (and implemented).

In this case, when you are using CREATE OBJECT, the parameters have to be filled according to the constructor interface, that is, optional parameters may, and non-optional parameters must be filled with actual parameters. If the constructor does not have any (formal) parameters, no parameters may or can be filled.

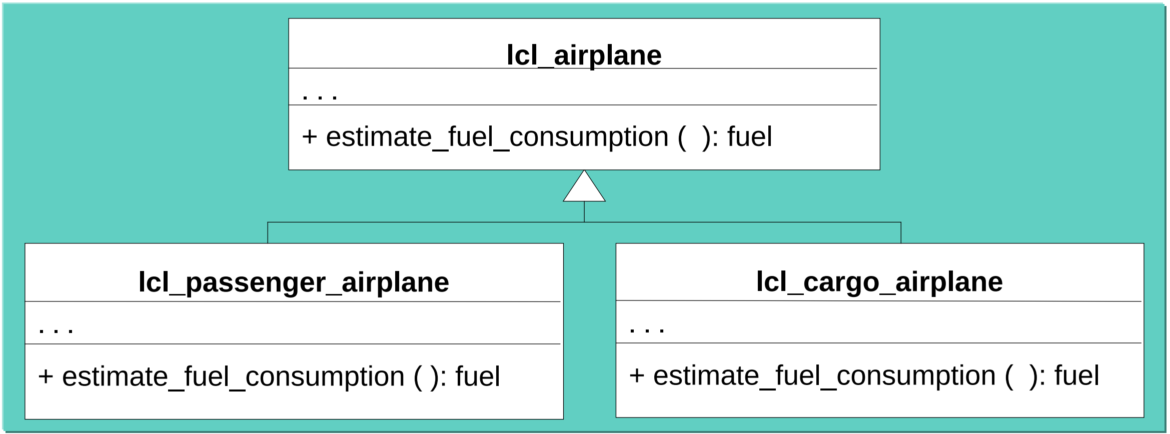
2. The instance constructor for that class has not been defined.

In this case, you must search the inheritance hierarchy for the next highest superclass in which the instance constructor has been defined and implemented. Then, when you are using CREATE OBJECT, the parameters of that class must be filled (similarly to the first method above).
If there is no superclass with a defined instance constructor, then no parameters may or can be filled.

If no instance constructor has been defined for a class, then a *default constructor, which is implicitly always present* is used. This default constructor calls the constructor from the immediate superclass.

Redefining Methods in ABAP Objects

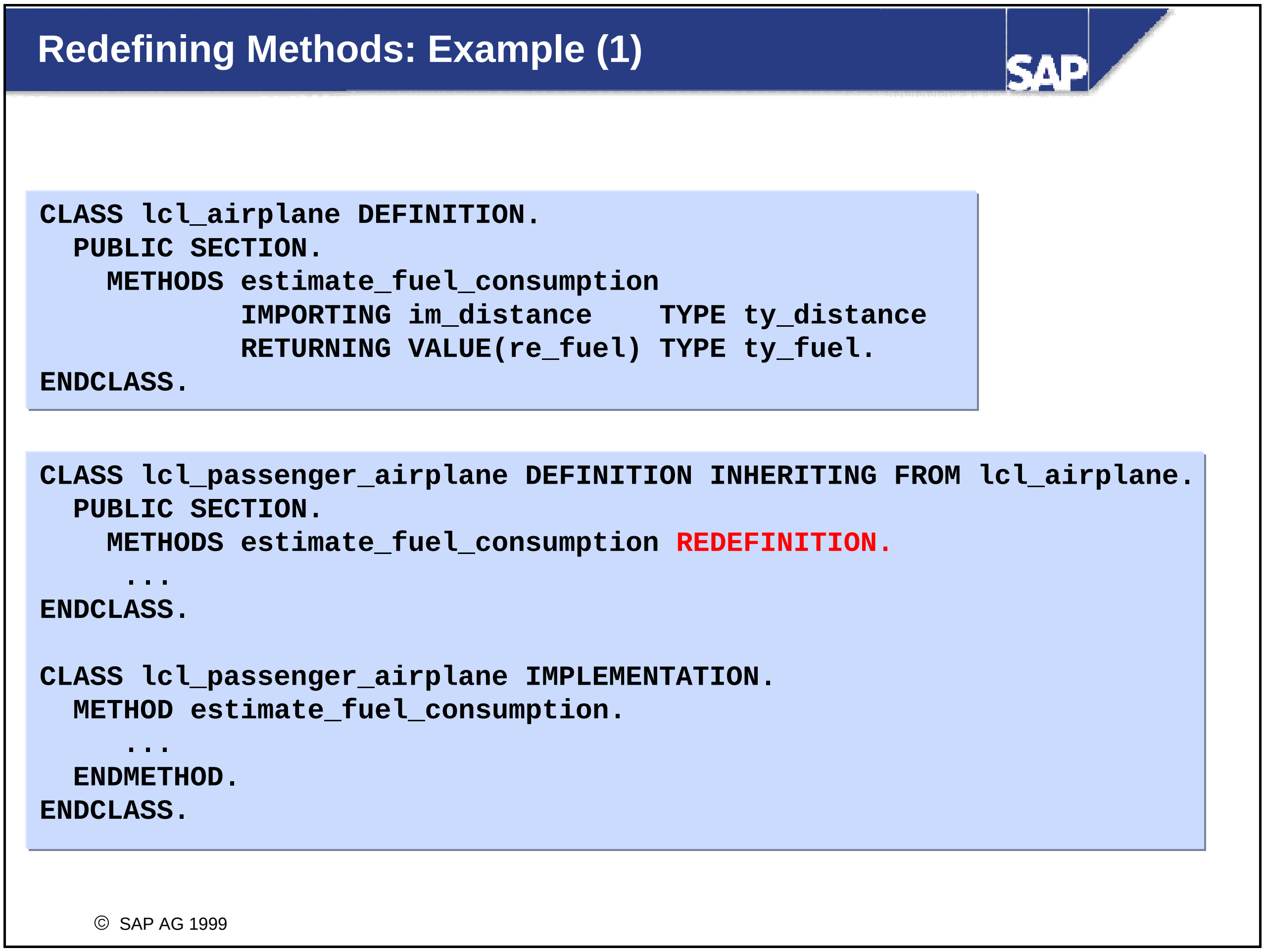
- Inherited methods can be redefined in subclasses
  - Redefined methods must be re-implemented in subclasses
  - The signature of redefined methods cannot be changed
  - You can only redefine instance methods, not static methods


lcl_airplane
. . .
+ estimate_fuel_consumption ( ): fuel
lcl_passenger_airplane
. . .
+ estimate_fuel_consumption ( ): fuel
lcl_cargo_airplane
. . .
+ estimate_fuel_consumption ( ): fuel

In ABAP Objects, you can not only add new components, but also provide inherited methods with new implementations. This is known as **redefinition**. You can only redefine (public and protected) instance methods, other components (static methods, attributes and so on) cannot be redefined. Furthermore, implementation is restricted to (re-)implementation of an inherited method; you cannot change method parameters (signature change).
You also cannot redefine a class’s (instance) constructor.

In UML, the redefinition of a method is represented by listing the method again in the subclass. Methods (and all other components) that are inherited but not redefined are not listed in the subclass, as their presence there is clear from the specialization relationship.

You should not confuse redefinition with “overlaying”. This describes the ability of a class to have methods with the same name but a different signature (number and type of parameters). This option is not available in ABAP Objects.



Redefining Methods: Example (1)

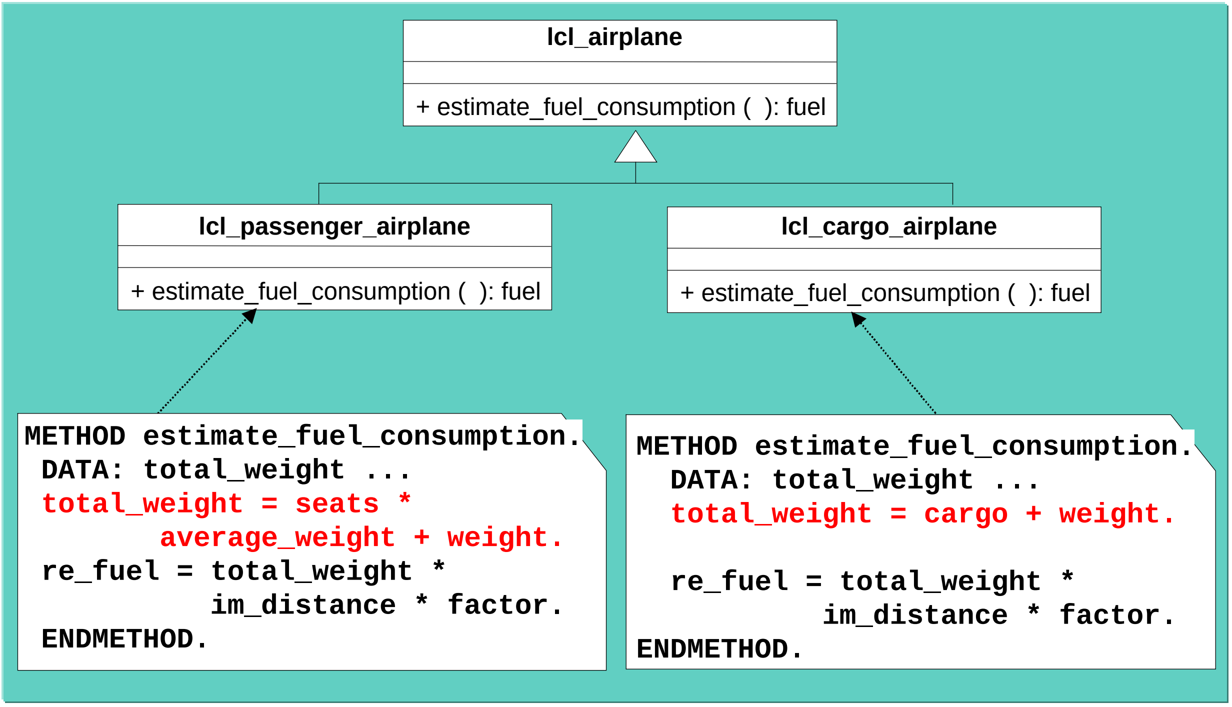
```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS estimate_fuel_consumption
            IMPORTING im_distance    TYPE ty_distance
            RETURNING VALUE(re_fuel) TYPE ty_fuel.
ENDCLASS.
```

```
CLASS lcl_passenger_airplane DEFINITION INHERITING FROM lcl_airplane.
  PUBLIC SECTION.
    METHODS estimate_fuel_consumption REDEFINITION.
     ...
ENDCLASS.

CLASS lcl_passenger_airplane IMPLEMENTATION.
  METHOD estimate_fuel_consumption.
     ...
  ENDMETHOD.
ENDCLASS.
```

The REDEFINITION statement for the inherited method must be in the same SECTION as the definition of the original method. (It can therefore not be in the PRIVATE SECTION, since a class's private methods are not visible and therefore not redefinable in subclasses!)

If you redefine a method, you do not need to enter its interface again in the subclass, but only the name of the method. The reason for this is that ABAP Objects does not support overlaying (see notes to previous slide).

Redefining Methods: Example (2)


lcl_airplane
+ estimate_fuel_consumption ( ): fuel
lcl_passenger_airplane
+ estimate_fuel_consumption ( ): fuel
lcl_cargo_airplane
+ estimate_fuel_consumption ( ): fuel

```
METHOD estimate_fuel_consumption.
 DATA: total_weight ...
 total_weight = seats *
        average_weight + weight.
 re_fuel = total_weight *
           im_distance * factor.
 ENDMETHOD.
```

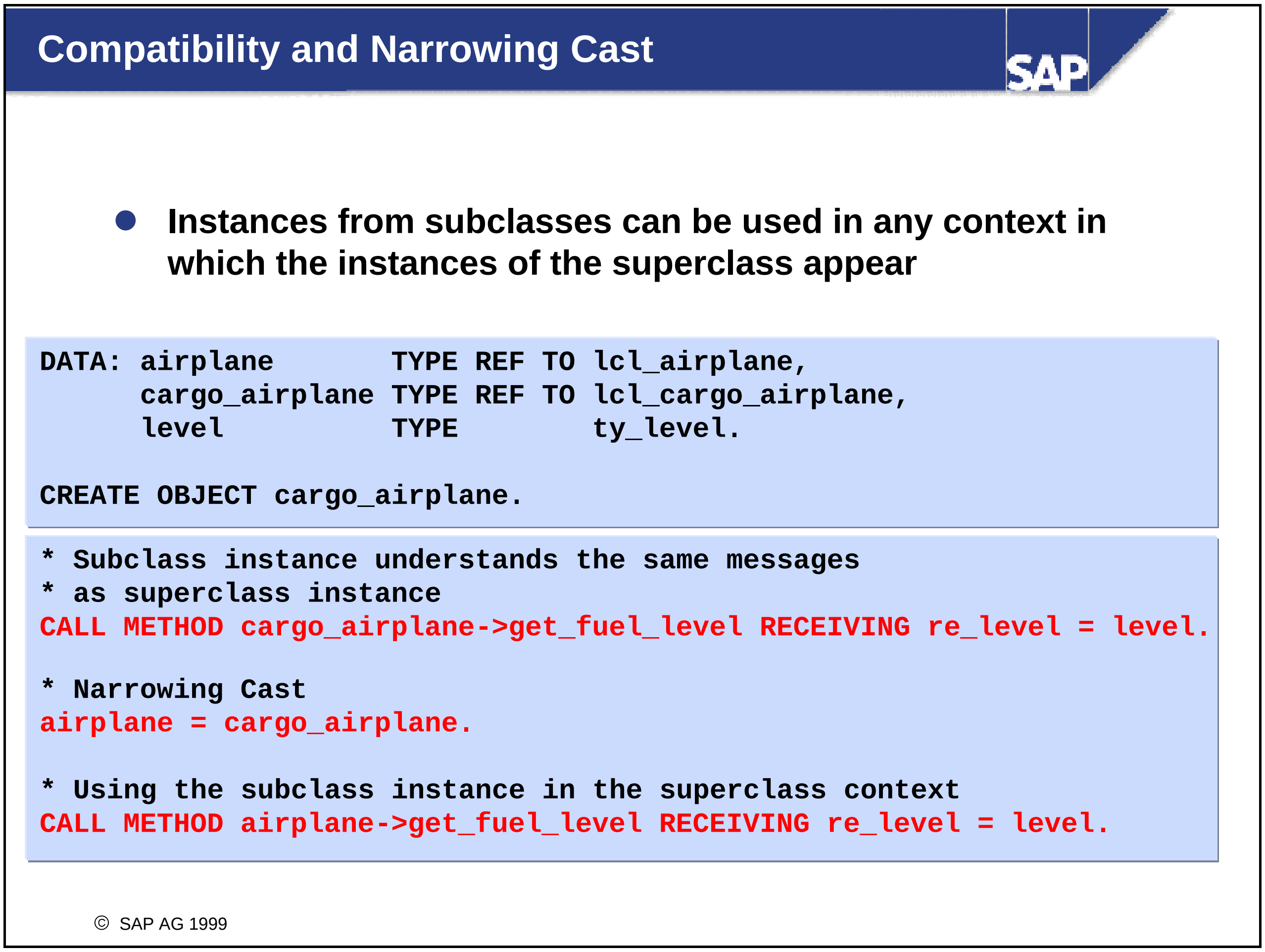
```
METHOD estimate_fuel_consumption.
  DATA: total_weight ...
  total_weight = cargo + weight.

  re_fuel = total_weight *
           im_distance * factor.
ENDMETHOD.
```

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

To implement a redefined method in a subclass, you often need to call the method of the same name in the *immediate* superclass. In ABAP Objects you can call the method from the superclass using the pseudo-reference **super**: CALL METHOD super->method_name.The pseudo-reference super can only be used in redefined methods.

Generalization/Specialization: Overview (2)

- Inheritance
- Cast
- Polymorphism
- Further Characteristics of Inheritance
- Interfaces
- Compound Interfaces



Compatibility and Narrowing Cast

- **Instances from subclasses can be used in any context in which the instances of the superclass appear**

```
DATA: airplane       TYPE REF TO lcl_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane,
      level          TYPE        ty_level.

CREATE OBJECT cargo_airplane.
```

```
* Subclass instance understands the same messages
* as superclass instance
CALL METHOD cargo_airplane->get_fuel_level RECEIVING re_level = level.

* Narrowing Cast
airplane = cargo_airplane.

* Using the subclass instance in the superclass context
CALL METHOD airplane->get_fuel_level RECEIVING re_level = level.
```

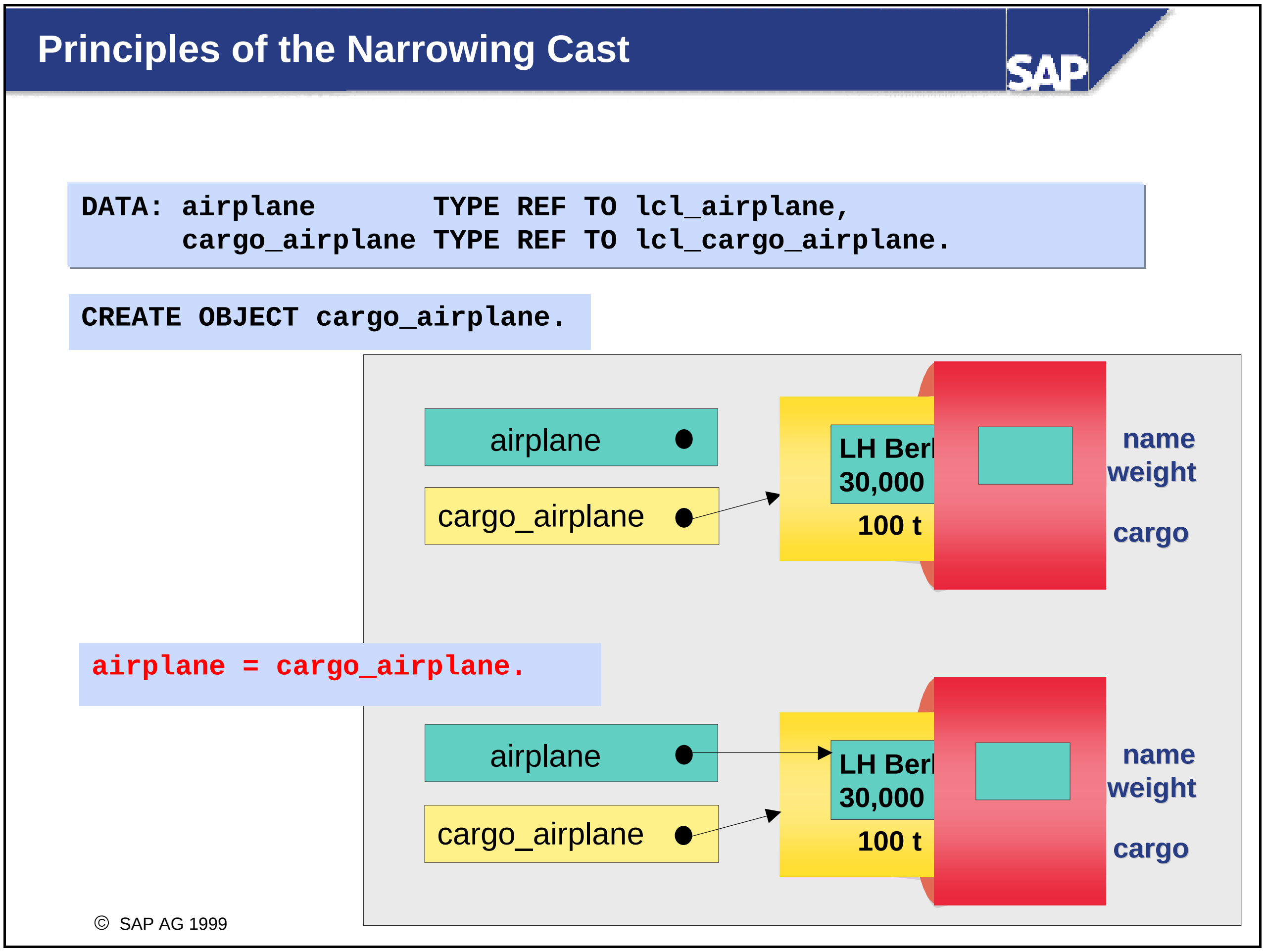
One of the significant principles of inheritance is that an instance from a subclass can be used in every context in which an instance from the superclass appears. This is possible because the subclass has inherited *all* components from the superclass and therefore has the same interface as the superclass. The user can therefore address the subclass instance in the same way as the superclass instance.

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

The assignment of a subclass instance to a reference variable of the type “reference to superclass” is described as a narrowing cast, because you are switching from a view with more detail to a view with less detail.

The description “up-cast” is also used.

What is a narrowing cast used for? A user who is not interested in the finer points of cargo or passenger planes (but only, for example, in the tank gauge) does not need to know about them. This user only needs to work with (references to) the lcl_airplane class. However, in order to allow the user to work with cargo or passenger planes, you would normally need a narrowing cast.



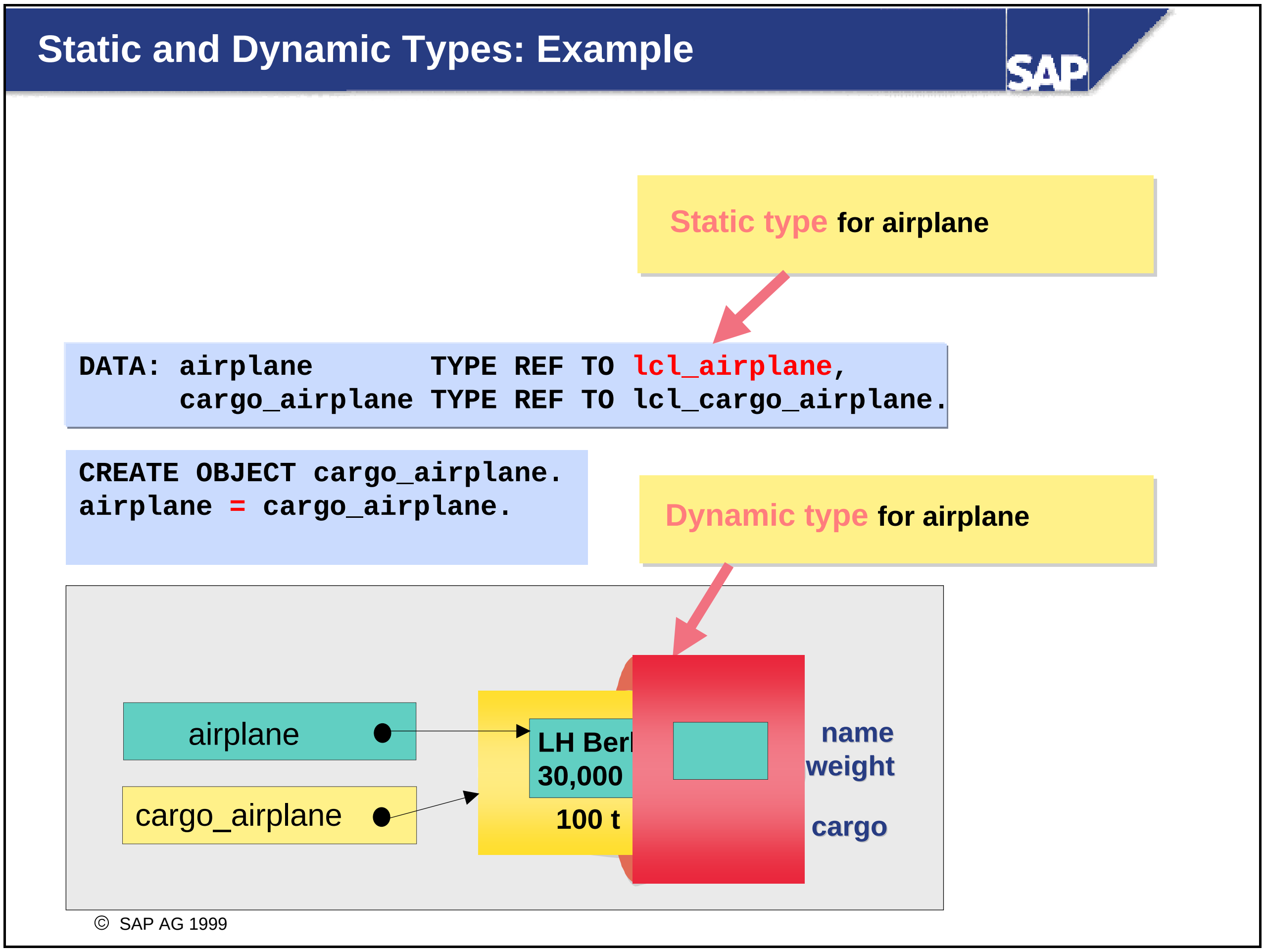
Principles of the Narrowing Cast

```
DATA: airplane      TYPE REF TO lcl_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane.
```

```
CREATE OBJECT cargo_airplane.
```

```
airplane = cargo_airplane.
```

After the narrowing cast you can use the airplane reference to access the components of the cargo plane instance that were inherited from lcl_airplane, obviously in some cases with the limitations entailed by their visibility. You can no longer access the cargo-plane-specific part of the instance (cargo in the above example) using the airplane reference.



Static and Dynamic Types: Example

Static type for airplane

```
DATA: airplane       TYPE REF TO lcl_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane.
```

```
CREATE OBJECT cargo_airplane.
airplane = cargo_airplane.
```

Dynamic type for airplane

© SAP AG 1999

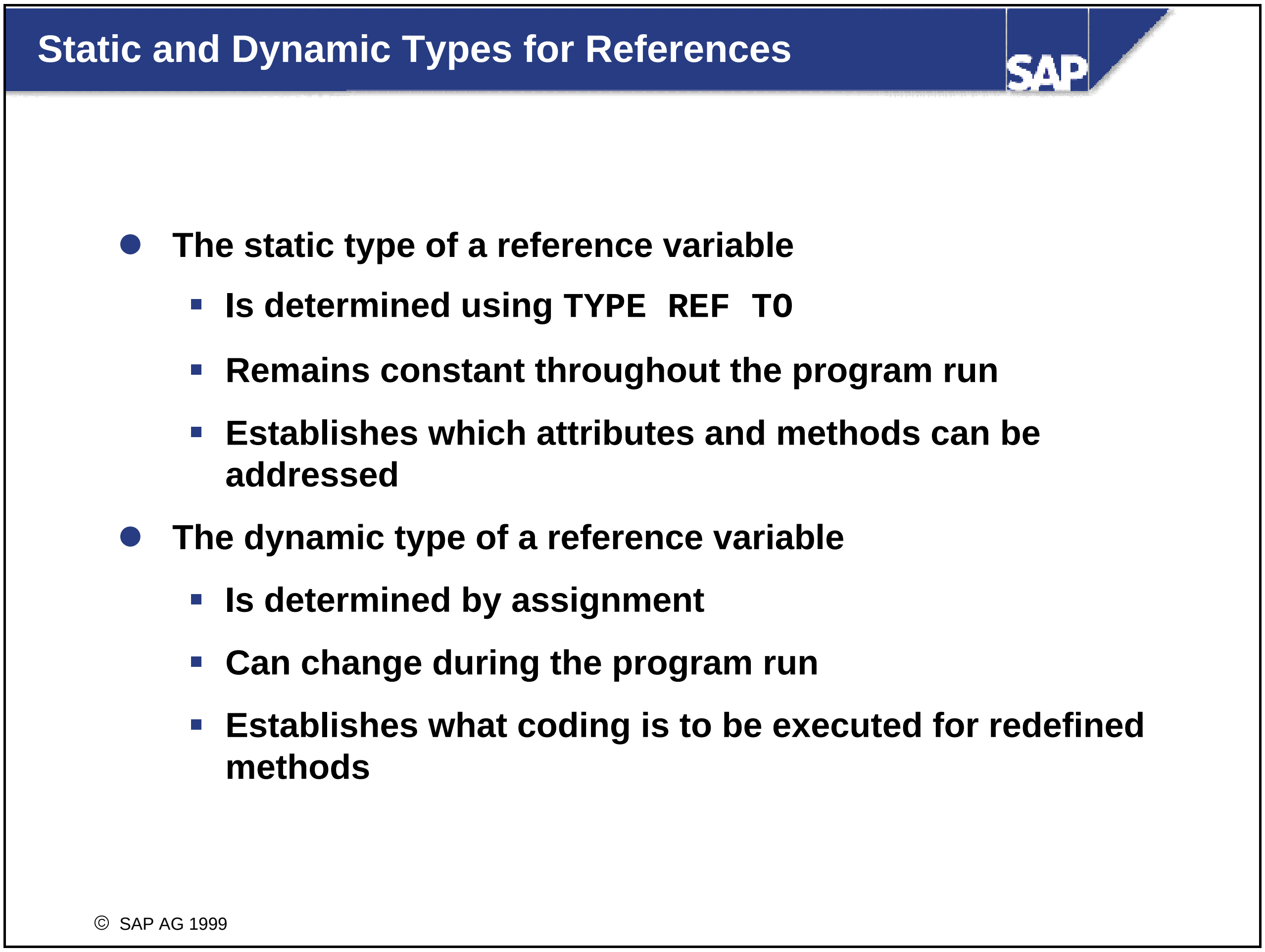
A reference variable always has two types: static and dynamic:

- The static type of a reference variable is determined by variable definition using TYPE REF TO. It cannot and does not change. It establishes which attributes and methods can be addressed

- The dynamic type of a reference variable is the type of the instance currently being referred to, it is therefore determined by assignment and can change during the program run. It establishes what coding is to be executed for redefined methods.

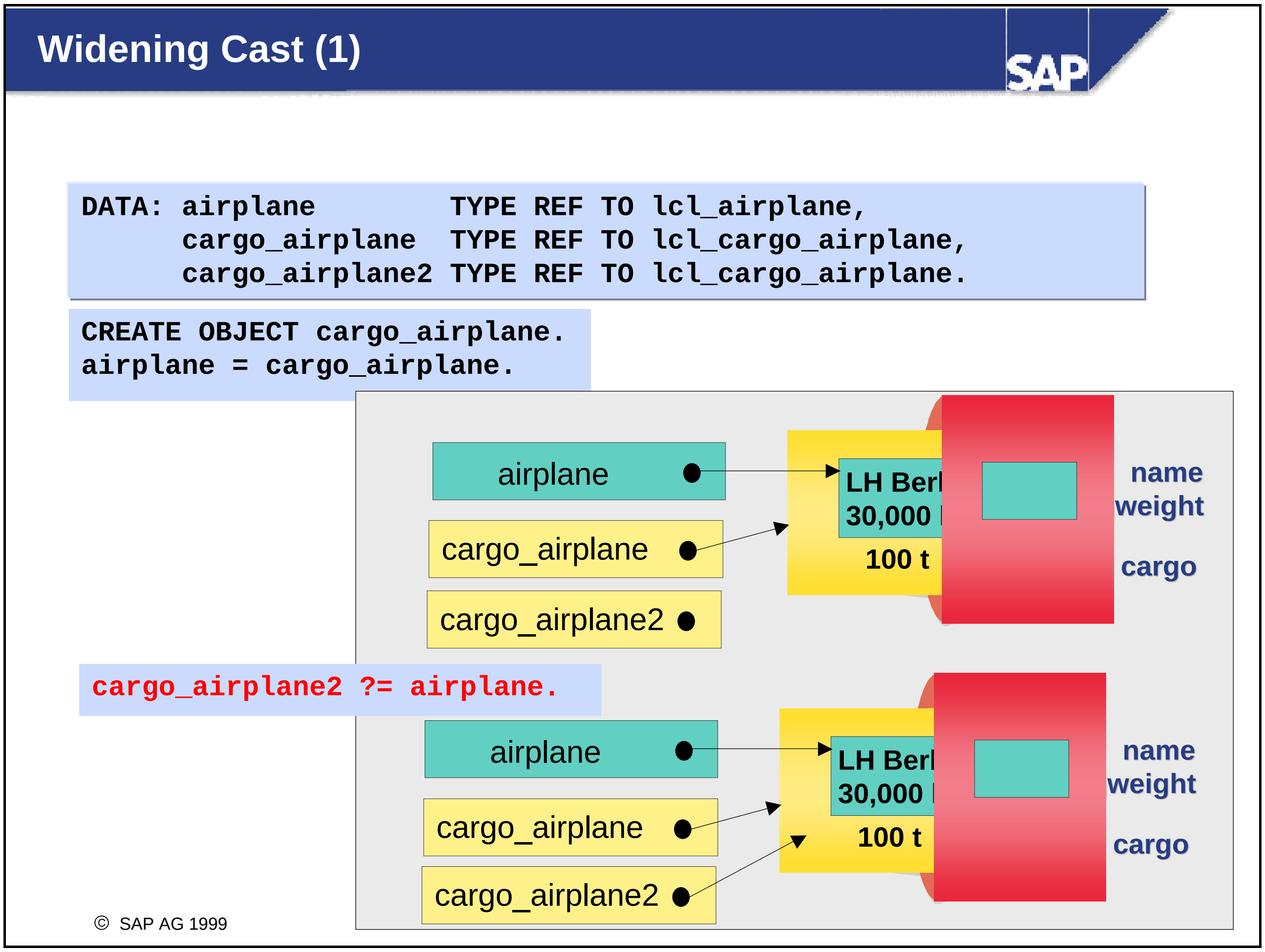
In the example, the static type of the airplane variable is always ‘REF TO lcl_airplane’, but its dynamic type after the cast is ‘REF TO lcl_cargo_airplane’.

The reference ME can be used to determine the dynamic type in the Debugger.



Static and Dynamic Types for References

- The static type of a reference variable
  - Is determined using `TYPE REF TO`
  - Remains constant throughout the program run
  - Establishes which attributes and methods can be addressed
- The dynamic type of a reference variable
  - Is determined by assignment
  - Can change during the program run
  - Establishes what coding is to be executed for redefined methods



Widening Cast (1)

```
DATA: airplane         TYPE REF TO lcl_airplane,
      cargo_airplane   TYPE REF TO lcl_cargo_airplane,
      cargo_airplane2 TYPE REF TO lcl_cargo_airplane.
```

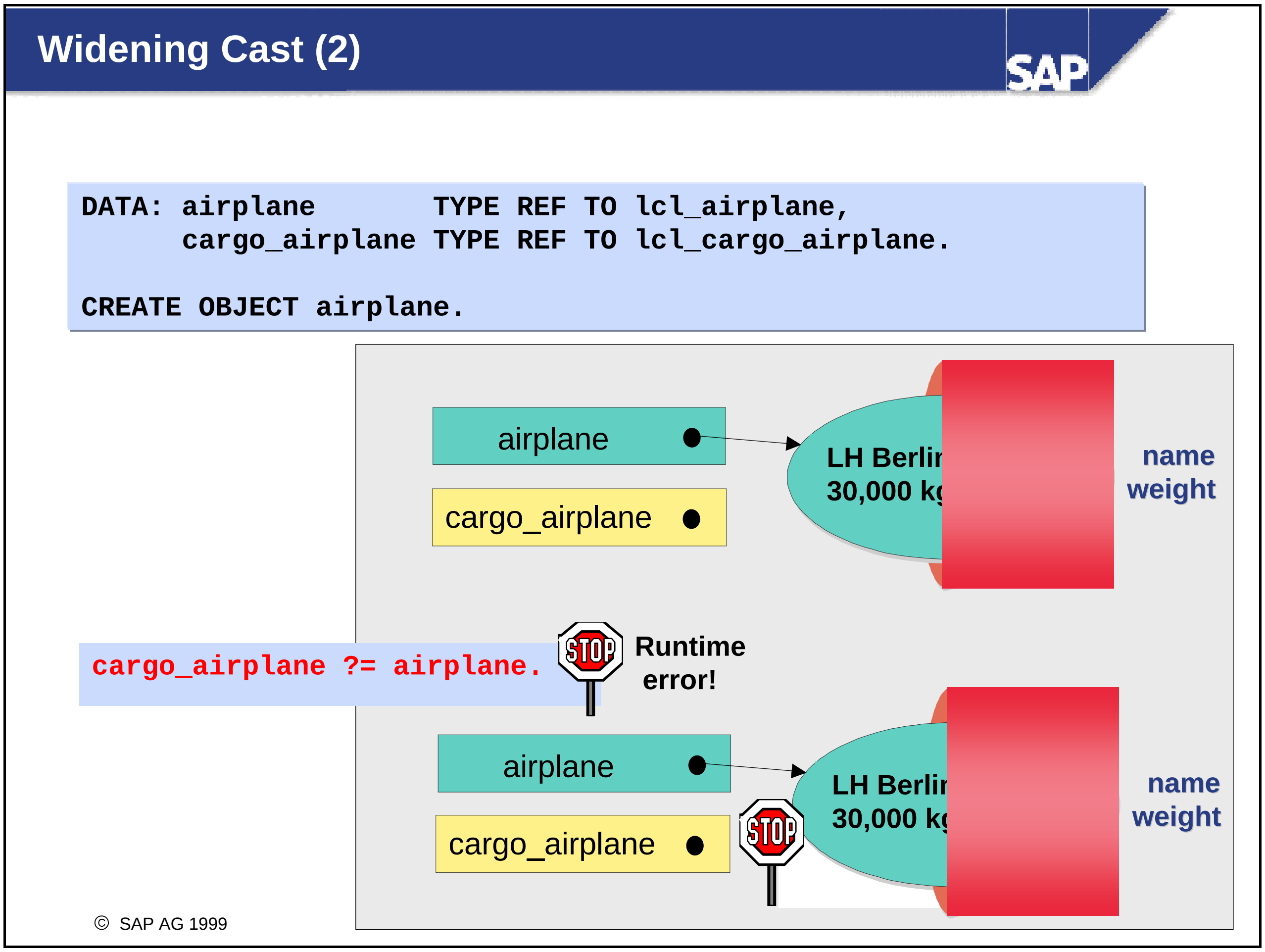
```
CREATE OBJECT cargo_airplane.
airplane = cargo_airplane.
```

```
cargo_airplane2 ?= airplane.
```

The type of case described above is known as a widening cast because it changes the view to one with more details. The instance assigned (a cargo plane in the above example) must correspond to the object reference (cargo_airplane in the above example), that is, the instance must have the details implied by the reference.

This is also known as a “down cast”.

The widening cast in this case does not cause an error because the reference airplane actually points to an instance in the subclass lcl_cargo_airplane. The dynamic type is therefore ‘REF TO lcl_cargo_airplane’.

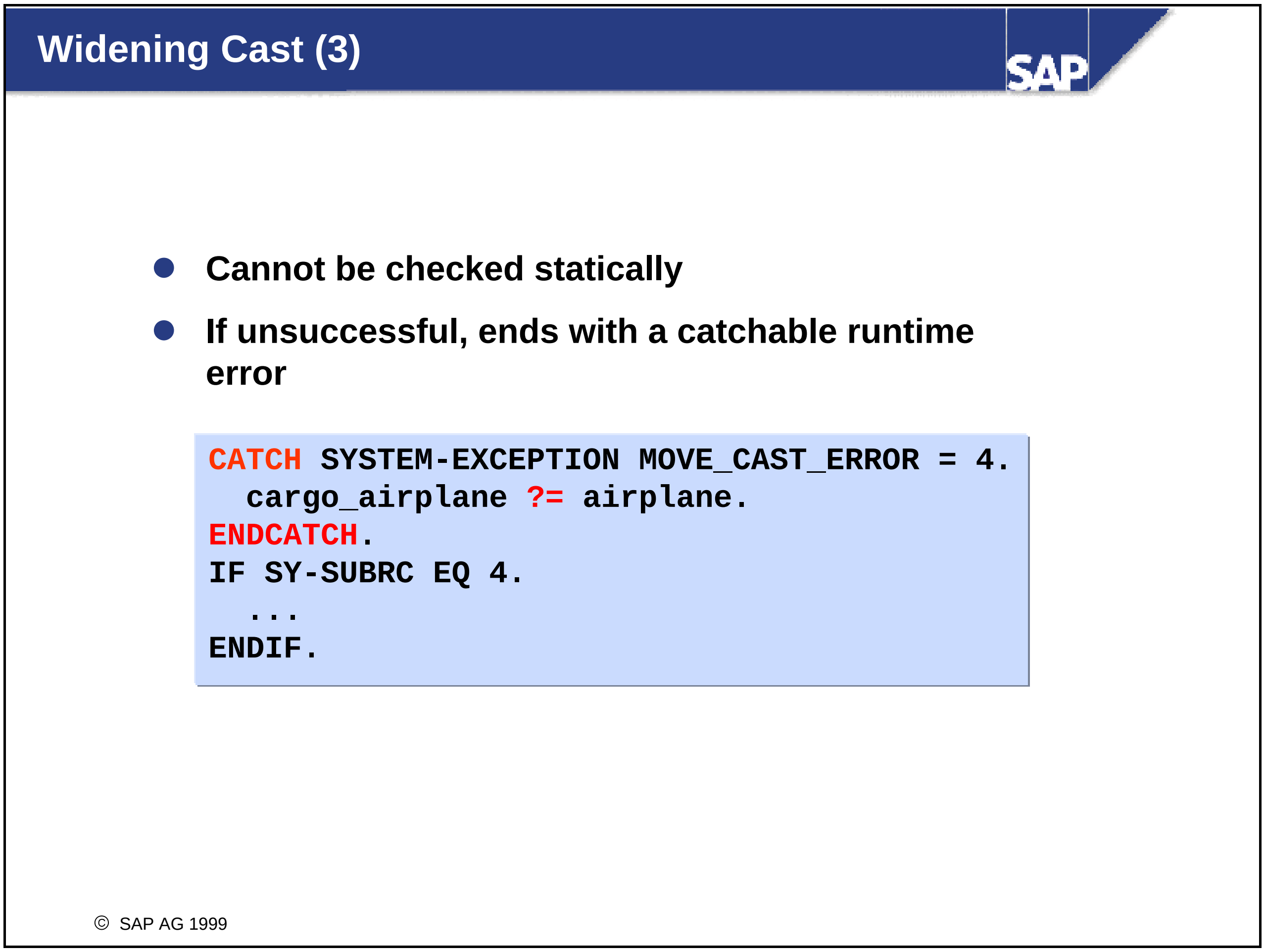


Widening Cast (2)

```
DATA: airplane        TYPE REF TO lcl_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane.

CREATE OBJECT airplane.
```

```
cargo_airplane ?= airplane.
```

Here the widening cast produces the MOVE_CAST_ERROR runtime error that can be caught with “CATCH ... ENDCATCH”, because the airplane reference does not point to an instance in the subclass lcl_cargo_airplane, but to a “general airplane object”. Its dynamic type is therefore ‘REF TO lcl_airplane’ and does not correspond to the reference type cargo_airplane.



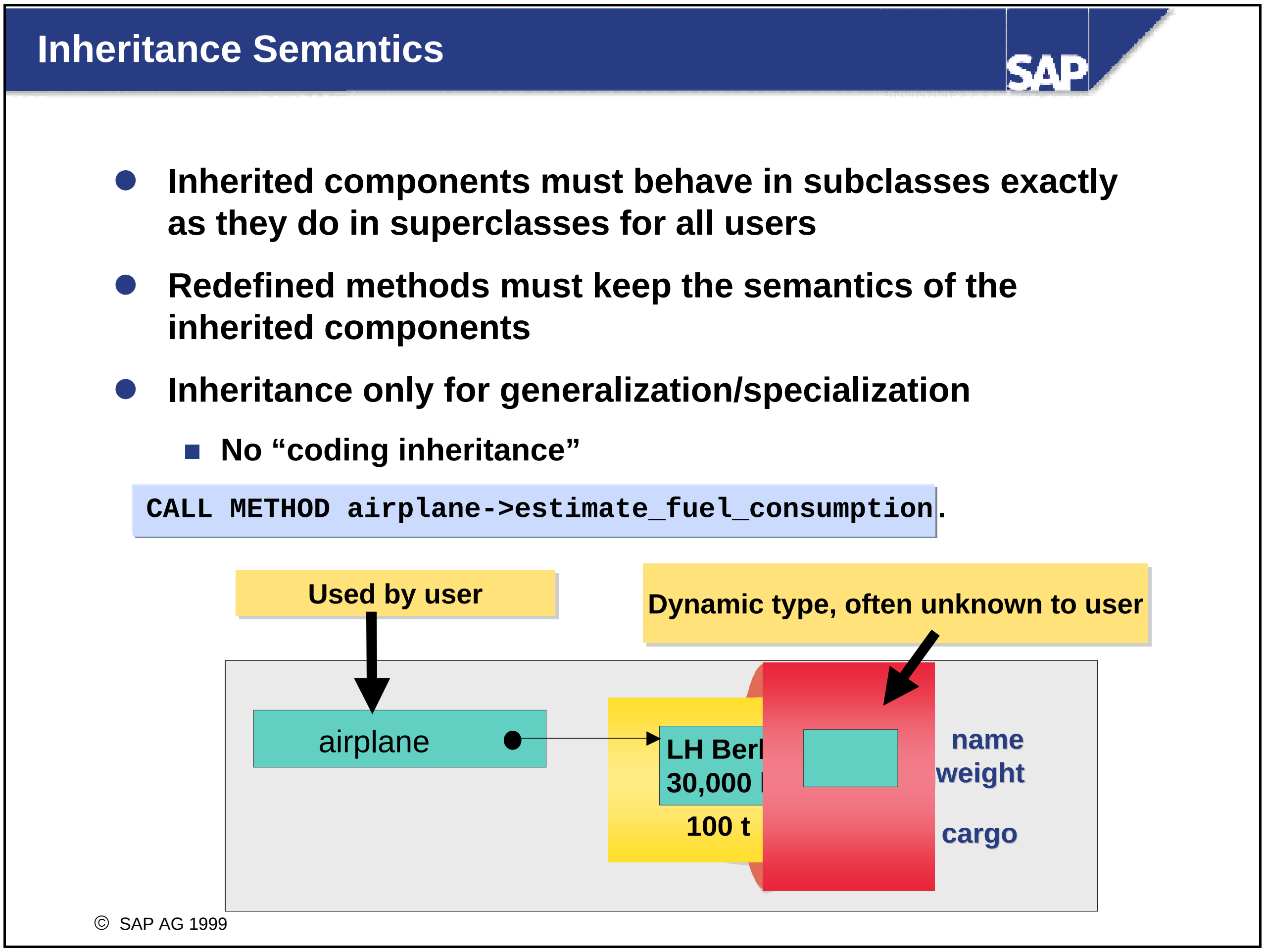
Widening Cast (3)

- Cannot be checked statically
- If unsuccessful, ends with a catchable runtime error

```
CATCH SYSTEM-EXCEPTION MOVE_CAST_ERROR = 4.
  cargo_airplane ?= airplane.
ENDCATCH.
IF SY-SUBRC EQ 4.
  ...
ENDIF.
```

The widening cast logically represents the opposite of the narrowing cast. The widening cast cannot be checked statically, only at runtime. The **Cast Operator "?="** (or the equivalent "MOVE ... ?TO …") must be used to make this visible.

With this kind of cast, a check is carried out at runtime to ensure that the current content of the source variable corresponds to the type requirements of the target variables. In this case therefore, it checks that the dynamic type of the source reference airplane is compatible with the static type of the target reference cargo_airplane. If it is, the assignment is carried out. Otherwise the catchable runtime error "MOVE_CAST_ERROR" occurs, and the original value of the target variable remains the same.



Inheritance Semantics

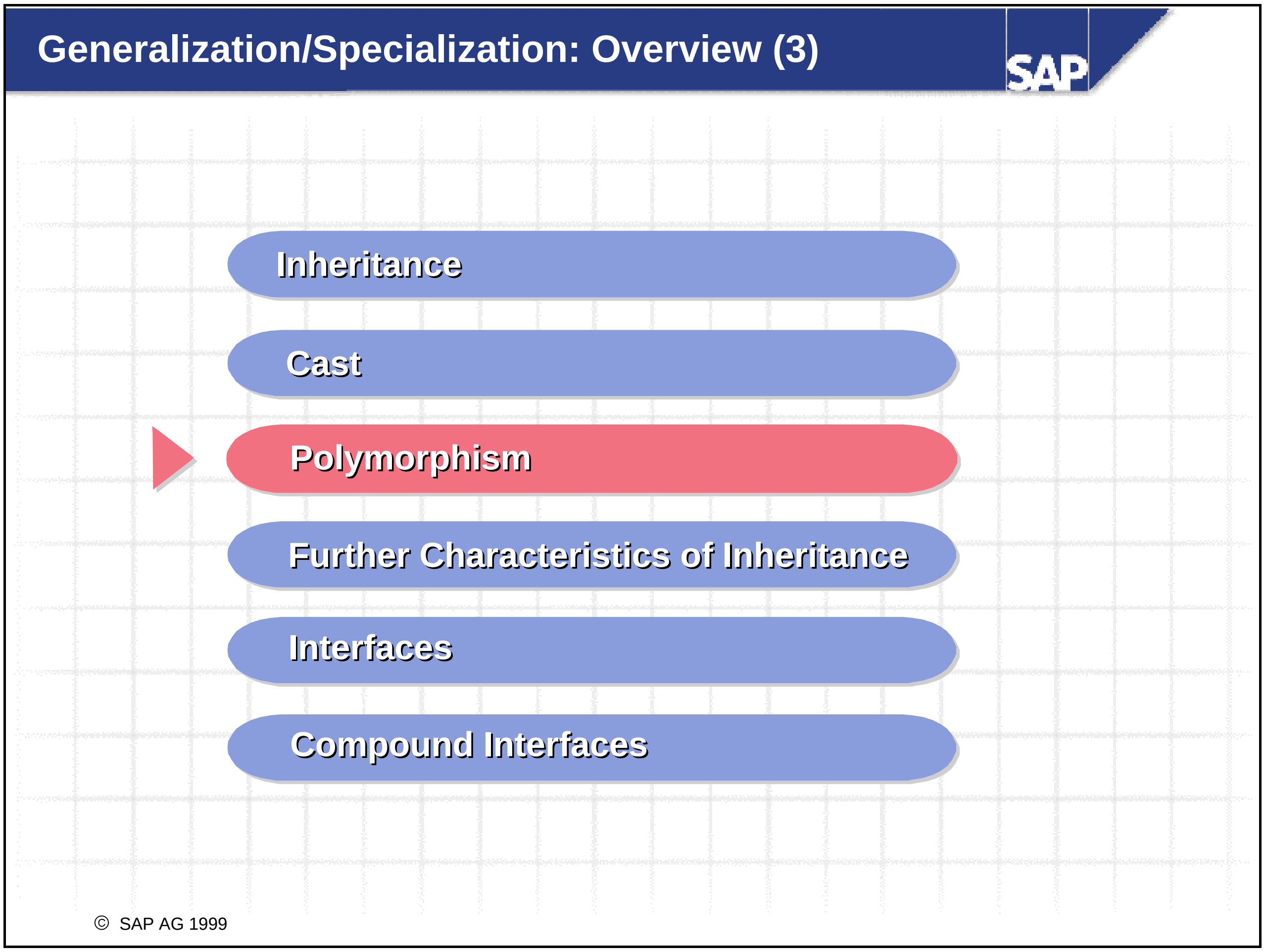
- Inherited components must behave in subclasses exactly as they do in superclasses for all users
- Redefined methods must keep the semantics of the inherited components
- Inheritance only for generalization/specialization
  - No “coding inheritance”

```
CALL METHOD airplane->estimate_fuel_consumption.
```

A subclass instance can be used in any context in which a superclass instance also appears. Moreover: the user does not and is not intended to know whether he/she is dealing with a subclass or a superclass. The user therefore works only with references to the superclass and must rely on the inherited components behaving in the subclass instances exactly as they do in the superclass instances, otherwise the program will not work!

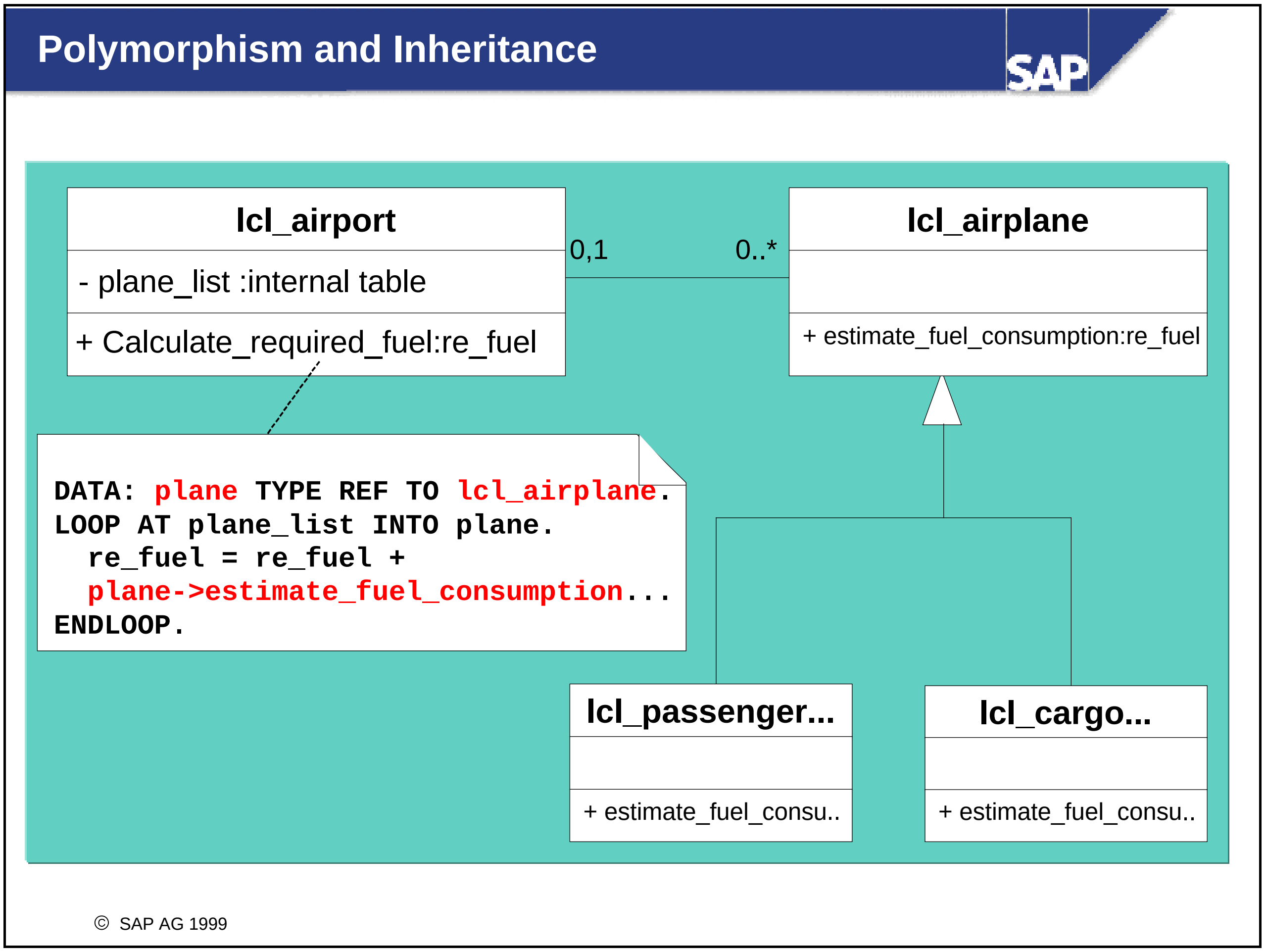
On the other hand, this ensures useful restrictions on the implementation of the subclasses: inherited components must keep their inherited semantics. You cannot use inherited attributes or events in any way other than intended in the superclass, and you cannot change method semantics by redefinition!

You must avoid coding inheritance: it is not correct for one class to inherit from another simply because *part* of the functionality required is already implemented there.



Generalization/Specialization: Overview (3)

- Inheritance
- Cast
- ▶ **Polymorphism**
- Further Characteristics of Inheritance
- Interfaces
- Compound Interfaces



Polymorphism and Inheritance

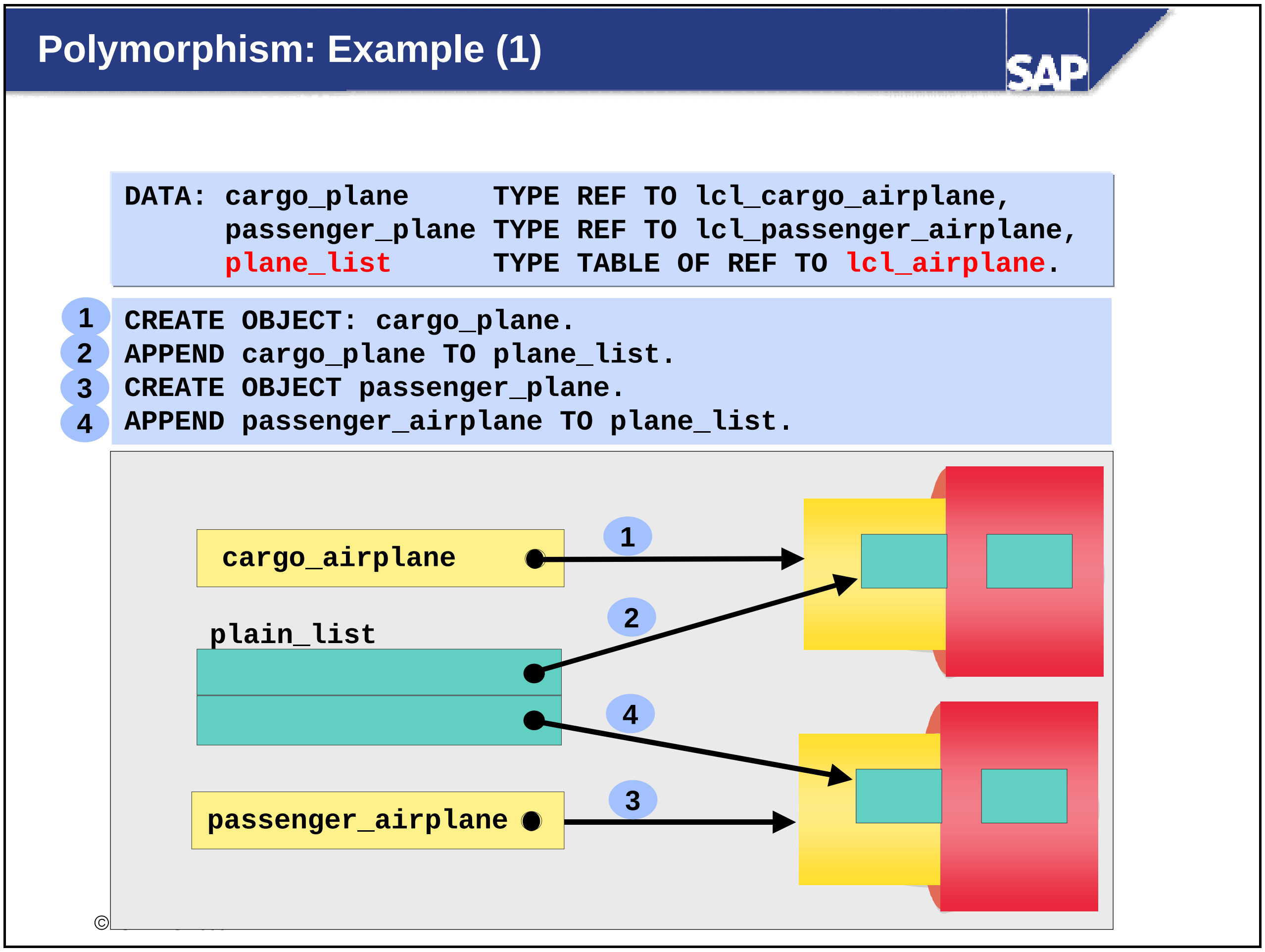
```
DATA: plane TYPE REF TO lcl_airplane.
LOOP AT plane_list INTO plane.
  re_fuel = re_fuel +
  plane->estimate_fuel_consumption...
ENDLOOP.
```

When objects from different classes react differently to the same method call, this is known as **polymorphism**. To do this, the classes implement the same method in different ways. This can done using inheritance, by redefining a method from the superclass in subclasses and implementing it differently. **Interfaces** are also introduced below: they too can enable polymorphic behavior!

When an instance receives a message to execute a particular method, then that method is executed if it has been implemented by the class the instance belongs to. If the class has not implemented that method, but only inherited and not redefined it, then a search up through the inheritance hierarchy is carried out until an implementation of that method is found.

Technically speaking, the dynamic type of the reference variable, not the static type, is used to search for the implementation of a method. In the above example of CALL METHOD plane->estimate_fuel_consumption, the class of the instance that plane actually refers to is used to search for the implementation of estimate_fuel_consumption; the static type of plane, which is always 'REF TO lcl_airplane' is not used.

Polymorphism is one of the main strengths of inheritance: the user can work in the same way with different classes, regardless of their implementation. The search for the right implementation of a method is carried out by the runtime system, not the user!

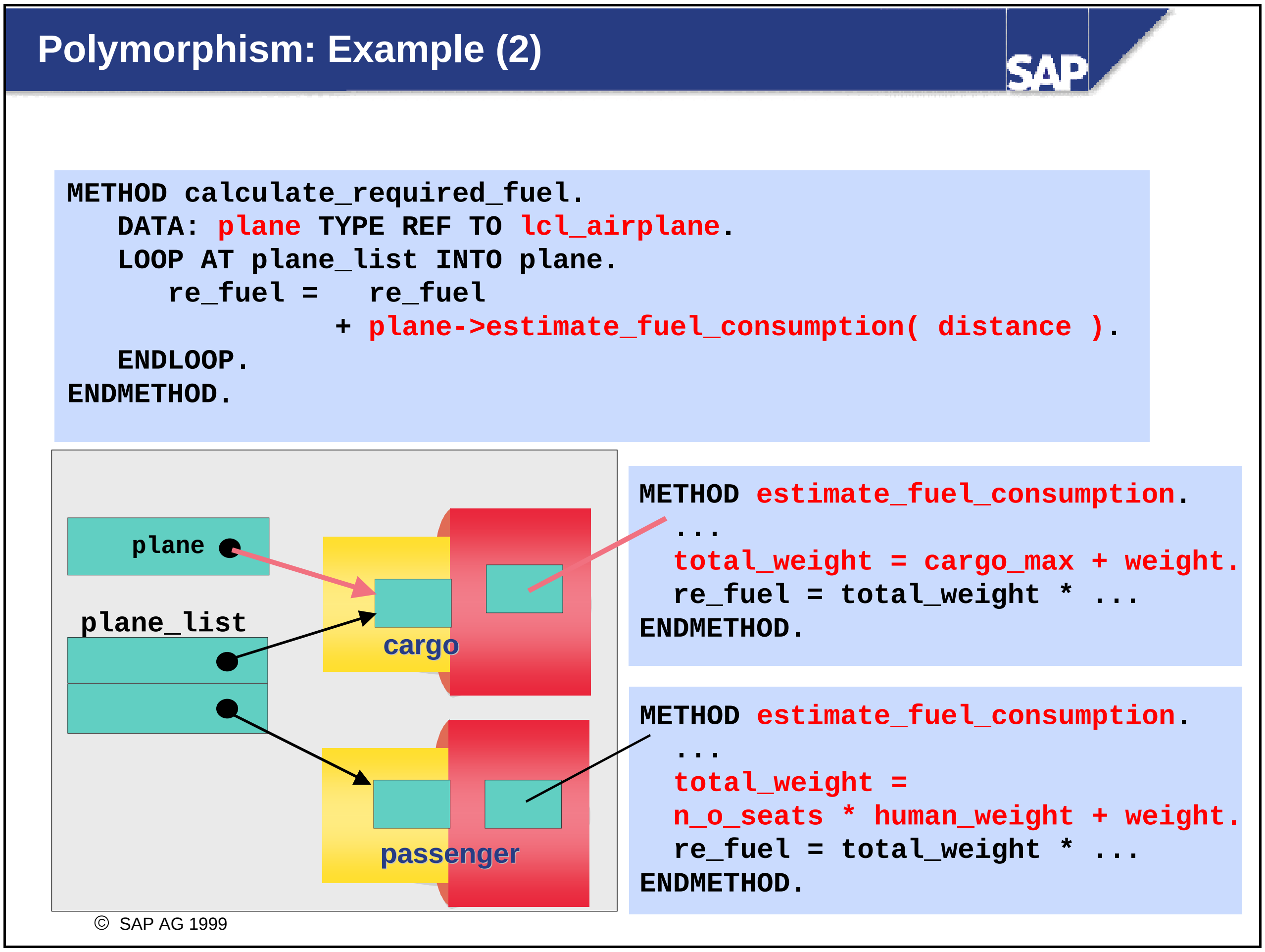


Polymorphism: Example (1)

```
DATA: cargo_plane     TYPE REF TO lcl_cargo_airplane,
      passenger_plane TYPE REF TO lcl_passenger_airplane,
      plane_list      TYPE TABLE OF REF TO lcl_airplane.
```

```
1 CREATE OBJECT: cargo_plane.
2 APPEND cargo_plane TO plane_list.
3 CREATE OBJECT passenger_plane.
4 APPEND passenger_airplane TO plane_list.
```

Objects from different classes (lcl_cargo_airplane and lcl_passenger_airplane in the above example) can be stored in an internal table consisting of references to the superclass (lcl_airplane in the above example, and then processed identically (polymorphically) (see next slide).



Polymorphism: Example (2)

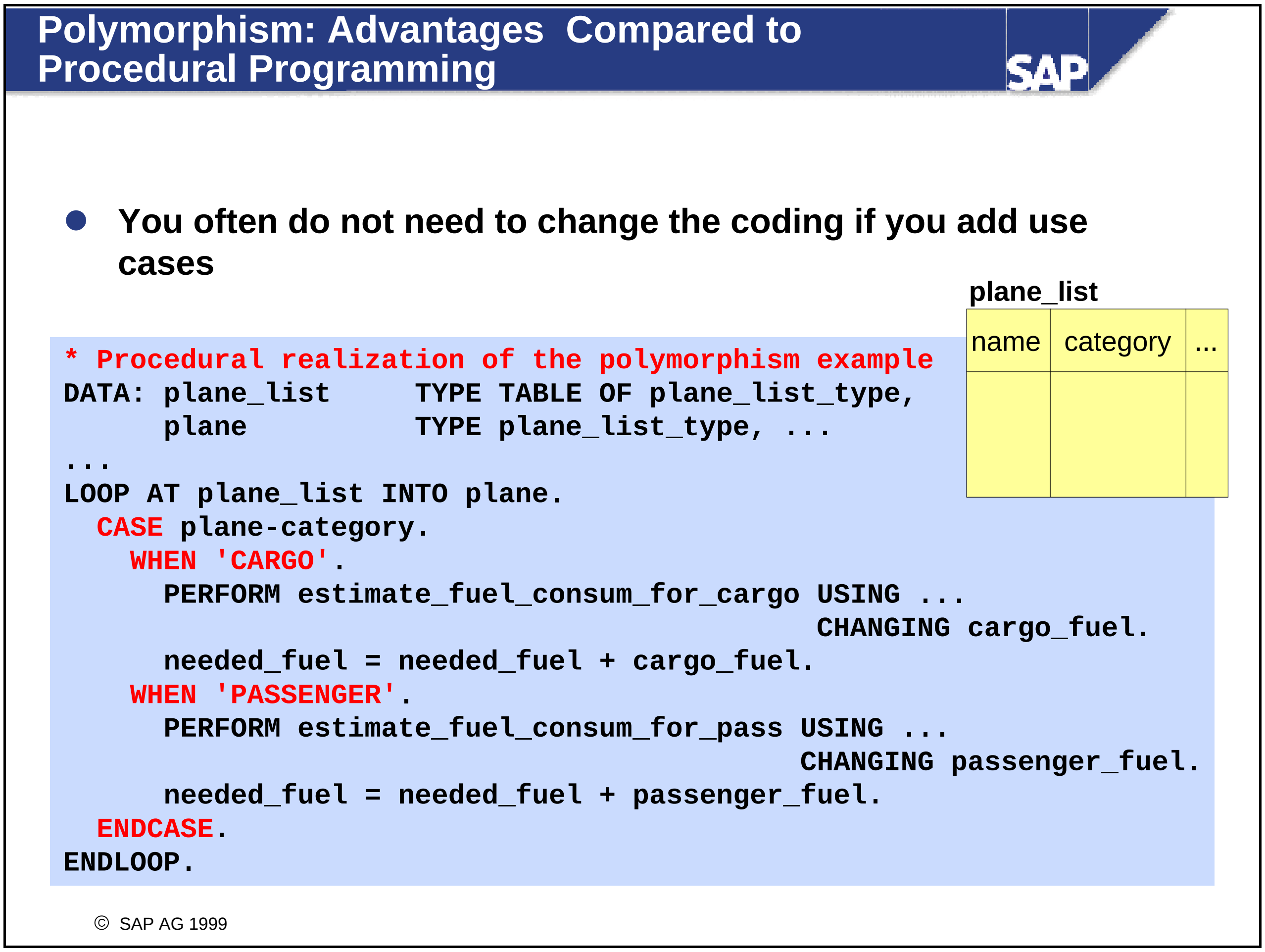
```
METHOD calculate_required_fuel.
   DATA: plane TYPE REF TO lcl_airplane.
   LOOP AT plane_list INTO plane.
      re_fuel =   re_fuel
                + plane->estimate_fuel_consumption( distance ).
   ENDLOOP.
ENDMETHOD.
```

```
METHOD estimate_fuel_consumption.
  ...
  total_weight = cargo_max + weight.
  re_fuel = total_weight * ...
ENDMETHOD.
```

```
METHOD estimate_fuel_consumption.
  ...
  total_weight =
  n_o_seats * human_weight + weight.
  re_fuel = total_weight * ...
ENDMETHOD.
```

What coding is actually executed when estimate_fuel_consumption is called depends on the dynamic type of the plane reference variable, that is, it depends on which object from which (sub)class plane points to.

You can use polymorphism to write programs that are generic to a high degree and that do not even need to be changed if use cases are added. In the simple example above, this means that, should a further subclass be added, for example, for airplanes that fly in space, the above coding would not need to be changed.



Polymorphism: Advantages Compared to Procedural Programming

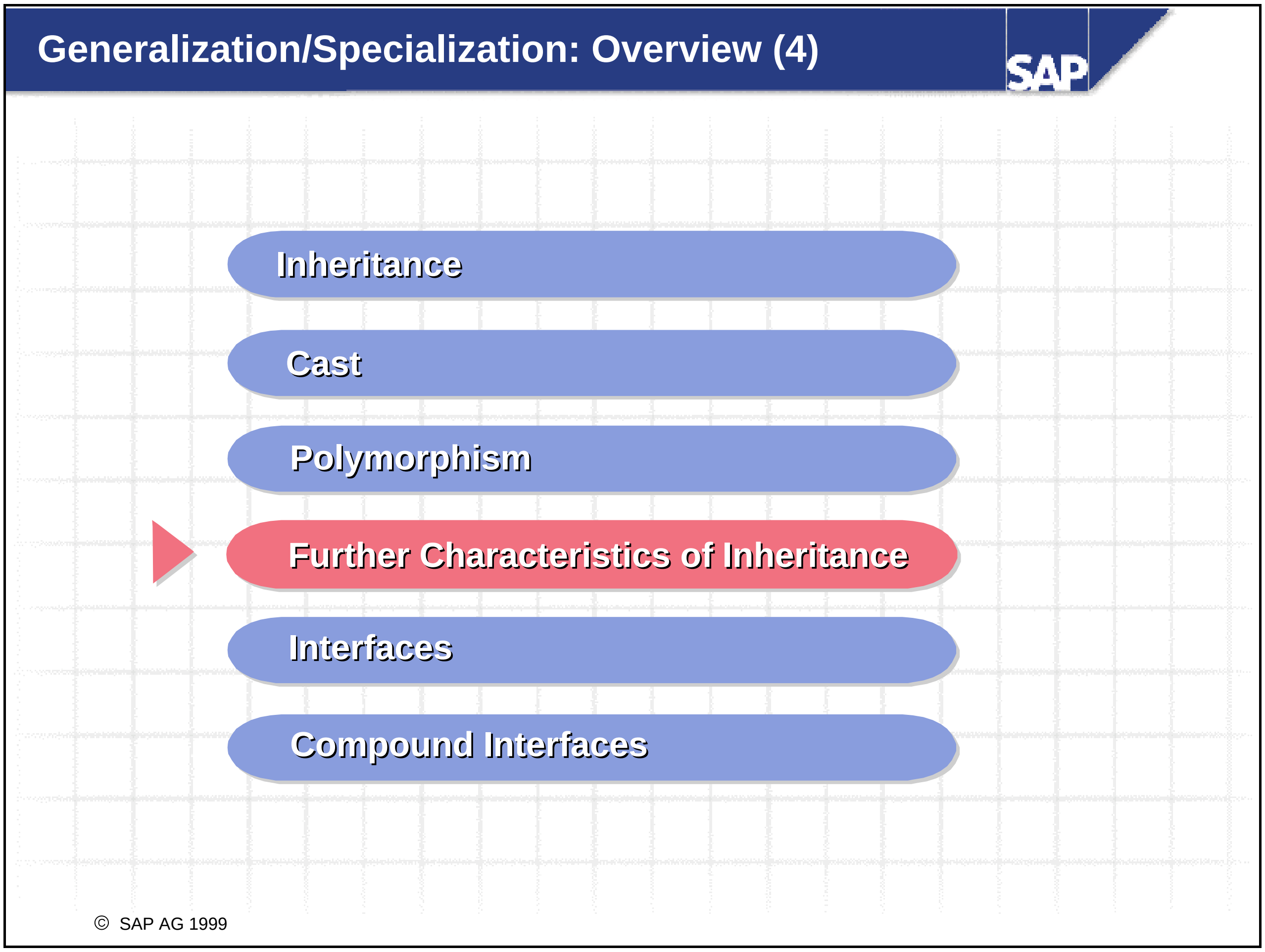
- You often do not need to change the coding if you add use cases

plane_list

name	category	...

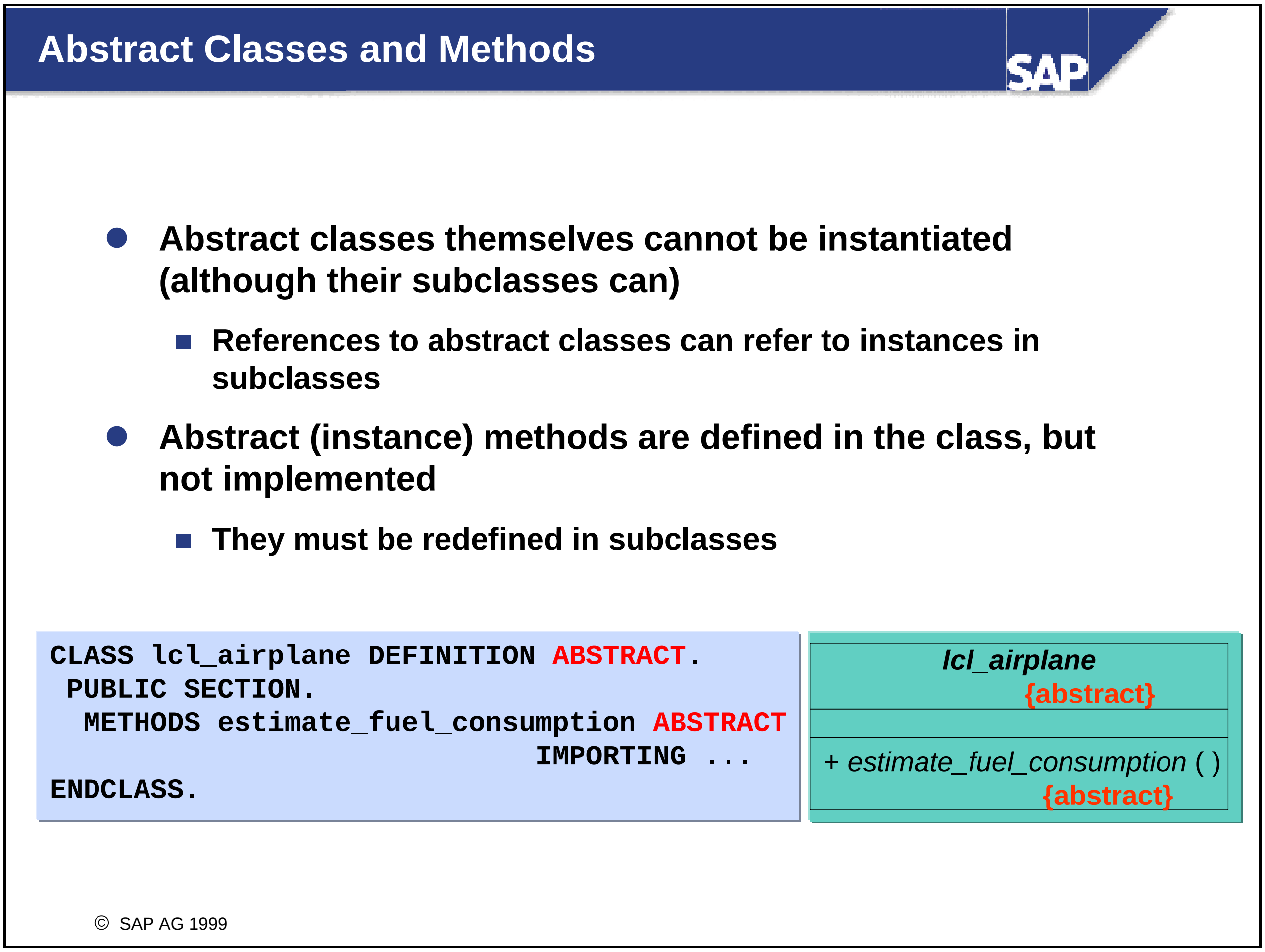
```
* Procedural realization of the polymorphism example
DATA: plane_list     TYPE TABLE OF plane_list_type,
      plane          TYPE plane_list_type, ...
...
LOOP AT plane_list INTO plane.
  CASE plane-category.
    WHEN 'CARGO'.
      PERFORM estimate_fuel_consum_for_cargo USING ...
                                       CHANGING cargo_fuel.
      needed_fuel = needed_fuel + cargo_fuel.
    WHEN 'PASSENGER'.
      PERFORM estimate_fuel_consum_for_pass USING ...
                                      CHANGING passenger_fuel.
      needed_fuel = needed_fuel + passenger_fuel.
  ENDCASE.
ENDLOOP.
```

Using polymorphism makes generic programming easier. Instead of implementing a CASE or IF statement, you can have one access or call, which improves readability and does not need to be changed if you extend the program by adding further subclasses.



Generalization/Specialization: Overview (4)

- Inheritance
- Cast
- Polymorphism
- Further Characteristics of Inheritance
- Interfaces
- Compound Interfaces



Abstract Classes and Methods

- Abstract classes themselves cannot be instantiated (although their subclasses can)
  - References to abstract classes can refer to instances in subclasses
- Abstract (instance) methods are defined in the class, but not implemented
  - They must be redefined in subclasses

```
CLASS lcl_airplane DEFINITION ABSTRACT.
 PUBLIC SECTION.
  METHODS estimate_fuel_consumption ABSTRACT
                             IMPORTING ...
ENDCLASS.
```

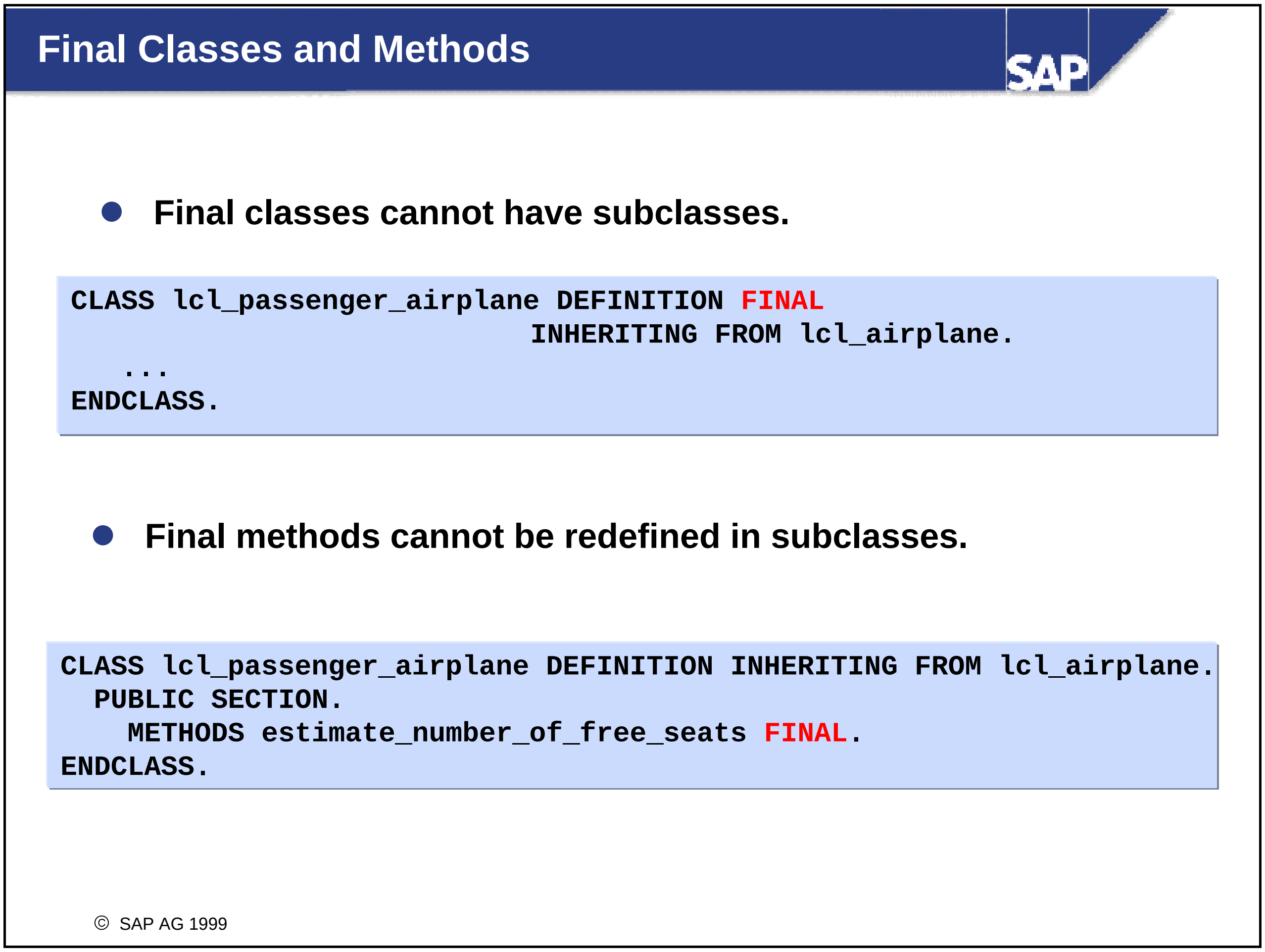
You cannot instantiate objects in an abstract class. This does not, however, mean that references to such classes are meaningless. On the contrary, they are very useful, since they can (and must) refer to instances in subclasses of the abstract class during runtime. The CREATE-OBJECT statement is extended in this context. You can enter the class of the instance to be created explicitly:
CREATE OBJECT <RefToAbstractClass> TYPE <NonAbstractSubclassName>.

Abstract classes are normally used as an incomplete blueprint for concrete (that is, non-abstract) subclasses, in order to define a uniform interface, for example.

Abstract instance methods are used to specify particular interfaces for subclasses, without having to immediately provide implementation for them. Abstract methods need to be redefined and thereby implemented in the subclass (here you also need to include the corresponding redefinition statement in the DEFINITION part of the subclass).

Classes with at least one abstract method are themselves abstract

Static methods and constructors cannot be abstract (they cannot be redefined).



Final Classes and Methods

- Final classes cannot have subclasses.

```
CLASS lcl_passenger_airplane DEFINITION FINAL
                           INHERITING FROM lcl_airplane.
   ...
ENDCLASS.
```

- Final methods cannot be redefined in subclasses.

```
CLASS lcl_passenger_airplane DEFINITION INHERITING FROM lcl_airplane.
  PUBLIC SECTION.
    METHODS estimate_number_of_free_seats FINAL.
ENDCLASS.
```

A final class cannot have subclasses, and can protect itself in this way against (uncontrolled) specialization.

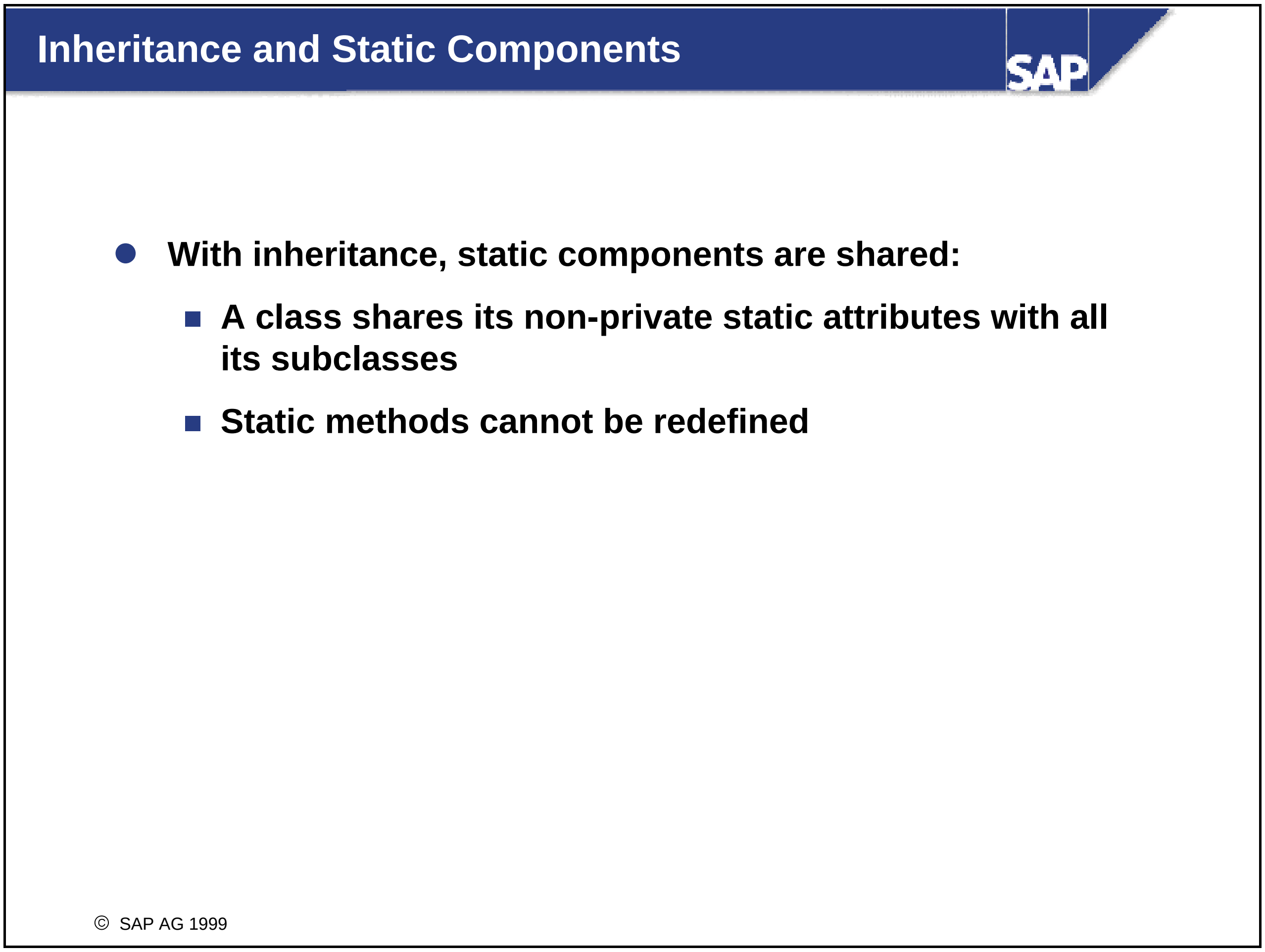
A final method in a class cannot be redefined in a subclass, and can protect itself in this way against (uncontrolled) redefinition.

Some characteristics:

A final class implicitly only contains final methods. You cannot enter FINAL explicitly for these methods in this case.

Methods cannot be both final and abstract.

Classes, on the other hand, can usefully be both abstract and final: only static components can be used there.



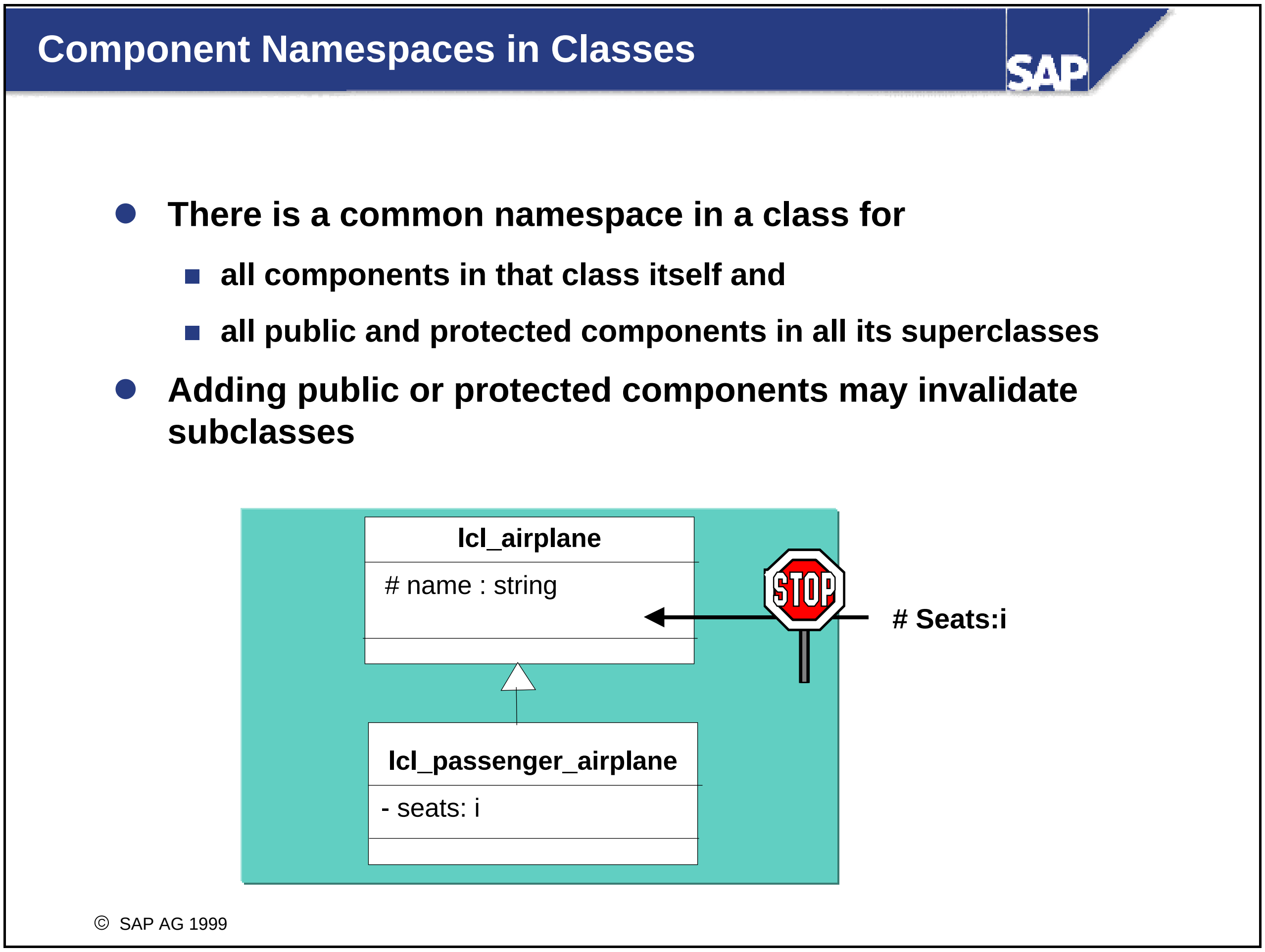
Inheritance and Static Components

- With inheritance, static components are shared:
  - A class shares its non-private static attributes with all its subclasses
  - Static methods cannot be redefined

In ABAP Objects, all static components in an inheritance relationship are shared.

For attributes this means that each static attribute only exists once per roll area. In this way a class that defines a public or protected static attribute shares this attribute with all its subclasses. The significant point here is that subclasses do not each receive their own copy of the static attribute.

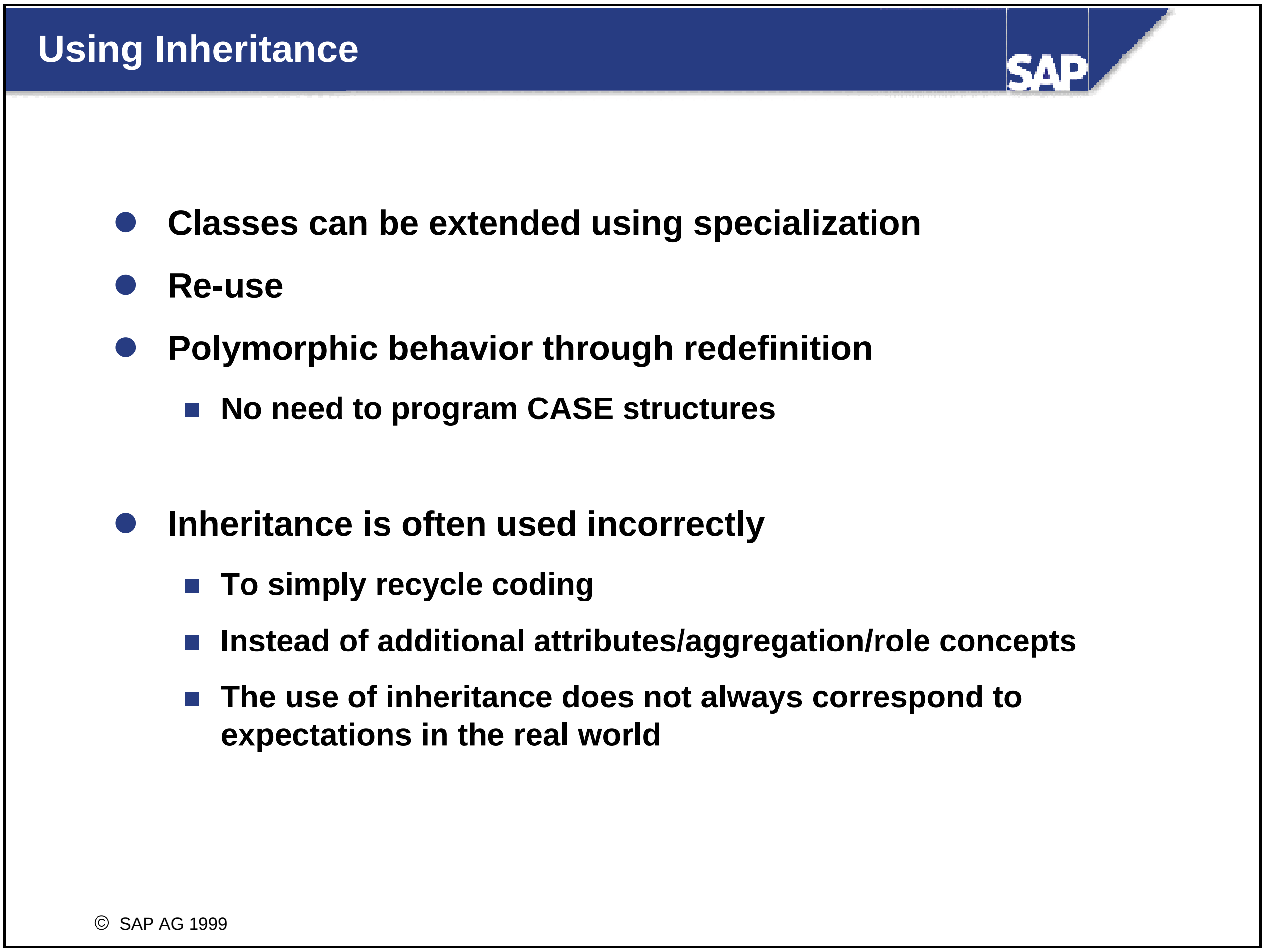
Shared static methods cannot be redefined in subclasses. However, you can call inherited (public or protected) static methods using subclasses.



Component Namespaces in Classes

- There is a common namespace in a class for
  - all components in that class itself and
  - all public and protected components in all its superclasses
- Adding public or protected components may invalidate subclasses

Adding private components is never a problem. Adding public or protected components to a class can however invalidate that class's subclasses, if they already contain components with the same name. When you add that component, you get the syntax error message that that component has already been declared.



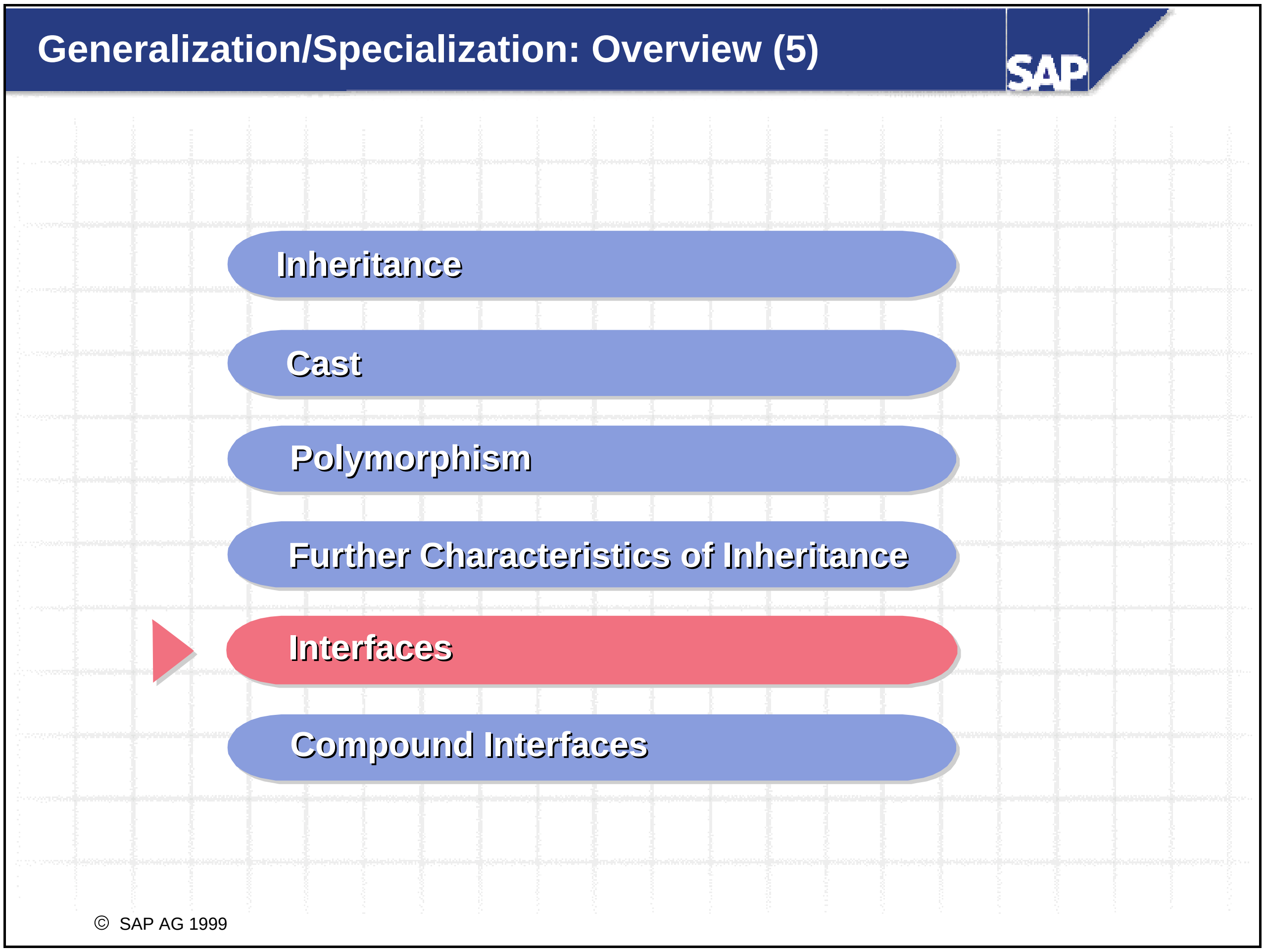
Using Inheritance

- Classes can be extended using specialization
- Re-use
- Polymorphic behavior through redefinition
  - No need to program CASE structures
- Inheritance is often used incorrectly
  - To simply recycle coding
  - Instead of additional attributes/aggregation/role concepts
  - The use of inheritance does not always correspond to expectations in the real world

Using inheritance instead of attributes, or a misunderstanding of inheritance as an “is-one” relationship often leads to the following kind of design: the superclass “car” has the subclasses “red car”, “green car”, and so on. These subclasses all have *an identical structure and identical behavior*.

As an instance cannot change its class, in circumstances like the following, you should not use inheritance directly, but use additional attributes to differentiate between cases (see appendix):
The class “employee” has the subclasses “full-time employee” and “part-time employee”. What happens when a part-time employee becomes a full-time employee? A new full-time-employee object would have to instantiated and all the information copied from the part-time-employee object. However, users who still have a reference to the part-time-employee instance would then be working with a part-time-employee object that logically does not exist anymore!

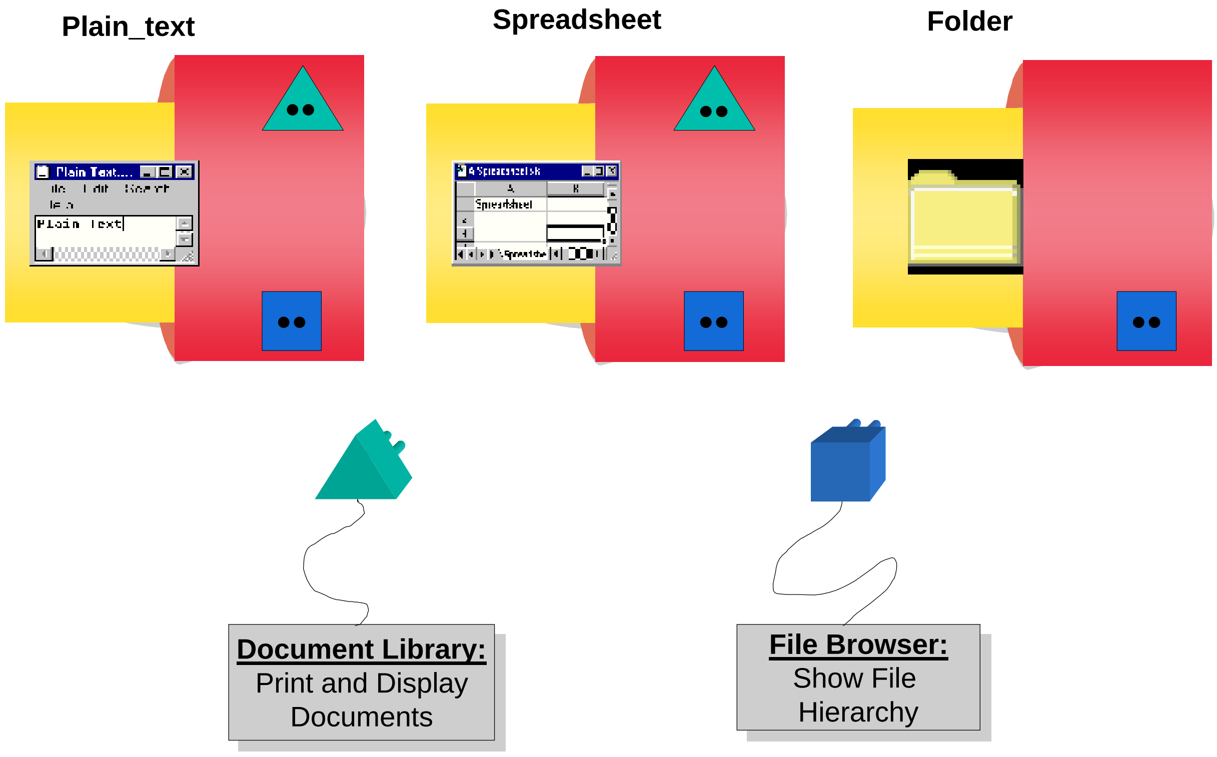
The use of inheritance does not always correspond to expectations in the real world: for example, if the class ‘square’ inherits from the class ‘rectangle’, the latter will have two separate methods for changing length and width, although the sides of the square actually need to be changed by the same measurement.



Generalization/Specialization: Overview (5)

- Inheritance
- Cast
- Polymorphism
- Further Characteristics of Inheritance
- Interfaces
- Compound Interfaces

Interfaces



Plain_text

Spreadsheet

Folder

Document Library:
Print and Display
Documents

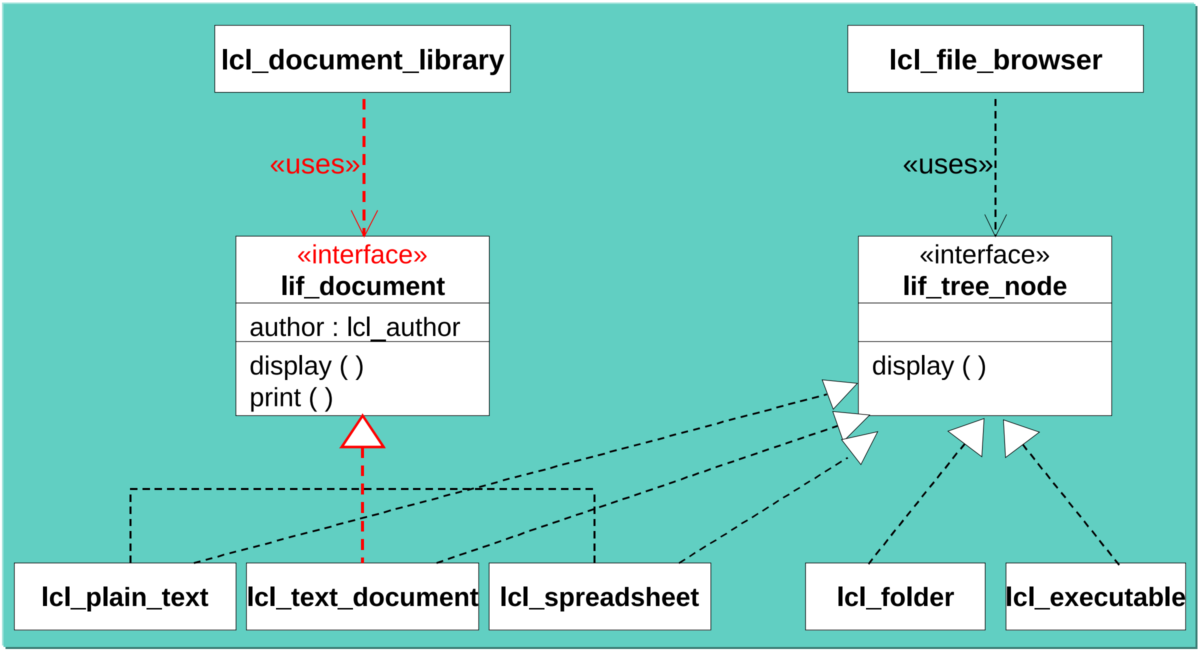
File Browser:
Show File
Hierarchy

In ABAP Objects, interfaces are implemented in addition to and independently of classes. Interfaces exclusively describe the external point of contact of a class, but they do not contain their own implementation part.

Interfaces are usually defined by a user. The user describes in the interface which services (technical and semantic) it needs in order to carry out a task. The user never actually knows the providers, but communicates with them through the interface. In this way the user is protected from actual implementations and can work in the same way with different classes/objects, as long as they provide the services required (this is **polymorphism** using interfaces).

The above example shows two users: the document library and the file browser. Both define the tasks that potential providers must be able to carry out: display and print in the first case and show a node for the file folder display in the second. Various providers (plain text files, spreadsheets) can perform all the services required, but one provider (the folder) can only perform the service required by the File Browser and can therefore not be used by the Document Library.

Interfaces in UML

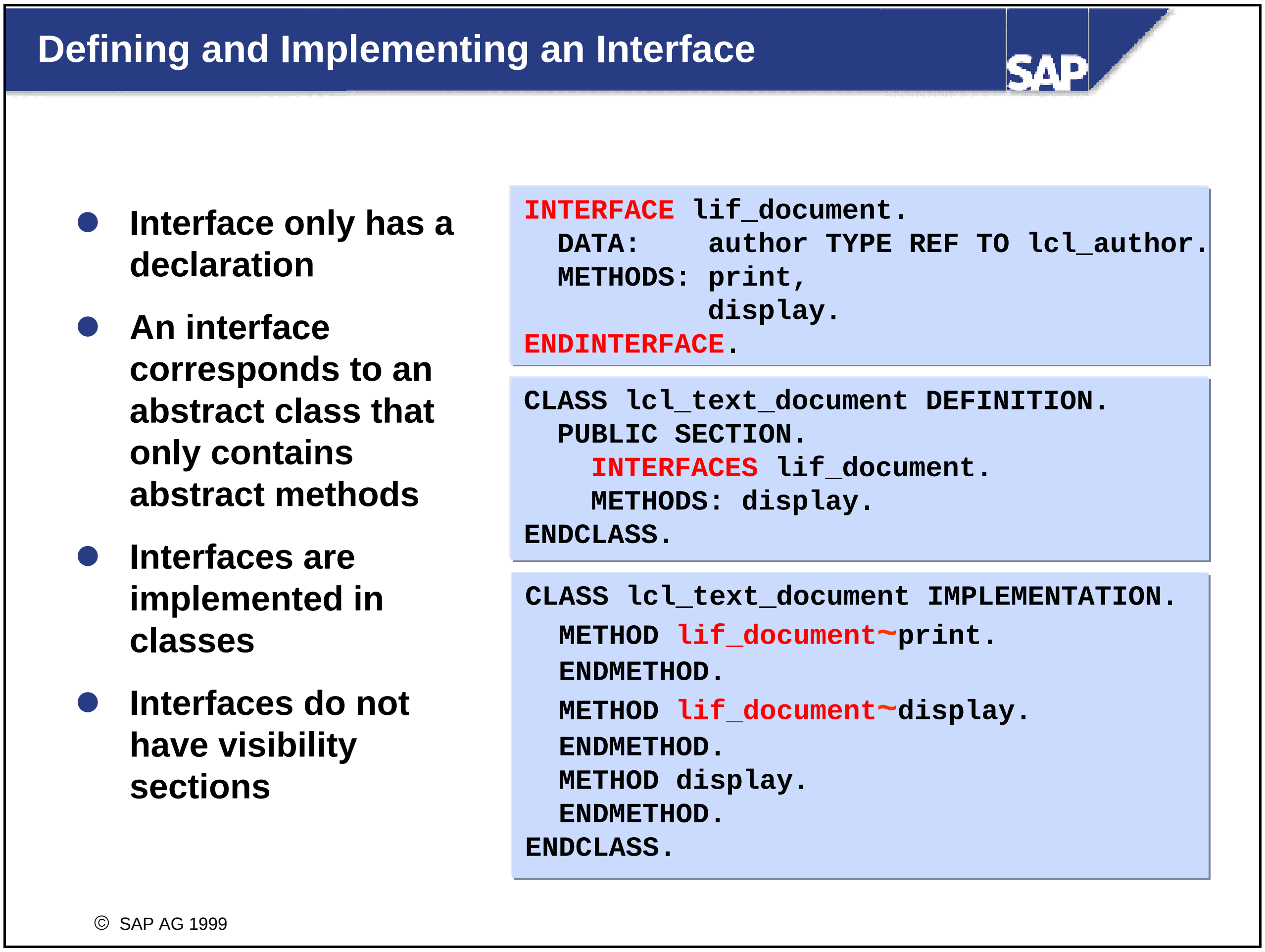


In UML, interfaces can represented in the same way as classes. However, they always have stereotype «interface» above their name and can therefore be told apart from classes.

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 line with a three-sided arrow from the class to the interface. The similarity to the representation of inheritance is intentional, as the interface can be seen as a generalization of the class implemented or the class can be seen as a specialization of the interface.

In ABAP Objects, the same components can be defined in interfaces and in classes. This allows you to shift part of the public point of contact of a class into an interface, even though the class is already in use; users will not notice the difference as long as you use alias names (see appendix) for the components that are now in the interface.

A class can implement any number of interfaces and an interface can be implemented by any number of classes.



Defining and Implementing an Interface

- Interface only has a declaration
- An interface corresponds to an abstract class that only contains abstract methods
- Interfaces are implemented in classes
- Interfaces do not have visibility sections

```
INTERFACE lif_document.
  DATA:    author TYPE REF TO lcl_author.
  METHODS: print,
           display.
ENDINTERFACE.
```

```
CLASS lcl_text_document DEFINITION.
  PUBLIC SECTION.
    INTERFACES lif_document.
    METHODS: display.
ENDCLASS.
```

```
CLASS lcl_text_document IMPLEMENTATION.
  METHOD lif_document~print.
  ENDMETHOD.
  METHOD lif_document~display.
  ENDMETHOD.
  METHOD display.
  ENDMETHOD.
ENDCLASS.
```

In ABAP Objects, the same components (attributes, methods, constants, types, alias names) can be defined in an interface in largely the same way as in classes. However, interfaces do not have component visibility sections.

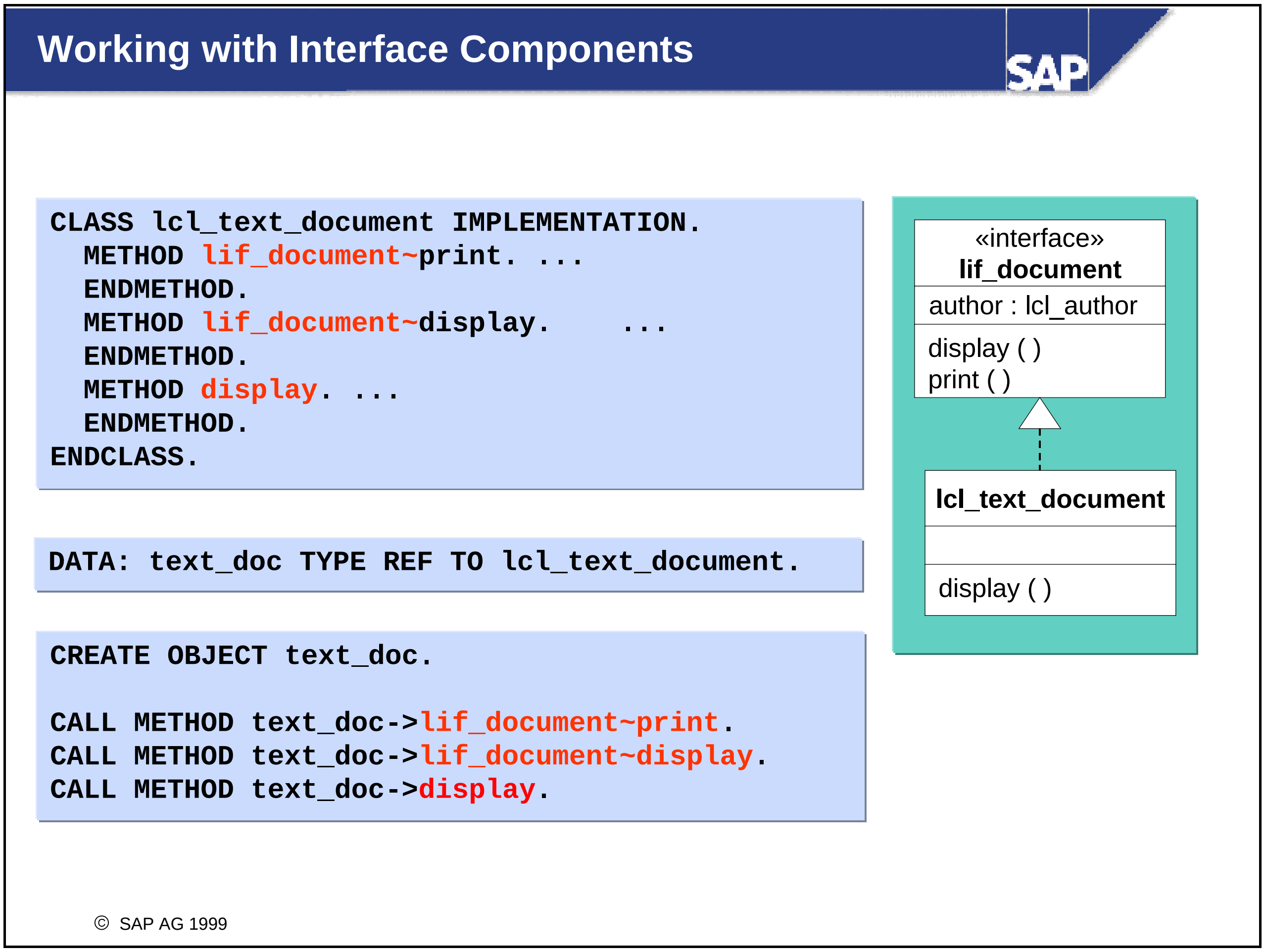
Interfaces are implemented in classes.

The interface name is listed in the definition part of the class. Interfaces can only be implemented ‘publicly’ and are therefore always in the PUBLIC SECTION (this is only valid as of Release 4.6). If you do not do this, you risk multiple implementations, if a superclass and a subclass both implement the same interface privately.

The operations defined in the interface are implemented as methods of a class. A check is carried out to ensure that all the methods defined in the interfaces are actually present in the implementation part of the class (for global interfaces, a missing or superfluous implementation of an interface method results in a ToDo warning).

The attributes, events, constants and types defined in the interface are automatically available to the class carrying out the implementation.

Interface components are addressed in the class carrying out the implementation by prefixing the interface name, followed by a tilde (the **Interface Resolution Operator**): <interfacename>~<componentname>.



Working with Interface Components

```
CLASS lcl_text_document IMPLEMENTATION.
  METHOD lif_document~print. ...
  ENDMETHOD.
  METHOD lif_document~display.    ...
  ENDMETHOD.
  METHOD display. ...
  ENDMETHOD.
ENDCLASS.
```

```
DATA: text_doc TYPE REF TO lcl_text_document.
```

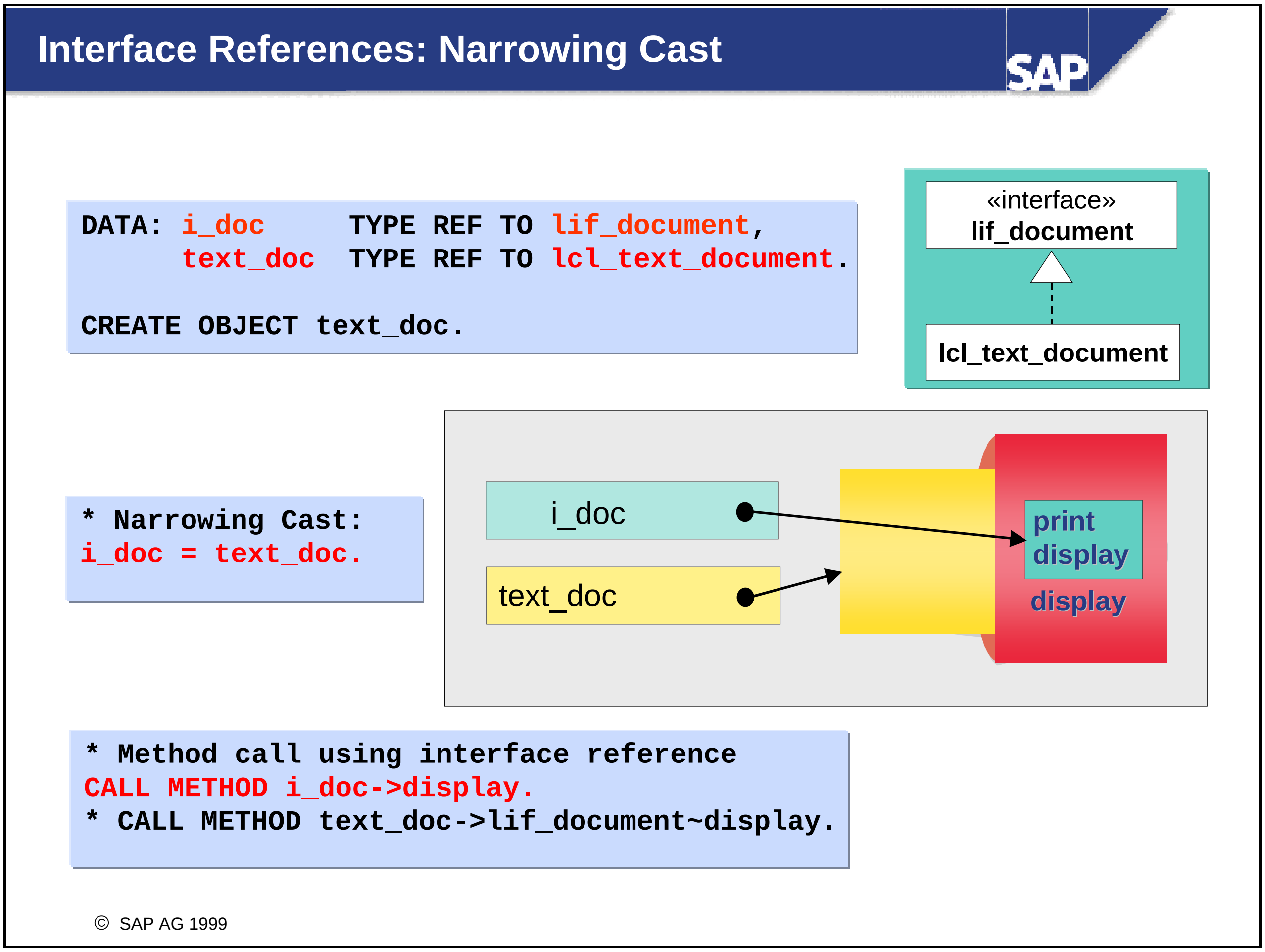
```
CREATE OBJECT text_doc.

CALL METHOD text_doc->lif_document~print.
CALL METHOD text_doc->lif_document~display.
CALL METHOD text_doc->display.
```

«interface»
lif_document
author : lcl_author
display ( )
print ( )

lcl_text_document
display ( )

The interface resolution operator enables you to access interface components using an object reference belonging to the class implementing the interface in exactly the same way as the method definition in the implementation part of the class.

This allows you to differentiate between components defined in the interface and components of the same name that are defined in the class itself. This is caused by the shared namespace.



Interface References: Narrowing Cast

```
DATA: i_doc      TYPE REF TO lif_document,
      text_doc   TYPE REF TO lcl_text_document.

CREATE OBJECT text_doc.
```

```
* Narrowing Cast:
i_doc = text_doc.
```

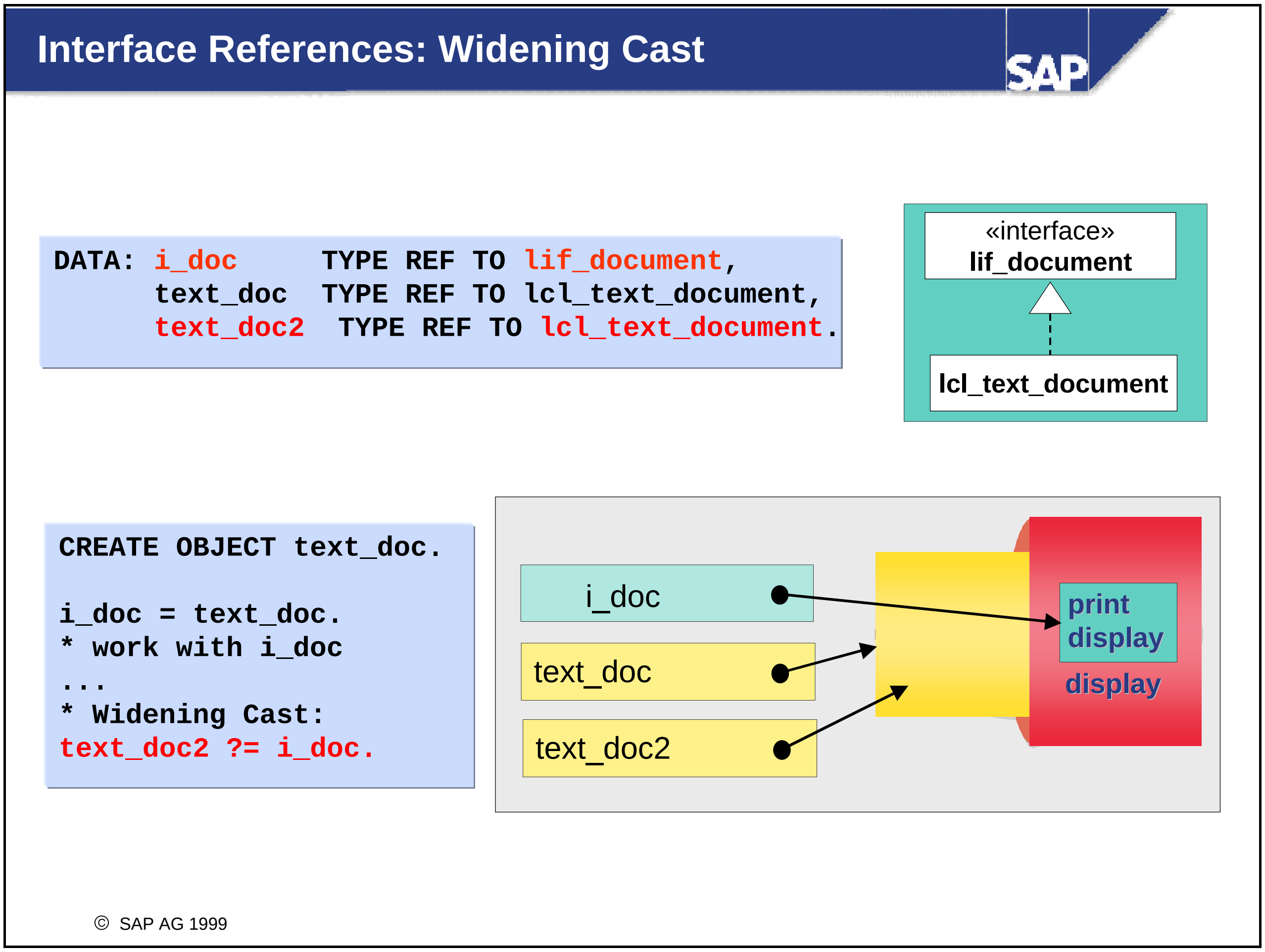
```
* Method call using interface reference
CALL METHOD i_doc->display.
* CALL METHOD text_doc->lif_document~display.
```

Interfaces are addressed using interface references. Interface references always refer to instances in the classes carrying out the implementation. Interface references therefore always have both static and dynamic types.

The assignment of an object reference to an interface reference is known as a narrowing cast since, as with inheritance, only a part of the object interface is visible once you have assigned the reference.

With an interface reference, you can no longer address all components in the class carrying out the implementation, but only the components defined in the interface. These components are now addressed using the interface reference exclusively with their own ‘short’ name!

When an object reference is assigned to an interface reference, the static types must be convertible, that is, the class that was used to define the object reference must have implemented the interface-reference interface. Otherwise there will be a syntax error.



Interface References: Widening Cast

```
DATA: i_doc      TYPE REF TO lif_document,
      text_doc   TYPE REF TO lcl_text_document,
      text_doc2  TYPE REF TO lcl_text_document.
```

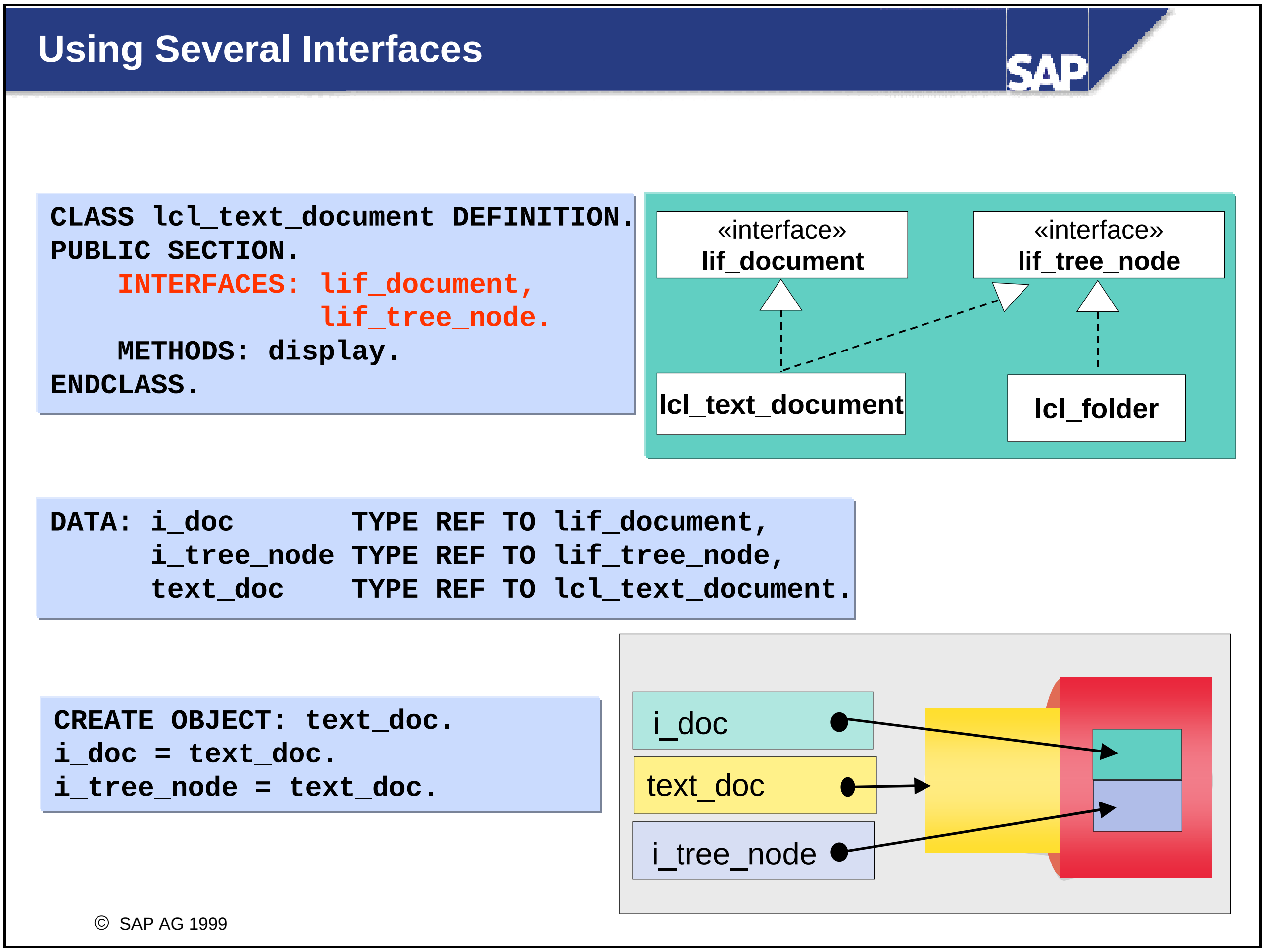
«interface»
lif_document

lcl_text_document

```
CREATE OBJECT text_doc.

i_doc = text_doc.
* work with i_doc
...
* Widening Cast:
text_doc2 ?= i_doc.
```

The widening cast is, as with inheritance, the opposite of the narrowing cast: here it is used to retrieve an object reference from an interface reference. Obviously it cannot be statically checked, since an interface can be implemented by more than one class.

An object reference cannot be assigned to an interface reference if it has itself not implemented the corresponding interface. It cannot be assigned even if a subclass has implemented the interface and the interface reference points to an object in this class.



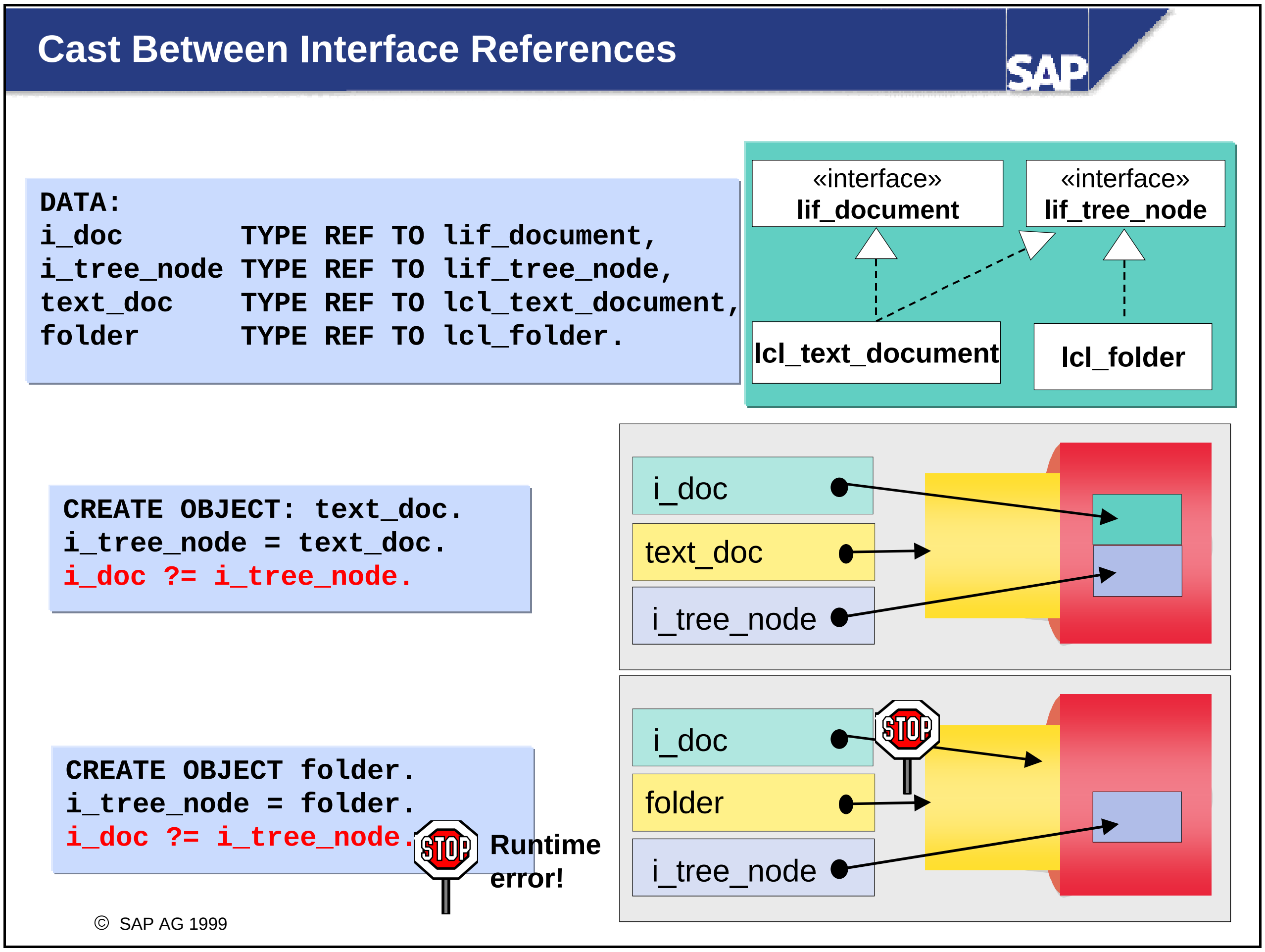
Using Several Interfaces

```
CLASS lcl_text_document DEFINITION.
PUBLIC SECTION.
    INTERFACES: lif_document,
                lif_tree_node.
    METHODS: display.
ENDCLASS.
```

```
DATA: i_doc       TYPE REF TO lif_document,
      i_tree_node TYPE REF TO lif_tree_node,
      text_doc    TYPE REF TO lcl_text_document.
```

```
CREATE OBJECT: text_doc.
i_doc = text_doc.
i_tree_node = text_doc.
```

In the above example, one class is implementing several interfaces. Even if these interfaces contain components with the same name, they are differentiated in the class carrying out the implementation by the prefix “<interfacename>~”.



Cast Between Interface References

```
DATA:
i_doc       TYPE REF TO lif_document,
i_tree_node TYPE REF TO lif_tree_node,
text_doc    TYPE REF TO lcl_text_document,
folder      TYPE REF TO lcl_folder.
```

```
CREATE OBJECT: text_doc.
i_tree_node = text_doc.
i_doc ?= i_tree_node.
```

```
CREATE OBJECT folder.
i_tree_node = folder.
i_doc ?= i_tree_node.
```

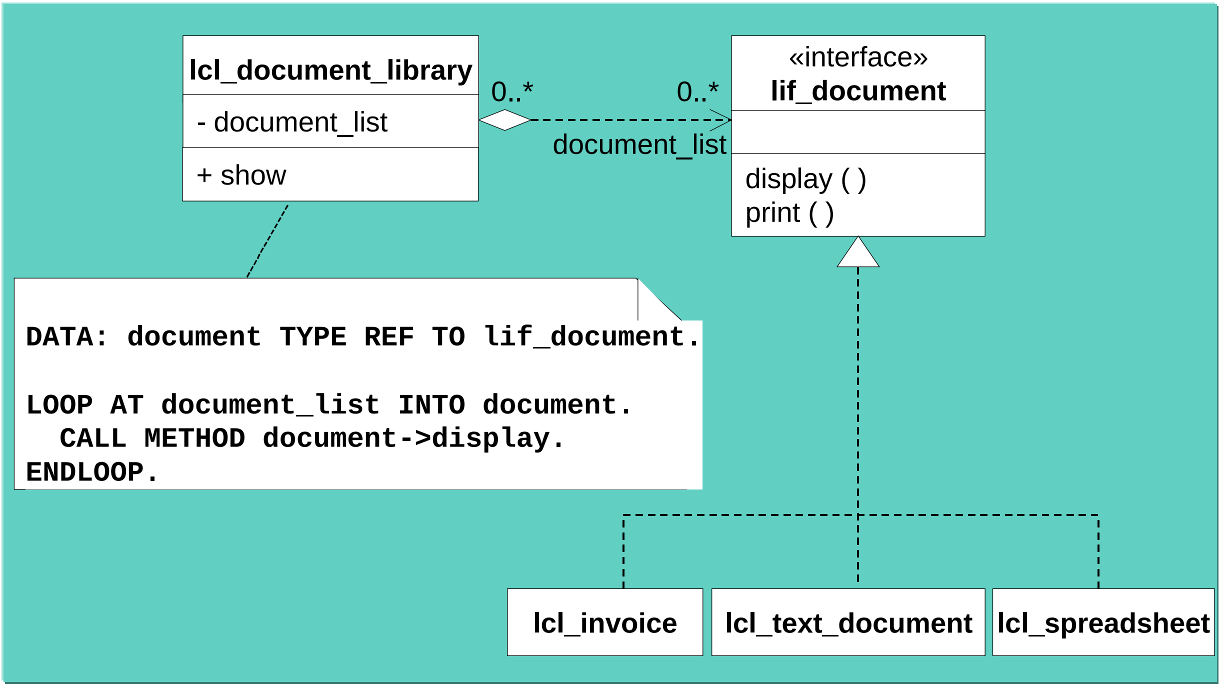
© SAP AG 1999

Assignments between interface references whose interfaces are not related to each other cannot be checked statically and must therefore be formulated using the cast operator “?=”.

For this type of assignment, a check must be carried out at runtime to see whether the class of the instance that the source reference points to also supports the interface that the target reference refers to. If this is the case, the cast is carried out, otherwise the catchable runtime MOVE_CAST_ERROR occurs.

This type of cast is neither a widening nor a narrowing cast, rather a switch from one view of an object to another.

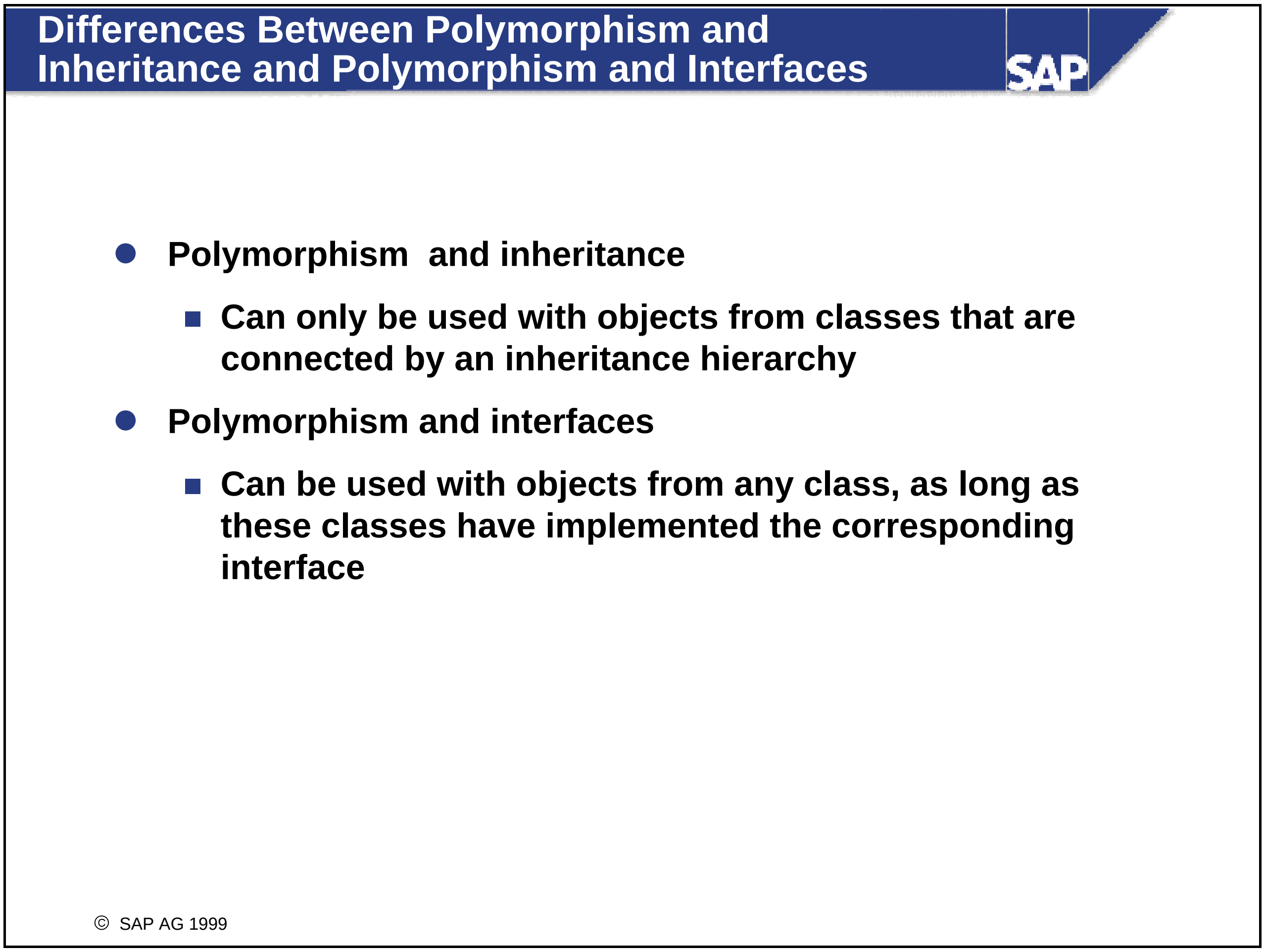
Polymorphism and Interfaces



```
DATA: document TYPE REF TO lif_document.

LOOP AT document_list INTO document.
  CALL METHOD document->display.
ENDLOOP.
```

Polymorphism can also be used for interfaces: you can use interface references to call methods that can have a different implementation depending on the object behind the reference.

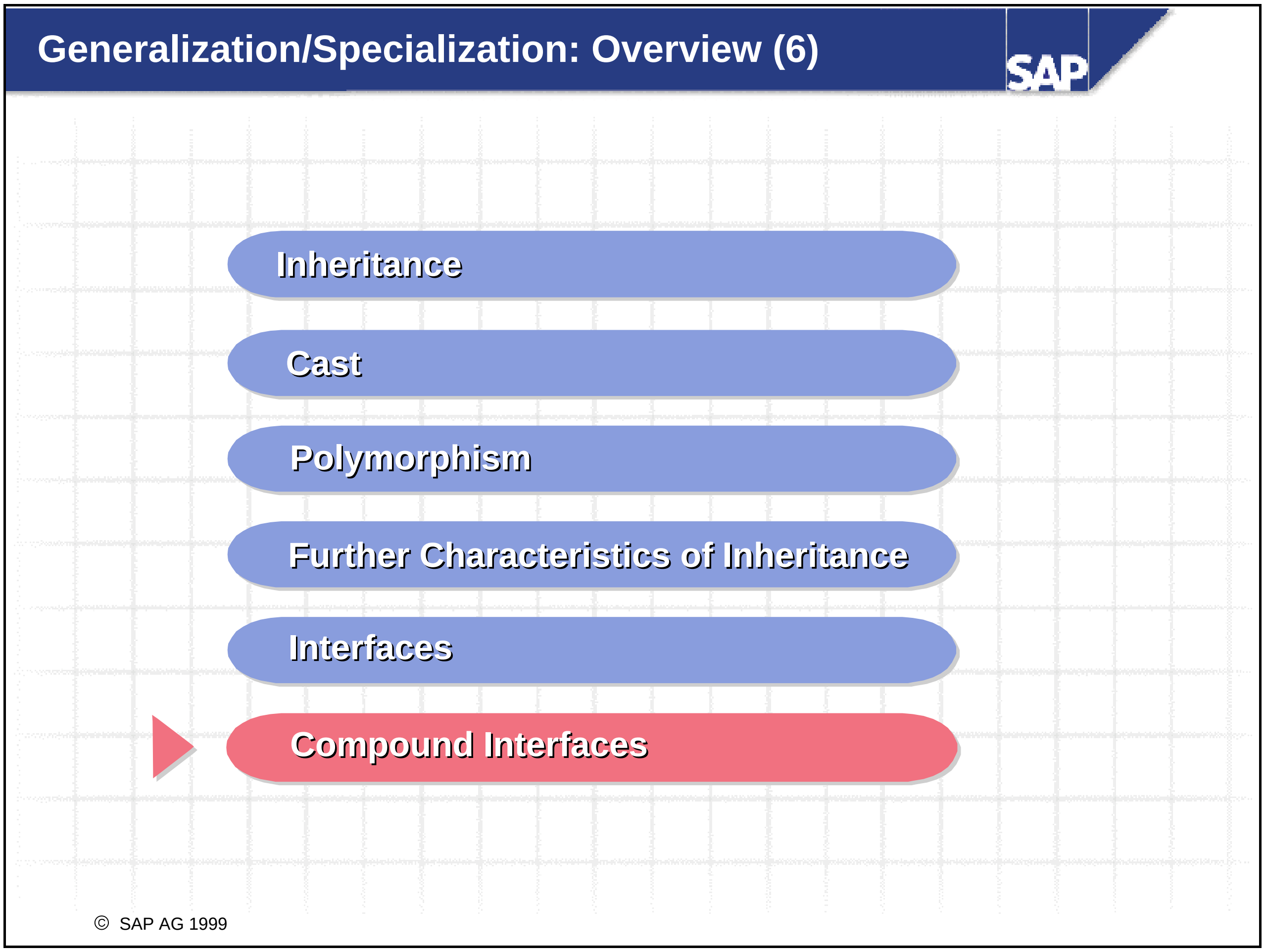
The dynamic type, not the static type of the reference variable is used to search for the implementation of a method. CALL METHOD document->display above therefore uses the class of the instance that document actually refers to to search for the implementation of display. The static type for document, which is always 'REF TO lif_doc' is not used.



Differences Between Polymorphism and Inheritance and Polymorphism and Interfaces

- Polymorphism and inheritance
  - Can only be used with objects from classes that are connected by an inheritance hierarchy
- Polymorphism and interfaces
  - Can be used with objects from any class, as long as these classes have implemented the corresponding interface

If you want to write polymorphic programs, you must first decide how the objects that you want to work with are related to each other. If the objects are dependent on each other through inheritance, then choose polymorphism and inheritance. However, if the objects are not directly related to each other, but simply ‘happen’ to have the same characteristics, then use interfaces to achieve polymorphism.



Generalization/Specialization: Overview (6)

Inheritance

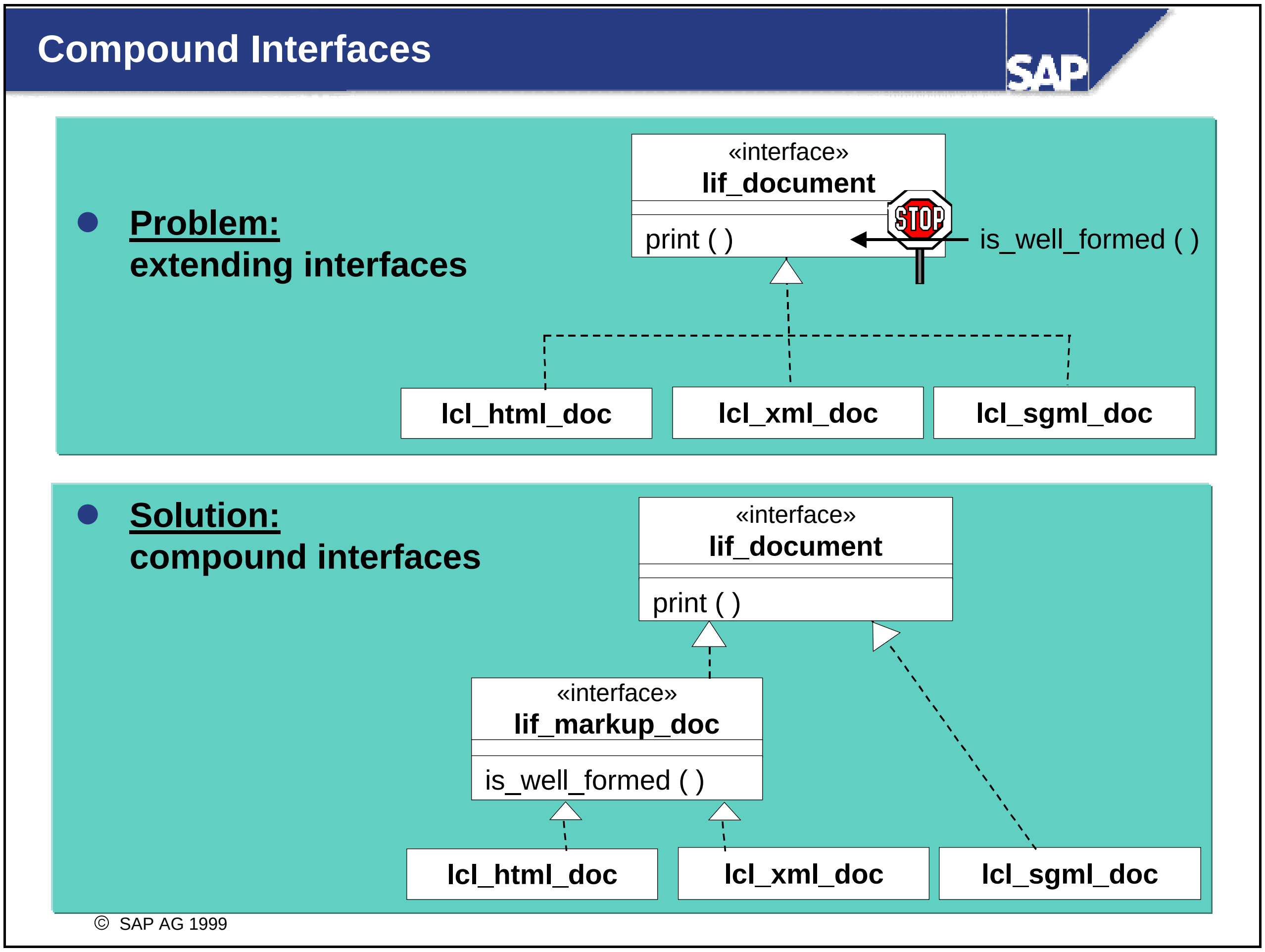
Cast

Polymorphism

Further Characteristics of Inheritance

Interfaces

▶ Compound Interfaces



Compound Interfaces

- **Problem:** **extending interfaces**

«interface»
lif_document
print ( )
is_well_formed ( )
STOP
lcl_html_doc
lcl_xml_doc
lcl_sgml_doc

- **Solution:** **compound interfaces**

«interface»
lif_document
print ( )
«interface»
lif_markup_doc
is_well_formed ( )
lcl_html_doc
lcl_xml_doc
lcl_sgml_doc

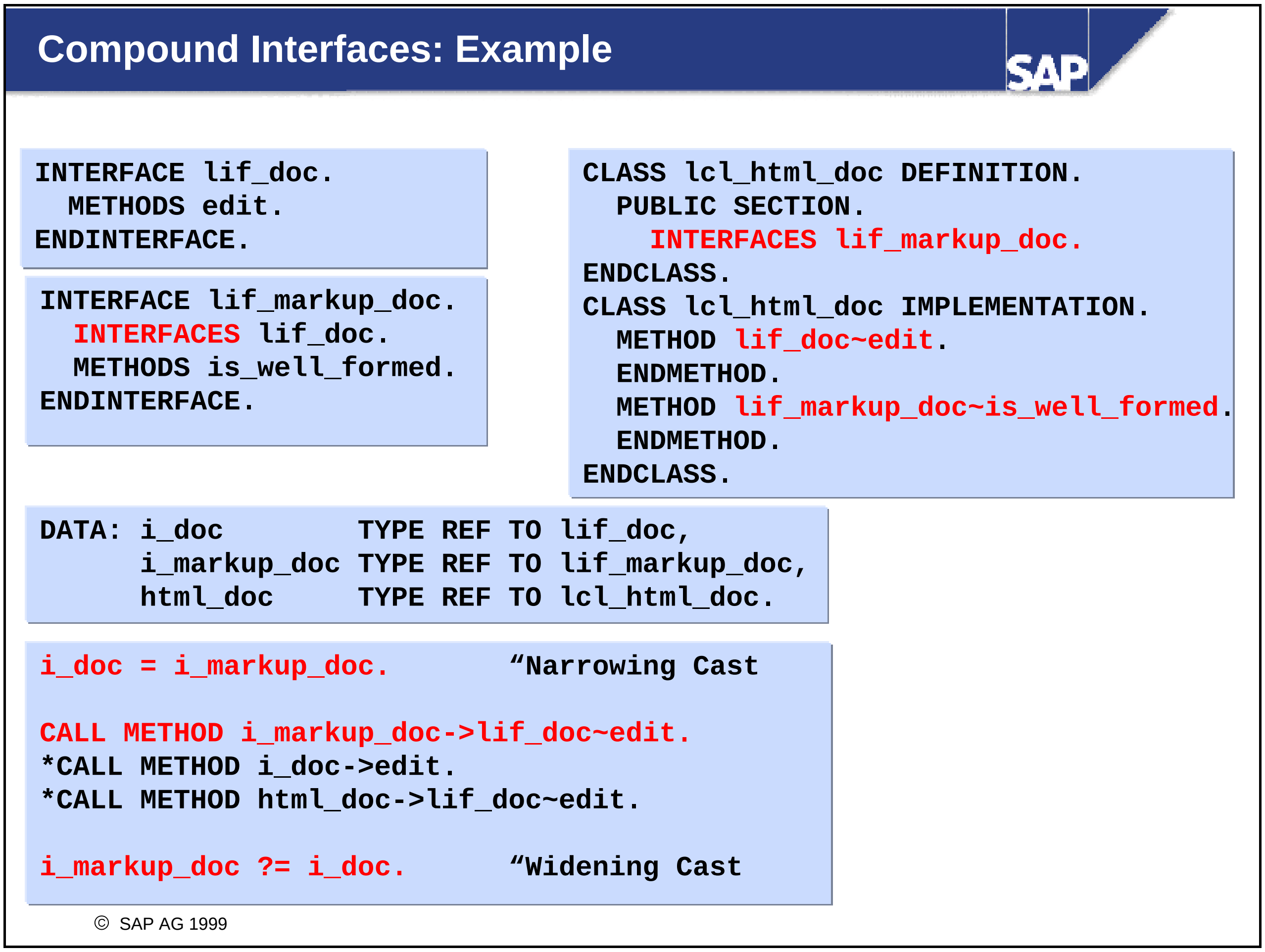
Changes to an interface usually invalidate all the classes implementing it.

ABAP Objects contains a composition model for interfaces. A compound interface contains other interfaces as components (component interfaces) and is therefore a summarized extension of these component interfaces. An elementary interface does not itself contain other interfaces.

One interface can be used as a component interface in several compound interfaces.

UML only deals with the specialization/generalization of interfaces. This relationship is represented by a dotted line with a three-sided arrow from the specialized to the generalized interface.

Compound interfaces in ABAP Objects can always be seen as specializations of their component interfaces and represented as such in UML.



Compound Interfaces: Example

```
INTERFACE lif_doc.
  METHODS edit.
ENDINTERFACE.
```

```
INTERFACE lif_markup_doc.
  INTERFACES lif_doc.
  METHODS is_well_formed.
ENDINTERFACE.
```

```
CLASS lcl_html_doc DEFINITION.
  PUBLIC SECTION.
    INTERFACES lif_markup_doc.
ENDCLASS.
CLASS lcl_html_doc IMPLEMENTATION.
  METHOD lif_doc~edit.
  ENDMETHOD.
  METHOD lif_markup_doc~is_well_formed.
  ENDMETHOD.
ENDCLASS.
```

```
DATA: i_doc        TYPE REF TO lif_doc,
      i_markup_doc TYPE REF TO lif_markup_doc,
      html_doc     TYPE REF TO lcl_html_doc.
```

```
i_doc = i_markup_doc.        “Narrowing Cast

CALL METHOD i_markup_doc->lif_doc~edit.
*CALL METHOD i_doc->edit.
*CALL METHOD html_doc->lif_doc~edit.

i_markup_doc ?= i_doc.       “Widening Cast
```

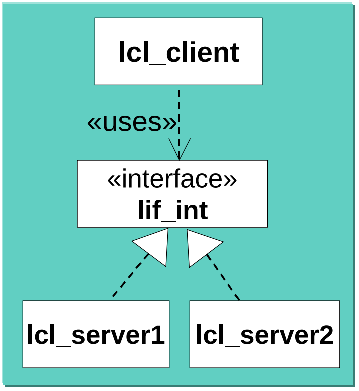
In a compound interface, the components of the component interface keep their original names, that is <component-interfacename>~<componentname>; no more prefixes are added! In other words: all components in a compound interface are on the same level, and components inherited from component interfaces are marked with the usual interface prefix.

This ‘equality principle’ for compound interfaces also affects how they are implemented. The procedure is as follows: first you implement the elementary interfaces, then the additional methods from the compound interfaces. For multiple compound interfaces, the process is simply repeated. In the class carrying out the implementation, all components of all interfaces implemented are again on the same level.

This means that interface components only ever exist once and are known by their original names <interfacename>~<componentname>. This is true both for compound interfaces and for the classes that implement them.

Using Interfaces (1)

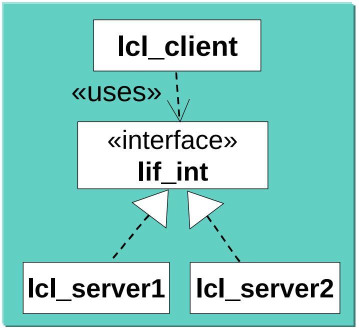
- **Separation of external point of contact (interface) and implementation (class)**
  - **The client defines the external point of contact, the server implements it**
  - **“Black Box principle”: Client only knows the interface, not the implementation**
  - **Looser linkage between client and server**



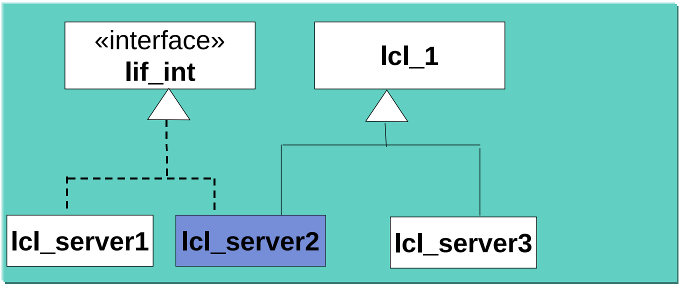
Interfaces are the means of choice for describing external points of contact, without linking them to a type of implementation. An extra layer is introduced between the client and the server to protect the client explicitly from the server, thereby making it much more independent!

Using Interfaces (2)

- **Polymorphism**
  - **Generic handling of objects from different classes**
- **Abstraction**
  - **Interface as a generalization of the class carrying out the implementation**



- **Simulation of multiple inheritance**



Interfaces enable you to work uniformly with different classes (providers). In particular, they always ensure polymorphic behavior as they do not have their own implementation, but instead allow the providers to carry it out.

The definition of an interface is always an abstraction: the user wants to handle various providers in the same way and must therefore abstract concrete implementations to a description of the services required to fulfill the task.

You can also use interfaces to achieve multiple inheritance by defining the functionality to be inherited by a second class as an interface that the inheriting class then has to implement.

Generalization/Specialization: Unit Summary

You are now able to:

- Use inheritance
- Carry out casts
- Define and implement interfaces
- Nest interfaces
- Develop generic programs using polymorphism

Generalization/Specialization Exercises

Unit: Generalization/Specialization

Topic: Inheritance

At the end of this exercise you will be able to:

- Define subclasses
- Redefine superclass methods in subclasses
-

An airline needs to manage its airplanes.

1-1 Make both instance attributes in class *lcl_airplane* (in the include program **ZBC404_##_LCL_AIRPLANE**) visible to their subclasses (PRIVATE SECTION -> PROTECTED SECTION).

1-2 Create subclass *lcl_passenger_airplane* for class *lcl_airplane*. Create the include program **ZBC404_##_LCL_PASSENGER_PLANE** for class *lcl_passenger_airplane*.

1-2-1 The class has a private instance attribute *n_o_seats*, that has the same type as table field *sflight-seatsmax*.

1-2-2 A public constructor is defined and implemented in the class. This constructor provides **all** instance attributes in the class with values.

1-2-3 Redefine method *display_attributes* of class *lcl_airplane*, so that, using the redefined method, the WRITE statement displays **all** instance attributes.

1-3 Create subclass *lcl_cargo_airplane* for class *lcl_airplane*. Create the include program **ZBC404_##_LCL_CARGO_PLANE** for class *lcl_cargo_airplane*.

1-3-1 The class has the private instance attribute *cargo_max*, that has the same type as table field *scplane-cagomax*.

1-3-2 A public constructor has been defined and implemented in the class. This constructor provides **all** instance attributes in the class with values.

1-3-3 Redefine method *display_attributes* of class *lcl_airplane*, so that, using the redefined method, the WRITE statement displays **all** instance attributes.

1-4 Create program **ZBC404_##_MAIN** (##: group number)**.**

1-4-1 Use the INCLUDE statement to include the following programs
- **ZBC404_##_LCL_AIRPLANE**
- **ZBC404_##_LCL_PASSENGER_PLANE**
- **ZBC404_##_LCL_CARGO_PLANE**.

1-4-2 Use the DATA statement to create a reference for each subclass (*lcl_passenger_airplane*, *lcl_cargo_airplane*).

1-4-3 Call the static method *display_n_o_airplanes* (before instantiating any objects).

1-4-4 Use the references 1-4-2 from to create one instance in each of the subclasses *lcl_passenger_airplane* and *lcl_cargo_airplane*. Decide for yourself how to fill the attributes.

1-4-5 Call the *display_attributes* method for each of the instances.

1-4-6 Call the static method *display_n_o_airplanes* again.

Unit: Generalization/Specialization

Topic: Polymorphism and Inheritance

At the end of this exercise you will be able to:

- Use references in internal tables
- Implement polymorphic method calls
-

An airline needs to manage its airplanes.

2-1 Copy the template **SAPBC404GENT_LCL_CARRIER** and call your new include program **ZBC404_##_ LCL_CARRIER**.

2-2 Add two public instance methods from program **ZBC404_##_ LCL_CARRIER** to the class *lcl_carrier*:

2-2-1 The first method is *add_a_new_airplane*, which adds airplanes to the *list_of_airplanes* list of airplanes already defined in the class. The transfer parameter is a reference to class *lcl_airplane*. Check the definition of the internal table *list_of_airplanes*.

2-2-2 The second method is *display_airplanes*, which displays the airplane attributes from the *list_of_airplanes* list. The *display_attributes* method from class *lcl_airplane* should be called at this point.
Question: which program part is executed for the method call *display_attributes*?

2-3 Go into program **ZBC404_##_MAIN**.

Add another INCLUDE statement including program **ZBC404_##_ LCL_CARRIER** to the existing INCLUDE statements. Make sure the include programs are in the correct sequence.

Use the DATA statement to create a reference to class *lcl_carrier*.

Comment out all the method calls in your program up till now that display data (*display_n_o_airplane*, *display_attributes*). (But only these!)

Create an airline instance using the reference from 2-3-2. Fill the transfer parameters with your own data.

Add the two planes you have created in the last exercise (one passenger and one cargo plane) to the *list_of_airplanes* list of planes. To do this, call method *add_a_new_airplane* from class *lcl_carrier*.
Create more planes and add them to the airplane list.

Display the attributes of all the planes in the airplane list by calling the *display_airplanes* method from class *lcl_carrier*.

Unit: Generalization/Specialization

Topic: Interfaces

At the end of this exercise, you will be able to:

- Define and implement interfaces
- Use polymorphism with interfaces
-

A travel agency needs to maintain its business connections to partners, such as airlines and hotels.

3-1 Create the include program **ZBC404_##_LIF_BUSI_PARTNERS**. Define the *lif_business_partners* interface in this program. The interface consists solely of the method *display_company_data*.

3-2 Go into the include program **ZBC404_##_ LCL_CARRIER**. Implement the *lif_business_partners* interface in class *lcl_carrier*.

3-2-1 Enter the interface in the definition part of the class.

3-2-2 Implement the interface's method *display_company_data*. Use the WRITE statement to display important data about an airline, such as the name and number of airplanes available for business partners. Use the *list_of_airplanes* internal table to find out the number of planes (-> statement DESCRIBE TABLE ...
LINES ...).

3-3 Copy template **SAPBC404GENT_LCL_HOTEL** and call the new include program **ZBC404_##_ LCL_HOTEL**. Implement the *lif_business_partners* interface in class *lcl_hotel*.

Enter the interface in the definition part of the class.

Implement the interface's method *display_company_data*. Use the WRITE statement to display important data on a hotel, such as the name, the town and the number of rooms available (see class attributes) for business partners.

3-4 Copy template **SAPBC404GENT_LCL_TRAVEL_AGENCY** and call the new include program **ZBC404_##_ LCL_ TRAVEL_AGENCY**. In class *lcl_travel_agency*, create a list of business partners using references from the *lif_business_partners* interface.

Create a *list_of_business_partners* internal table as a private instance attribute using references to the *lif_business_partners* interface.

Create a public instance method *add_business_partner* , which adds business partners to the *list_of_business_partners* list. The transfer parameter is a reference to the *lif_business_partners* interface .

3-4-3 Create a public instance method *display_business_partners*, that displays the most important company data of all business partners in the *list_of_business_partners* list. The *display_company_data* method from the *lif_business_partners* interface must be called.
Question: which program part is executed for the method call
display_ company_data?

3-5 Go into program **ZBC404_##_MAIN**.

Use the INCLUDE statement to include the following include programs in your program:
- ZBC404_##_LIF_BUSI_PARTNERS
- ZBC404_##_ LCL_HOTEL
- **ZBC404_##_ LCL_ TRAVEL_AGENCY**.
Make sure that the INCLUDE statement with
ZBC404_##_ LCL_ CARRIER comes before INCLUDE
ZBC404_##_ LCL_ TRAVEL_AGENCY.

Use the DATA statement to define a reference to class *lcl_travel_agency* and at least one reference to class *lcl_hotel*.

Comment out the "CALL METHOD carrier->display_airplanes." statement from the last exercise.

At the end of the program, create a travel agency and at least one hotel using the reference(s) defined in 3-5-2. Fill the transfer parameters with your own data.

Add the airline instance you created in the last exercise and the hotels you created in 3-5-4 to the *list_of_business_partners* list of the travel agency you created in 3-5-4. To do this, call method *add_business_partner*.

Call method *display_business_partners* to display a list of important company data for all the travel agency's business partners.

Generalization/Specialization Solutions

Unit: Generalization/Specialization

Topic: Inheritance

```
REPORT  sapbc404gens_inheritance     .

INCLUDE sapbc404gens_lcl_airplane.
INCLUDE sapbc404gens_lcl_passenger_air.
INCLUDE sapbc404gens_lcl_cargo_air.

DATA: passenger_airplane TYPE REF TO lcl_passenger_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane.


START-OF-SELECTION.

  CALL METHOD lcl_airplane=>display_n_o_airplanes.

  CREATE OBJECT passenger_airplane EXPORTING
  im_name      = 'LH Berlin'
                                 im_planetype = '747-400'
                                 im_n_o_seats = 580.

  CREATE OBJECT cargo_airplane EXPORTING
    im_name     = 'US Hercules'
                             im_planetype = 'Galaxy'
                             im_cargo_max = 30000.


  CALL METHOD passenger_airplane->display_attributes.

  CALL METHOD cargo_airplane->display_attributes.

  CALL METHOD lcl_airplane=>display_n_o_airplanes.
```

Include program SAPBC404GENS_LCL_AIRPLANE

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

  PUBLIC SECTION.

    TYPES: name_type(25) TYPE c.
    CONSTANTS: pos_1 TYPE i VALUE 30.

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

    CLASS-METHODS: display_n_o_airplanes.

* NEW: protected section
  PROTECTED SECTION.

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

  PRIVATE SECTION.

    CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.


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

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

  METHOD display_attributes.
    WRITE: / 'Name of the airplane: '(001), AT pos_1 name,
           / 'Plane type:           '(002), AT pos_1 planetype.
  ENDMETHOD.

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

ENDCLASS.
```

Include program SAPBC404GENS_LCL_PASSENGER_AIR

```
*---------------------------------------------------------------*
*       CLASS lcl_passenger_airplane DEFINITION                 *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane DEFINITION INHERITING FROM
  lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
   im_name      TYPE name_type
                  im_planetype TYPE saplane-planetype
                  im_n_o_seats TYPE sflight-seatsmax,
           display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: n_o_seats TYPE sflight-seatsmax.

ENDCLASS.
```

```
*---------------------------------------------------------------*
*      CLASS lcl_passenger_airplane IMPLEMENTATION                *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name      = im_name
                                 im_planetype = im_planetype.
    n_o_seats = im_n_o_seats.
  ENDMETHOD.

  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Number of seats:      '(003), 25 n_o_seats, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_CARGO_AIR

```
*---------------------------------------------------------------*
*      CLASS lcl_cargo_airplane DEFINITION                  *
*---------------------------------------------------------------*
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
   im_name      TYPE name_type
                      im_planetype TYPE saplane-planetype
                      im_cargo_max TYPE p,
            display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: cargo_max TYPE scplane-cargomax.

ENDCLASS.
```

```
*---------------------------------------------------------------*
*       CLASS lcl_cargo_airplane IMPLEMENTATION                    *
*---------------------------------------------------------------*
CLASS lcl_cargo_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name      = im_name
                                   im_planetype = im_planetype.
    cargo_max = im_cargo_max.
  ENDMETHOD.

  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Maximal cargo:      '(004),
           at pos_1 cargo_max left-justified, /.
  ENDMETHOD.

ENDCLASS.
```

Unit: Generalization/Specialization

Topic: Polymorphism and Inheritance

```
REPORT  sapbc404gens_inh_polymorphism .

INCLUDE sapbc404gens_lcl_airplane.
INCLUDE sapbc404gens_lcl_passenger_air.
INCLUDE sapbc404gens_lcl_cargo_air.
* New include
INCLUDE sapbc404gens_lcl_carrier_1.

DATA: passenger_airplane TYPE REF TO lcl_passenger_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane,
*     new reference
      carrier TYPE REF TO lcl_carrier.


START-OF-SELECTION.

*  CALL METHOD lcl_airplane=>display_n_o_airplanes.

  CREATE OBJECT passenger_airplane EXPORTING
  im_name      = 'LH Berlin'
                                  im_planetype = '747-400'
                                  im_n_o_seats = 580.

  CREATE OBJECT cargo_airplane EXPORTING
    im_name      = 'US Hercules'
                              im_planetype = 'Galaxy'
                              im_cargo_max = 30000.


*  CALL METHOD passenger_airplane->display_attributes.

*  CALL METHOD cargo_airplane->display_attributes.

*  CALL METHOD lcl_airplane=>display_n_o_airplanes.

* new coding
* Create a carrier
  CREATE OBJECT carrier EXPORTING im_name = 'Lufthansa'.
```

```
* Add passenger airplane to airplane list
 CALL METHOD carrier->add_a_new_airplane EXPORTING
                  im_airplane = passenger_airplane.

* Create new passenger airplane
 CREATE OBJECT passenger_airplane EXPORTING
 im_name      = 'LH München'
                  im_planetype = 'A310-300'
                  im_n_o_seats = 280.

* Add new passenger airplane to airplane list
 CALL METHOD carrier->add_a_new_airplane EXPORTING
                  im_airplane = passenger_airplane.


* Add cargo airplane to airplane list
 CALL METHOD carrier->add_a_new_airplane EXPORTING
                    im_airplane = cargo_airplane.

* Display all airplanes of airplane list
 CALL METHOD carrier->display_airplanes.
```

Include program SAPBC404GENS_LCL_AIRPLANE

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

 PUBLIC SECTION.

  TYPES: name_type(25) TYPE c.
  CONSTANTS: pos_1 TYPE i VALUE 30.

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

  CLASS-METHODS: display_n_o_airplanes.
```

```
* NEW: protected section
  PROTECTED SECTION.

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

  PRIVATE SECTION.

    CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.


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

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

  METHOD set_attributes.
    name      = im_name.
    planetype = im_planetype.
  ENDMETHOD.

  METHOD display_attributes.
    WRITE: / 'Name of the airplane: '(001), AT pos_1 name,
           / 'Plane type:           '(002), AT pos_1 planetype.
  ENDMETHOD.
  METHOD display_n_o_airplanes.
    WRITE: /, / 'Total number of airplanes: '(ca1),
           AT pos_1 n_o_airplanes LEFT-JUSTIFIED, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_PASSENGER_AIR

```
*---------------------------------------------------------------*
*       CLASS lcl_passenger_airplane DEFINITION                   *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane DEFINITION INHERITING FROM
  lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
     im_name      TYPE name_type
                     im_planetype TYPE saplane-planetype
                     im_n_o_seats TYPE sflight-seatsmax,
             display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: n_o_seats TYPE sflight-seatsmax.

ENDCLASS.




*---------------------------------------------------------------*
*       CLASS lcl_passenger_airplane IMPLEMENTATION               *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name      = im_name
                          im_planetype = im_planetype.
    n_o_seats = im_n_o_seats.
  ENDMETHOD.
  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Number of seats:      '(003), 25 n_o_seats, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_CARGO_AIR

```
*---------------------------------------------------------------------*
*       CLASS lcl_cargo_airplane DEFINITION                            *
*---------------------------------------------------------------------*
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
   im_name      TYPE name_type
                      im_planetype TYPE saplane-planetype
                      im_cargo_max TYPE p,
             display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: cargo_max TYPE scplane-cargomax.

ENDCLASS.
```

```
*---------------------------------------------------------------------*
*       CLASS lcl_cargo_airplane IMPLEMENTATION                        *
*---------------------------------------------------------------------*
CLASS lcl_cargo_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name      = im_name
                                im_planetype = im_planetype.
    cargo_max = im_cargo_max.
  ENDMETHOD.

  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Maximal cargo:      '(004),
           at pos_1 cargo_max left-justified, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_CARRIER_1

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

  PUBLIC SECTION.

    TYPES: BEGIN OF flight_list_type,
             connid   TYPE sflight-connid,
             fldate   TYPE sflight-fldate,
             airplane TYPE REF TO lcl_airplane,
             seatsocc TYPE sflight-seatsocc,
             cargo(5) TYPE p DECIMALS 3,
           END OF flight_list_type.

    METHODS: constructor IMPORTING im_name TYPE string,
             get_name returning value(ex_name) type string,
*            add a new airplane
             add_a_new_airplane IMPORTING
                           im_airplane TYPE REF TO lcl_airplane,
             create_a_new_flight IMPORTING
                            im_connid   TYPE sflight-connid
                            im_fldate   TYPE sflight-fldate
                            im_airplane TYPE REF TO lcl_airplane
                            im_seatsocc TYPE sflight-seatsocc
                                    OPTIONAL
                            im_cargo    TYPE p OPTIONAL,
*            display airplanes
             display_airplanes.


  PRIVATE SECTION.

    DATA: name              TYPE string,
          list_of_airplanes TYPE TABLE OF REF TO lcl_airplane,
          list_of_flights   TYPE TABLE OF flight_list_type.
```

```
ENDCLASS.


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

  METHOD constructor.
    name = im_name.
  ENDMETHOD.

  METHOD get_name.
    ex_name = name.
  ENDMETHOD.

* add a new airplane
  METHOD add_a_new_airplane.
    APPEND im_airplane TO list_of_airplanes.
  ENDMETHOD.

  METHOD create_a_new_flight.
    DATA: wa_list_of_flights TYPE flight_list_type.

    wa_list_of_flights-connid   = im_connid.
    wa_list_of_flights-fldate   = im_fldate.
    wa_list_of_flights-airplane = im_airplane.
    IF im_seatsocc IS INITIAL.
      wa_list_of_flights-cargo = im_cargo.
    ELSE.
      wa_list_of_flights-seatsocc = im_seatsocc.
    ENDIF.
    APPEND wa_list_of_flights TO list_of_flights.
  ENDMETHOD.



* display airplanes
  METHOD display_airplanes.
    DATA airplane TYPE REF TO lcl_airplane.

    LOOP AT list_of_airplanes INTO airplane.
*     Polymorphism: calling different method implementations
*                   by one call
      CALL METHOD airplane->display_attributes.
    ENDLOOP.
  ENDMETHOD.
```

```
ENDCLASS.
```

Unit: Generalization/Specialization

Topic: Interfaces

```
REPORT  sapbc404gens_interfaces .

INCLUDE sapbc404gens_lcl_airplane.
INCLUDE sapbc404gens_lcl_passenger_air.
INCLUDE sapbc404gens_lcl_cargo_air.
* new includes
INCLUDE sapbc404gens_lif_busi_partners.
INCLUDE sapbc404gens_lcl_hotel.
* lcl_carrier is implementing the interface BUSINESS_PARTNERS
INCLUDE sapbc404gens_lcl_carrier_2.
INCLUDE sapbc404gens_lcl_travel_agency.


DATA: passenger_airplane TYPE REF TO lcl_passenger_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane,
      carrier TYPE REF TO lcl_carrier,
*     New references for hotels and travel agency
      hotel1 TYPE REF TO lcl_hotel, hotel2 TYPE REF TO lcl_hotel,
      agency TYPE REF TO lcl_travel_agency.



START-OF-SELECTION.

  CREATE OBJECT passenger_airplane EXPORTING
  im_name      = 'LH Berlin'
                                   im_planetype = '747-400'
                                   im_n_o_seats = 580.

  CREATE OBJECT cargo_airplane EXPORTING
    im_name      = 'US Hercules'
                               im_planetype = 'Galaxy'
                               im_cargo_max = 30000.


  CREATE OBJECT carrier EXPORTING im_name = 'Lufthansa'.

  CALL METHOD carrier->add_a_new_airplane EXPORTING
                                   im_airplane = passenger_airplane.
```

```
 CREATE OBJECT passenger_airplane EXPORTING
 im_name      = 'LH München'
                im_planetype = 'A310-300'
                im_n_o_seats = 280.

 CALL METHOD carrier->add_a_new_airplane EXPORTING
                im_airplane = passenger_airplane.


 CALL METHOD carrier->add_a_new_airplane EXPORTING
                  im_airplane = cargo_airplane.

*  CALL METHOD carrier->display_airplanes.

* Create hotels

 CREATE OBJECT hotel1 EXPORTING im_name      = 'Budget Inn'
               im_city      = 'Washington'
               im_n_o_rooms = 112.

 CREATE OBJECT hotel2 EXPORTING im_name      = 'Ambassador'
               im_city      = 'Frankfurt'
               im_n_o_rooms = 85.

* Create travel agency
 CREATE OBJECT agency EXPORTING im_name = 'Happy Holiday'.

* Add new business partners
 CALL METHOD agency->add_business_partner
           EXPORTING im_business_partner = carrier.
*       narrowing cast: type ref to interface =
*                    type ref to LCL_CARRIR

 CALL METHOD agency->add_business_partner
           EXPORTING im_business_partner = hotel1.
*       narrowing cast: type ref to interface =
*                    type ref to LCL_HOTEL


 CALL METHOD agency->add_business_partner
           EXPORTING im_business_partner = hotel2.
*       narrowing cast: type ref to interface =
*                    type ref to LCL_HOTEL

* Display business partners: Polymorphism with interfaces
 CALL METHOD agency->display_business_partners.
```

Include program SAPBC404GENS_LCL_AIRPLANE

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

  PUBLIC SECTION.

    TYPES: name_type(25) TYPE c.
    CONSTANTS: pos_1 TYPE i VALUE 30.

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

    CLASS-METHODS: display_n_o_airplanes.

* NEW: protected section
  PROTECTED SECTION.

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

  PRIVATE SECTION.

    CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.


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

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

  METHOD set_attributes.
    name      = im_name.
    planetype = im_planetype.
  ENDMETHOD.

  METHOD display_attributes.
    WRITE: / 'Name of the airplane: '(001), AT pos_1 name,
           / 'Plane type:           '(002), AT pos_1 planetype.
  ENDMETHOD.




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

ENDCLASS.
```

Include program SAPBC404GENS_LCL_PASSENGER_AIR

```
*---------------------------------------------------------------*
*       CLASS lcl_passenger_airplane DEFINITION                 *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane DEFINITION INHERITING FROM
  lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
   im_name      TYPE name_type
                      im_planetype TYPE saplane-planetype
                      im_n_o_seats TYPE sflight-seatsmax,
             display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: n_o_seats TYPE sflight-seatsmax.

ENDCLASS.




*---------------------------------------------------------------*
*       CLASS lcl_passenger_airplane IMPLEMENTATION             *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name      = im_name
                               im_planetype = im_planetype.
    n_o_seats = im_n_o_seats.
  ENDMETHOD.

  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Number of seats:      '(003), 25 n_o_seats, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_CARGO_AIR

```
*---------------------------------------------------------------*
*       CLASS lcl_cargo_airplane DEFINITION                     *
*---------------------------------------------------------------*
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
   im_name       TYPE name_type
                       im_planetype TYPE saplane-planetype
                       im_cargo_max TYPE p,
            display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: cargo_max TYPE scplane-cargomax.

ENDCLASS.
```

```
*---------------------------------------------------------------*
*       CLASS lcl_cargo_airplane IMPLEMENTATION                 *
*---------------------------------------------------------------*
CLASS lcl_cargo_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name       = im_name
                                 im_planetype = im_planetype.
    cargo_max = im_cargo_max.
  ENDMETHOD.

  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Maximal cargo:      '(004),
           at pos_1 cargo_max left-justified, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LIF_BUSI_PARTNERS

```
INTERFACE lif_business_partners.
```

```
  METHODS: display_company_data.

ENDINTERFACE.
```

Include program SAPBC404GENS_LCL_HOTEL

```
*---------------------------------------------------------------*
*       CLASS lcl_hotel DEFINITION                              *
*---------------------------------------------------------------*
CLASS lcl_hotel DEFINITION.

  PUBLIC SECTION.

*   Interface declaration
    INTERFACES: lif_business_partners.

    METHODS: constructor IMPORTING im_name      TYPE string
                                   im_city      TYPE string
                                   im_n_o_rooms TYPE i.

  PRIVATE SECTION.

    DATA: name      TYPE string,
          city      TYPE string,
          n_o_rooms TYPE i.
ENDCLASS.


*---------------------------------------------------------------*
*       CLASS lcl_hotel IMPLEMENTATION                          *
*---------------------------------------------------------------*
CLASS lcl_hotel IMPLEMENTATION.

  METHOD constructor.
    name = im_name.
    city = im_city.
    n_o_rooms = im_n_o_rooms.
  ENDMETHOD.

* Interface Implementation
  METHOD lif_business_partners~display_company_data.
    WRITE: / 'Hotel '(h01), name COLOR COL_NEGATIVE, 'in'(h02), city,
           / 'The number of available rooms is'(h03),
             n_o_rooms left-justified, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_CARRIER_2

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

  PUBLIC SECTION.

    TYPES: BEGIN OF flight_list_type,
             connid   TYPE sflight-connid,
             fldate   TYPE sflight-fldate,
             airplane TYPE REF TO lcl_airplane,
             seatsocc TYPE sflight-seatsocc,
             cargo(5) TYPE p DECIMALS 3,
           END OF flight_list_type.

*   Interface declaration
    INTERFACES: lif_business_partners.

    METHODS: constructor IMPORTING im_name TYPE string,
             get_name RETURNING value(ex_name) TYPE string,
             add_a_new_airplane IMPORTING
                        im_airplane TYPE REF TO lcl_airplane,
             create_a_new_flight IMPORTING
                        im_connid   TYPE sflight-connid
                        im_fldate   TYPE sflight-fldate
                        im_airplane TYPE REF TO lcl_airplane
                        im_seatsocc TYPE sflight-seatsocc
                                    OPTIONAL
                        im_cargo    TYPE p OPTIONAL,
             display_airplanes.


  PRIVATE SECTION.

    DATA: name              TYPE string,
          list_of_airplanes TYPE TABLE OF REF TO lcl_airplane,
          list_of_flights   TYPE TABLE OF flight_list_type.

ENDCLASS.
```

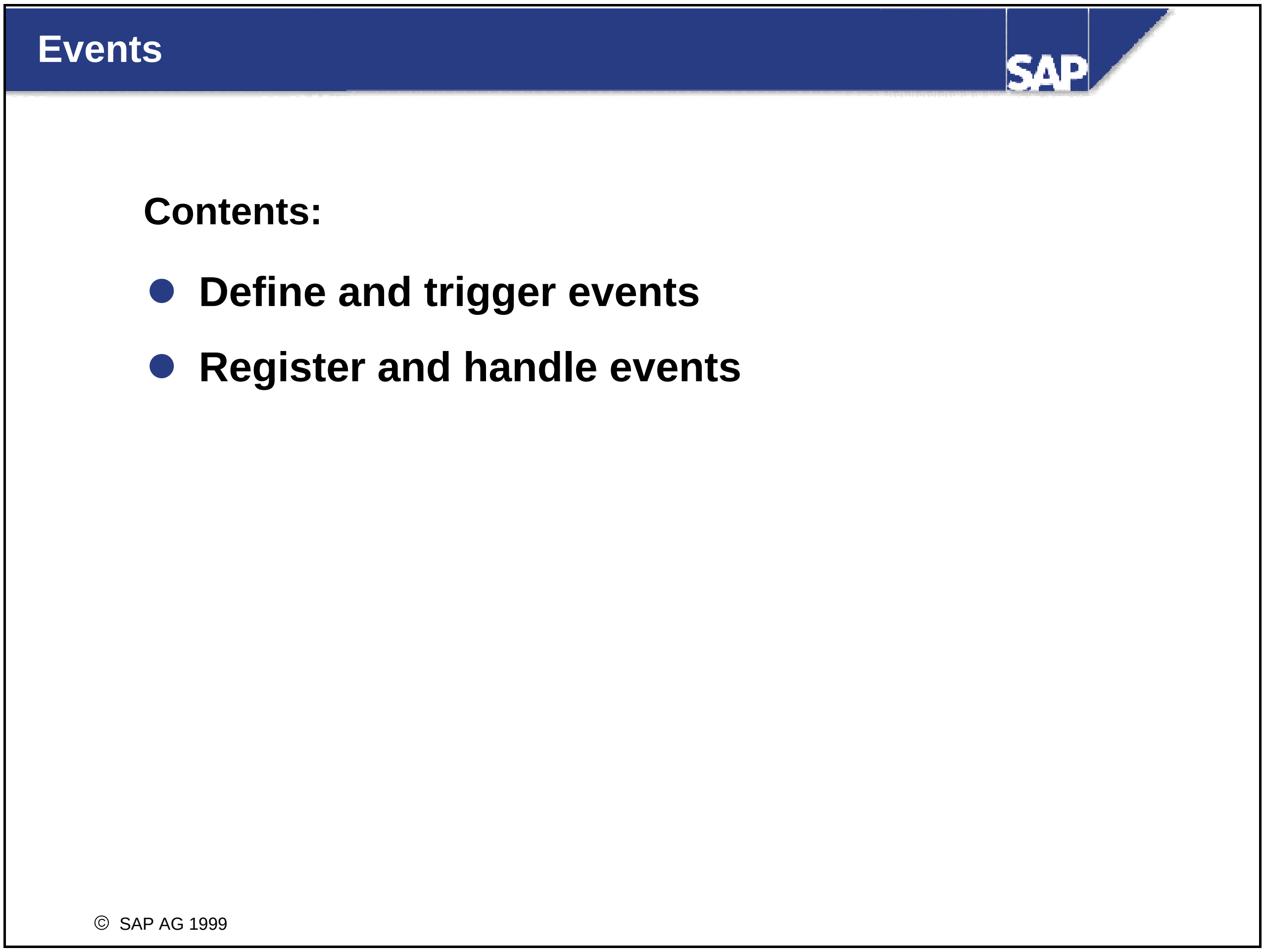
```
*---------------------------------------------------------------*
*       CLASS lcl_carrier IMPLEMENTATION                        *
*---------------------------------------------------------------*
CLASS lcl_carrier IMPLEMENTATION.

  METHOD constructor.
    name = im_name.
  ENDMETHOD.

  METHOD get_name.
    ex_name = name.
  ENDMETHOD.

* Interface Implementation
  METHOD lif_business_partners~display_company_data.
    DATA: n_o_airplanes TYPE i.
    DESCRIBE TABLE list_of_airplanes LINES n_o_airplanes.
    WRITE: / name COLOR COL_POSITIVE, / 'Number of Airplanes:'(c01),
             n_o_airplanes LEFT-JUSTIFIED, /.
  ENDMETHOD.

  METHOD add_a_new_airplane.
    APPEND im_airplane TO list_of_airplanes.
  ENDMETHOD.

  METHOD create_a_new_flight.
    DATA: wa_list_of_flights TYPE flight_list_type.

    wa_list_of_flights-connid   = im_connid.
    wa_list_of_flights-fldate   = im_fldate.
    wa_list_of_flights-airplane = im_airplane.


    IF im_seatsocc IS INITIAL.
      wa_list_of_flights-cargo = im_cargo.
    ELSE.
      wa_list_of_flights-seatsocc = im_seatsocc.
    ENDIF.
    APPEND wa_list_of_flights TO list_of_flights.
  ENDMETHOD.

  METHOD display_airplanes.
    DATA airplane TYPE REF TO lcl_airplane.

    LOOP AT list_of_airplanes INTO airplane.
```

```
   CALL METHOD airplane->display_attributes.
  ENDLOOP.
 ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_TRAVEL_AGENCY

```
*---------------------------------------------------------------*
*      CLASS lcl_travel_agency DEFINITION                       *
*---------------------------------------------------------------*
CLASS lcl_travel_agency DEFINITION.

 PUBLIC SECTION.

  METHODS: constructor IMPORTING im_name TYPE string,
*          Add new business partner
       add_business_partner IMPORTING im_business_partner
                  TYPE REF TO lif_business_partners,
*          Display business partners
       display_business_partners.

 PRIVATE SECTION.

  DATA: name TYPE string,
*   List of business partners
  list_of_business_partners TYPE TABLE OF REF TO
                       lif_business_partners.

ENDCLASS.


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

 METHOD constructor.
  name = im_name.
 ENDMETHOD.

* Add new business partner
 METHOD add_business_partner.
  APPEND im_business_partner TO list_of_business_partners.
 ENDMETHOD.
```

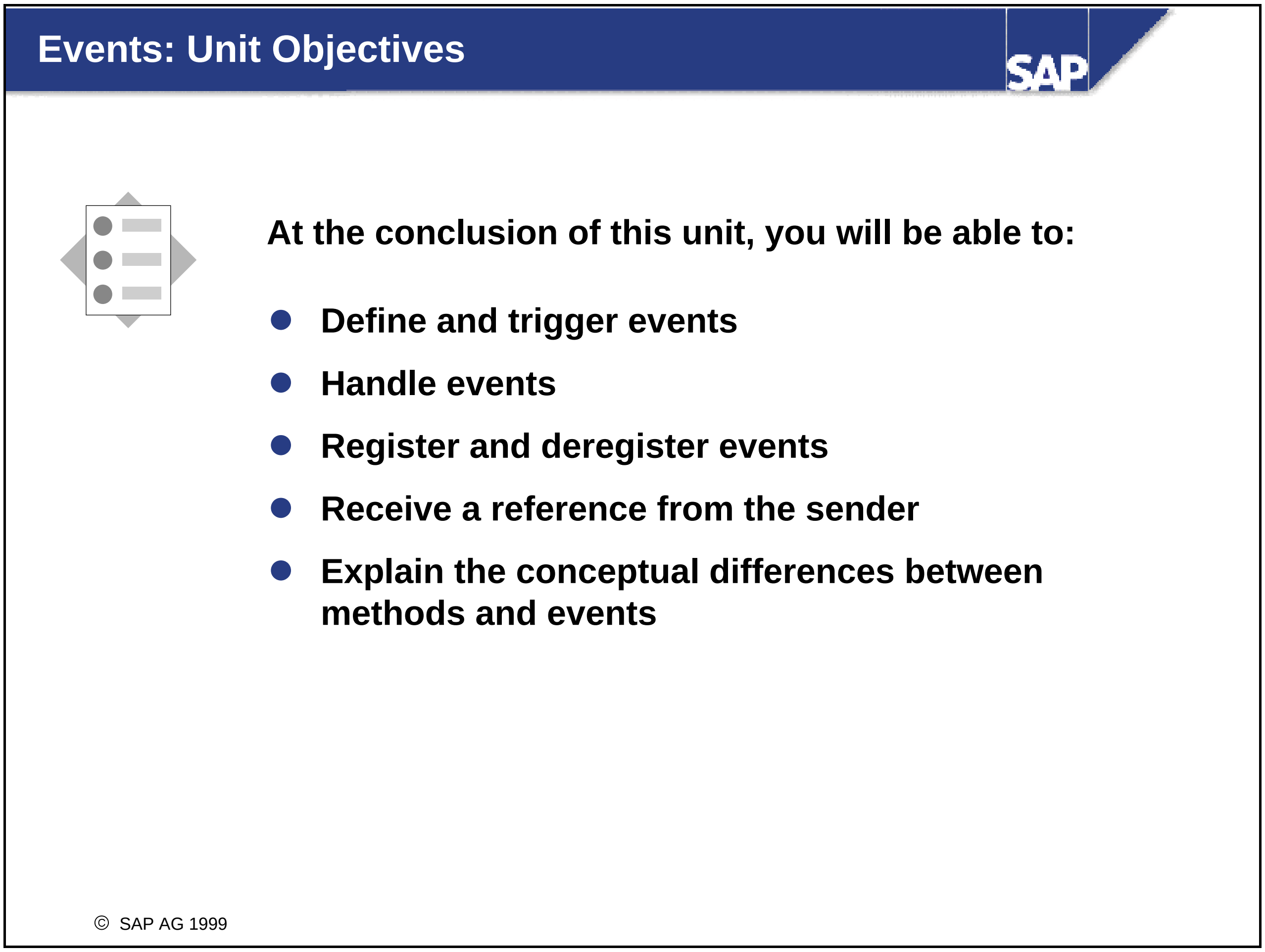
```
* Display business partners
  METHOD display_business_partners.
    DATA business_partner TYPE REF TO lif_business_partners.
    write: / 'Travel Agency:'(ta1), name color col_heading,
           / 'Business partners:'(ta2), /.
    LOOP AT list_of_business_partners INTO business_partner.
      CALL METHOD business_partner->display_company_data.
    ENDLOOP.
  ENDMETHOD.

ENDCLASS.
```



Events

Contents:

- Define and trigger events
- Register and handle events

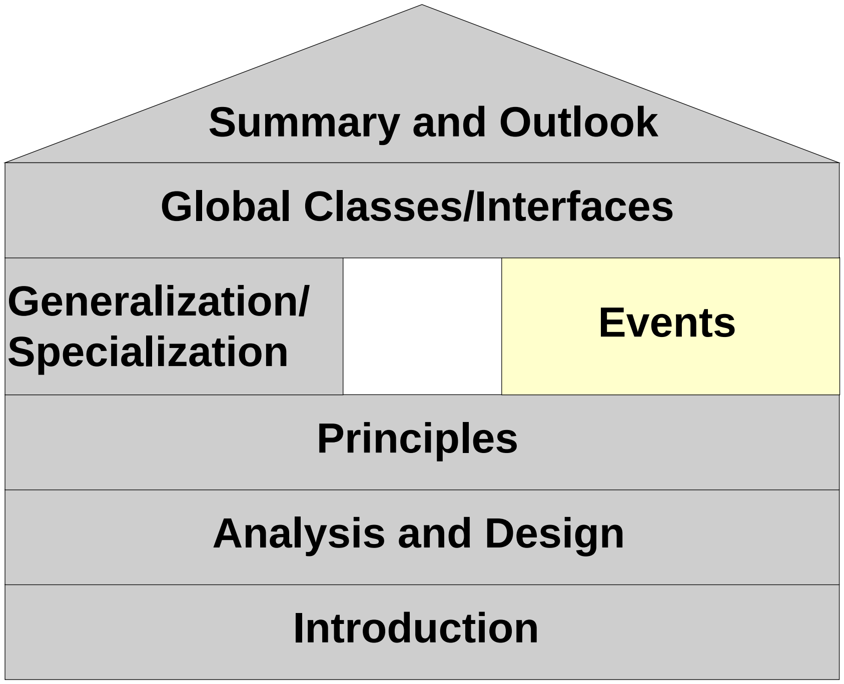


Events: Unit Objectives

At the conclusion of this unit, you will be able to:

- Define and trigger events
- Handle events
- Register and deregister events
- Receive a reference from the sender
- Explain the conceptual differences between methods and events

Events: Course Overview Diagram



Summary and Outlook

Global Classes/Interfaces

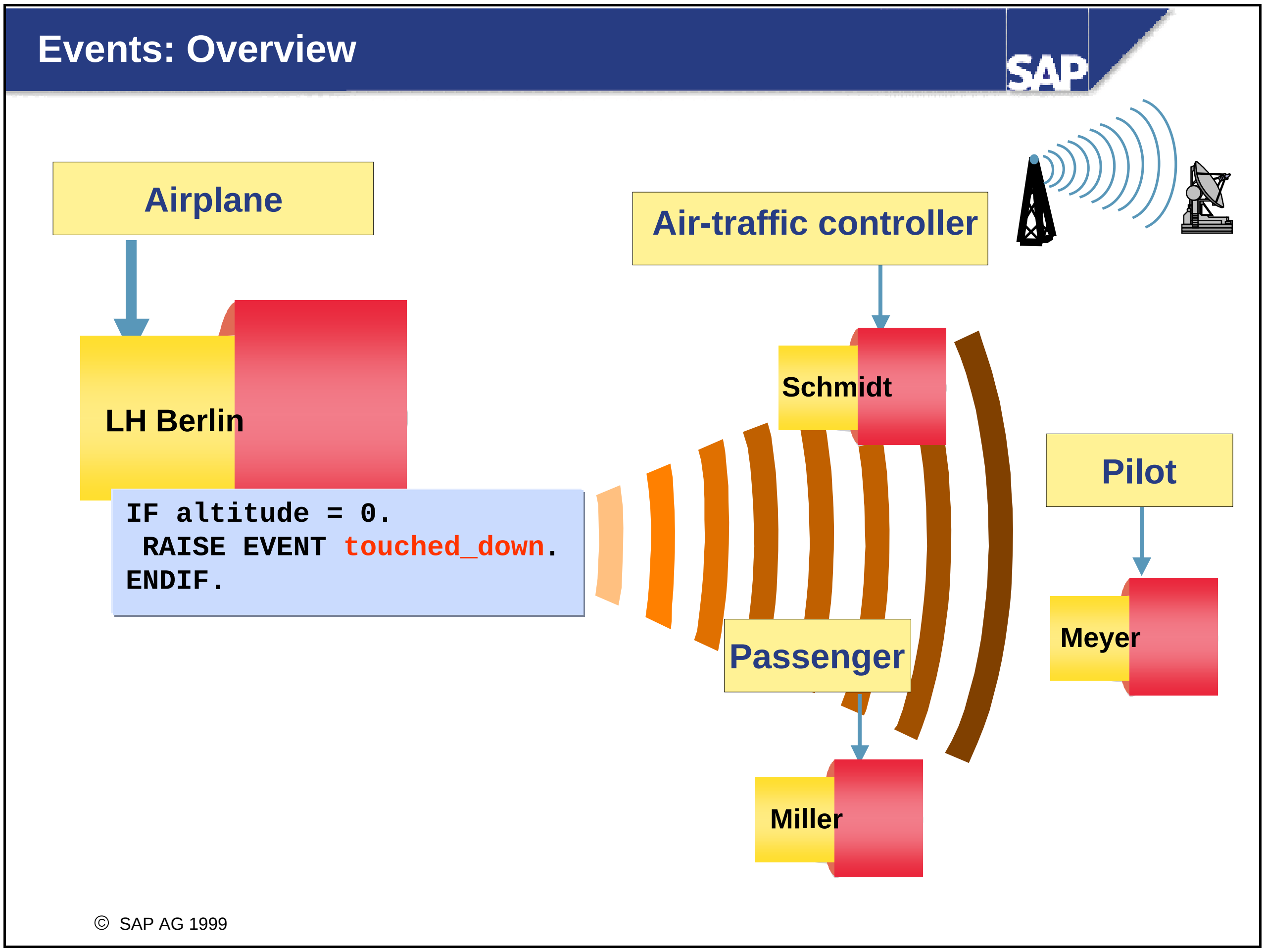
Generalization/ Specialization

Events

Principles

Analysis and Design

Introduction



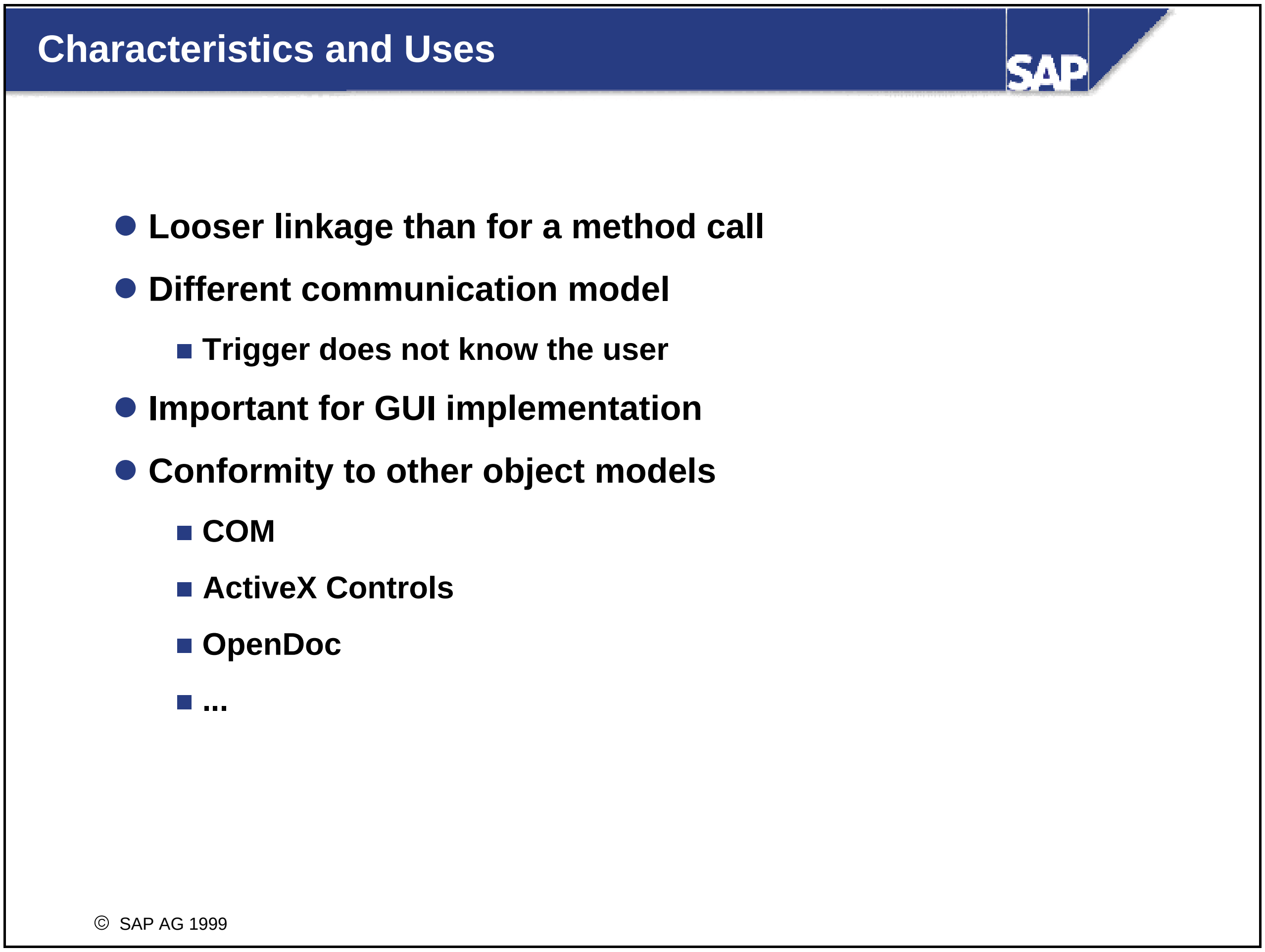
Events: Overview

```
IF altitude = 0.
 RAISE EVENT touched_down.
ENDIF.
```

By triggering an event, an object or a class announces a change of state, or that a certain state has been achieved.

In the above example, the airplane class triggers the event ‘touched_down’. Other classes subscribe to this event and process it. The air-traffic controller marks the plane as landed on the list, the pilot breathes a sigh of relief and the passenger, Mr. Miller, applauds.

Note:
The events discussed here are **not** ABAP events such as INITIALIZATION, START-OF-SELECTION, and so on.



Characteristics and Uses

- Looser linkage than for a method call
- Different communication model
  - Trigger does not know the user
- Important for GUI implementation
- Conformity to other object models
  - COM
  - ActiveX Controls
  - OpenDoc
  - ...

Events link objects or classes more loosely than direct method calls do. Method calls establish precisely when and in which statement sequence the method is called. However, with events, the reaction of the object to the event is determined by the triggering of the event itself.

Events are most often used in GUI implementations.

Other external object models, such as COM, ActiveX Controls etc, also provide events.

Triggering and Handling Events: Overview

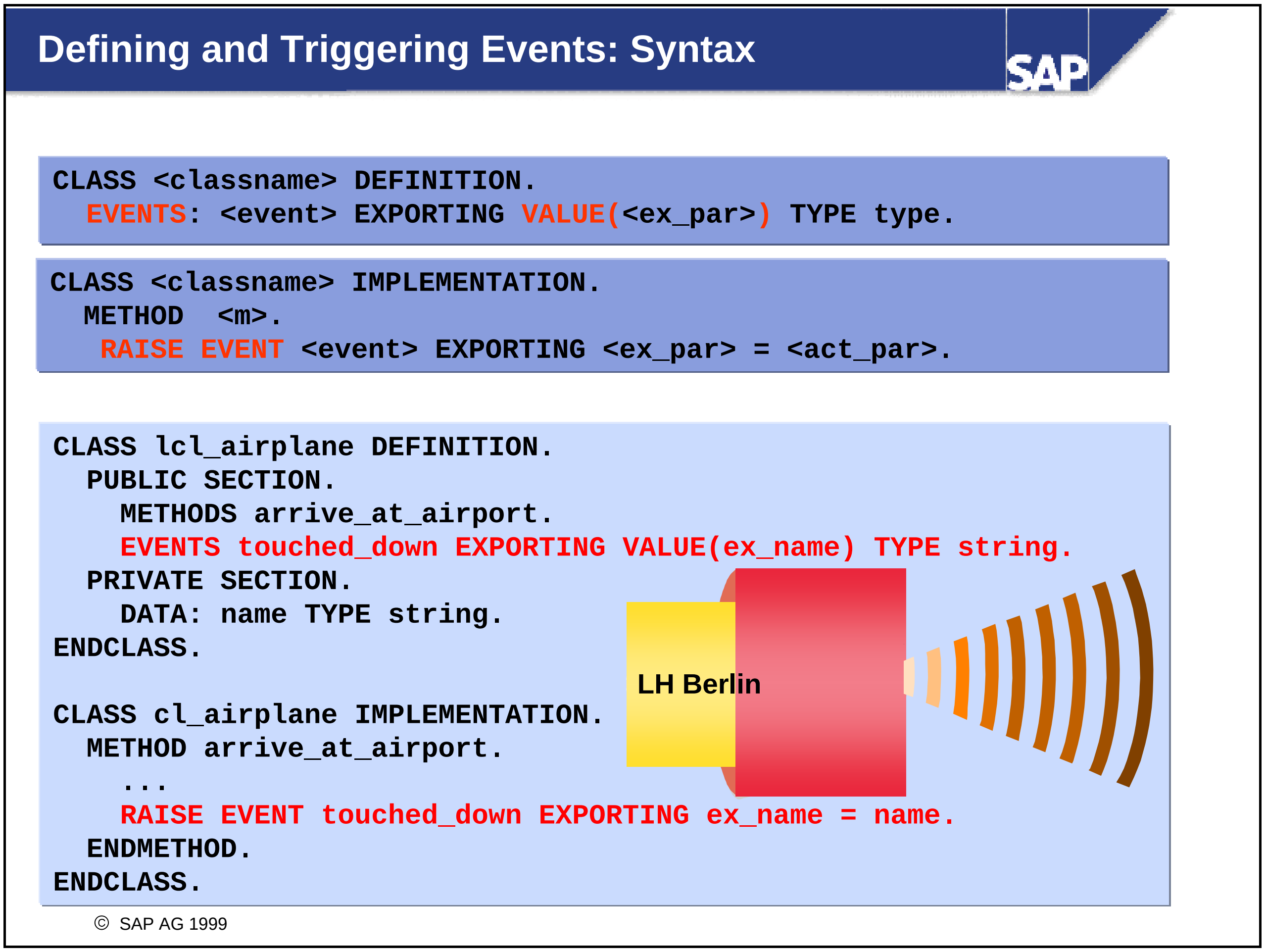
- **Triggering events**
  - **Class defines event**
    (`EVENTS, CLASS-EVENTS`)
  - **Object or class triggers event**
    (`RAISE EVENT`)
- **Handling events**
  - **Event handler class defines and implements event handler method**
    (`[CLASS-]METHODS... FOR EVENT ... OF ...`)
  - **"Event handler object" or handler class registers itself to specific object/class events at runtime**
    (`SET HANDLER`)

At the moment of implementation, a class defines its
instance events (using the statement EVENTS) and
static events (using the statement CLASS-EVENTS)

Classes or their instances that receive a message when an event is triggered at runtime and want to react to this event define event handler methods.
Statement : (CLASS-)METHODS <handler_method> FOR EVENT <event> OF <classname>.

These classes or their instances register themselves at runtime to one or more events.
Statement : SET HANDLER <handler_method> FOR <reference>. (for instance events)
SET HANDLER <handler_method>. (for static events).

A class or an instance can trigger an event at runtime using the statement RAISE EVENT.



Defining and Triggering Events: Syntax

```
CLASS <classname> DEFINITION.
  EVENTS: <event> EXPORTING VALUE(<ex_par>) TYPE type.
```

```
CLASS <classname> IMPLEMENTATION.
  METHOD  <m>.
   RAISE EVENT <event> EXPORTING <ex_par> = <act_par>.
```

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS arrive_at_airport.
    EVENTS touched_down EXPORTING VALUE(ex_name) TYPE string.
  PRIVATE SECTION.
    DATA: name TYPE string.
ENDCLASS.

CLASS cl_airplane IMPLEMENTATION.
  METHOD arrive_at_airport.
    ...
    RAISE EVENT touched_down EXPORTING ex_name = name.
  ENDMETHOD.
ENDCLASS.
```

Both instance and static events can be triggered in instance methods.

Only static events can be triggered in static methods.

Events can only have EXPORTING parameters which must be passed by value.

Triggering an event using the statement RAISE EVENT has the following effect:

the program flow is interrupted at that point

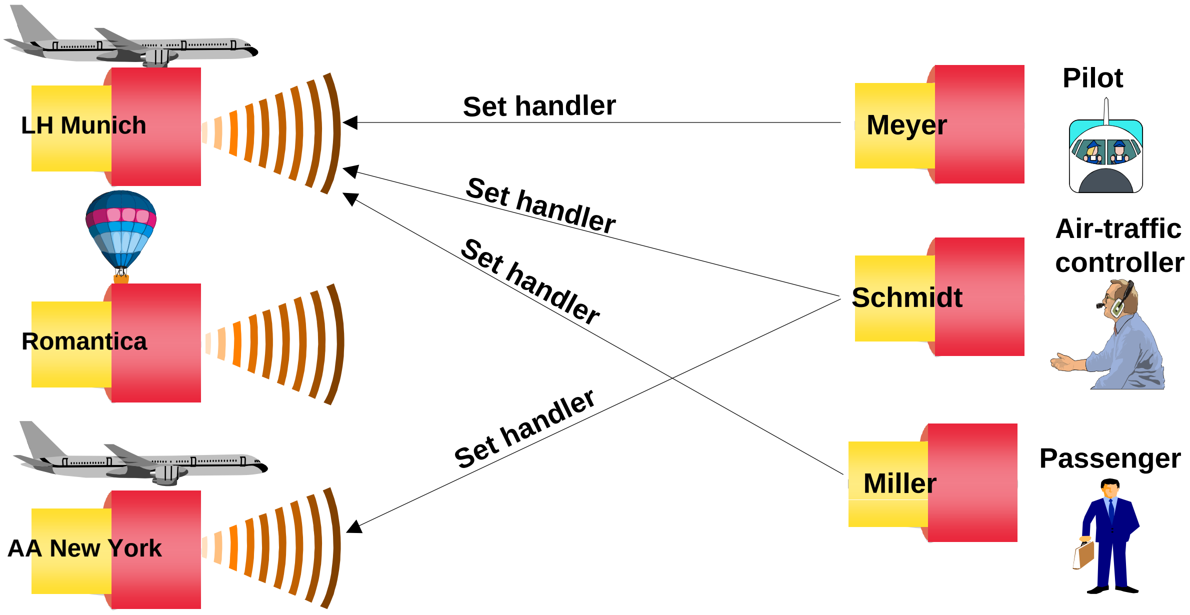
the event handler methods registered to this event are called and processed

once all event handler methods have been executed, the program flow starts again.

If an event handler method in turn triggers an event, then the program flow is again interrupted and all event handler methods are executed (nesting).

Handling and Registering Events

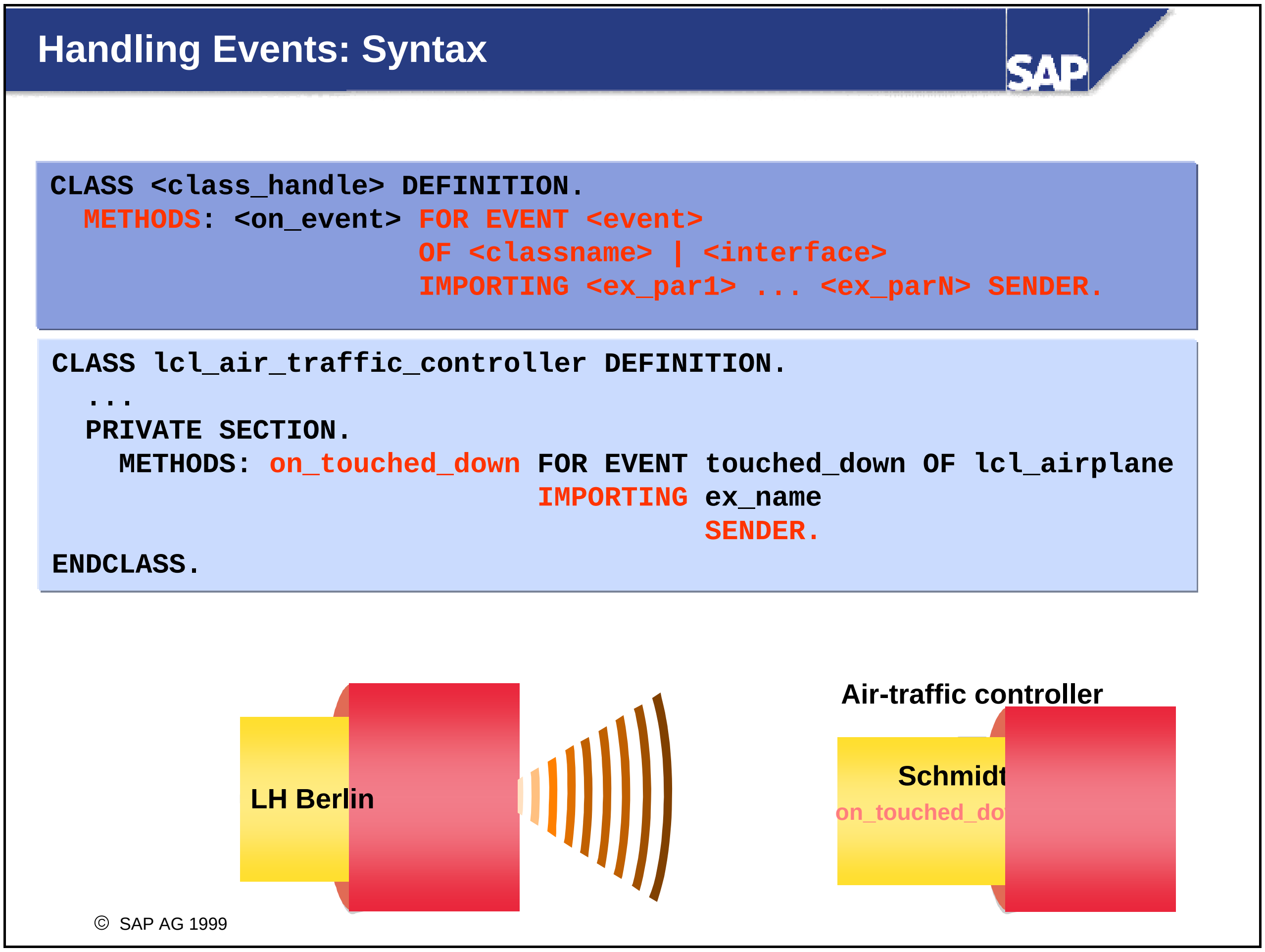
```
METHODS on_touched_down FOR EVENT touched_down OF cl_airplane.
```



Events are registered using the command SET HANDLER. Registration is only active at program runtime. Events cannot be persistent.

You want to register an object to an event belonging to another object. The SET HANDLER... statement enters the registration in that object's list. All handlers for one event are entered in this list.

When the event is triggered, the list shows which event handler methods need to be called.



Handling Events: Syntax

```
CLASS <class_handle> DEFINITION.
  METHODS: <on_event> FOR EVENT <event>
                      OF <classname> | <interface>
                      IMPORTING <ex_par1> ... <ex_parN> SENDER.
```

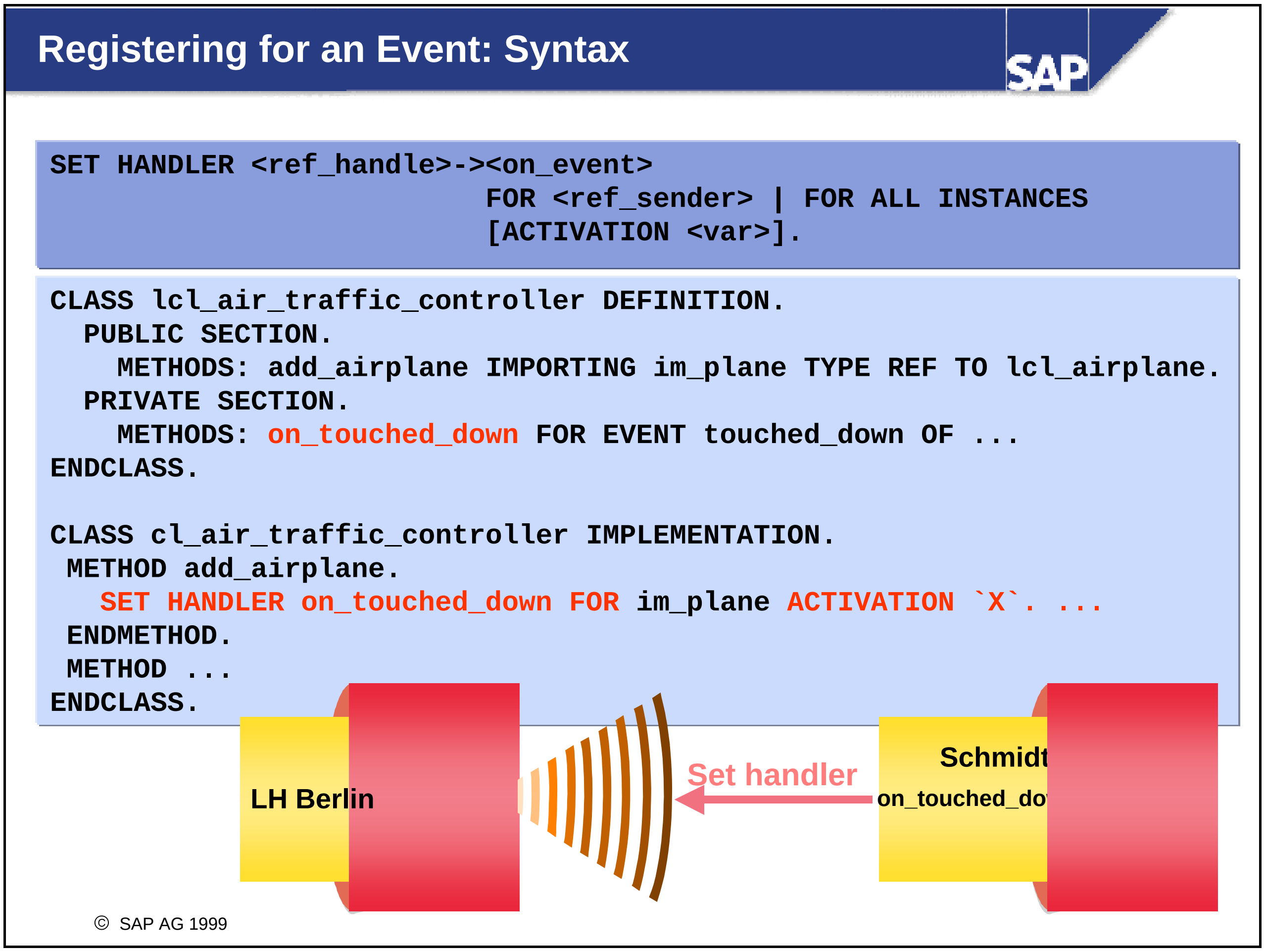
```
CLASS lcl_air_traffic_controller DEFINITION.
  ...
  PRIVATE SECTION.
    METHODS: on_touched_down FOR EVENT touched_down OF lcl_airplane
                             IMPORTING ex_name
                                       SENDER.
ENDCLASS.
```

LH Berlin
Air-traffic controller
Schmidt
on_touched_do

Event handler methods are triggered by events (RAISE EVENT), although they can also be called like normal methods (CALL METHOD).

The interface of the event handler method consists solely of IMPORTING parameters. Only parameters from the definition of the corresponding event (event interface) can be used. An event interface only has EXPORTING parameters and is defined using the EVENTS statement in the declaration of the event. The parameters are typed in the event definition and the typing is passed to the event handler method, that is, the interface parameters of the event handler method cannot be typed in the definition of the event handler method.

In addition to the explicitly defined event interface parameters, the implicit parameter SENDER can also be listed as an IMPORTING parameter for instance events. This passes on a reference to the object that triggered the event.



Registering for an Event: Syntax

```
SET HANDLER <ref_handle>-><on_event>
                          FOR <ref_sender> | FOR ALL INSTANCES
                          [ACTIVATION <var>].
```

```
CLASS lcl_air_traffic_controller DEFINITION.
  PUBLIC SECTION.
    METHODS: add_airplane IMPORTING im_plane TYPE REF TO lcl_airplane.
  PRIVATE SECTION.
    METHODS: on_touched_down FOR EVENT touched_down OF ...
ENDCLASS.

CLASS cl_air_traffic_controller IMPLEMENTATION.
 METHOD add_airplane.
   SET HANDLER on_touched_down FOR im_plane ACTIVATION `X`. ...
 ENDMETHOD.
 METHOD ...
ENDCLASS.
```

When an event is triggered, only those event handler methods that have registered themselves using SET HANDLER by this point at runtime are executed.

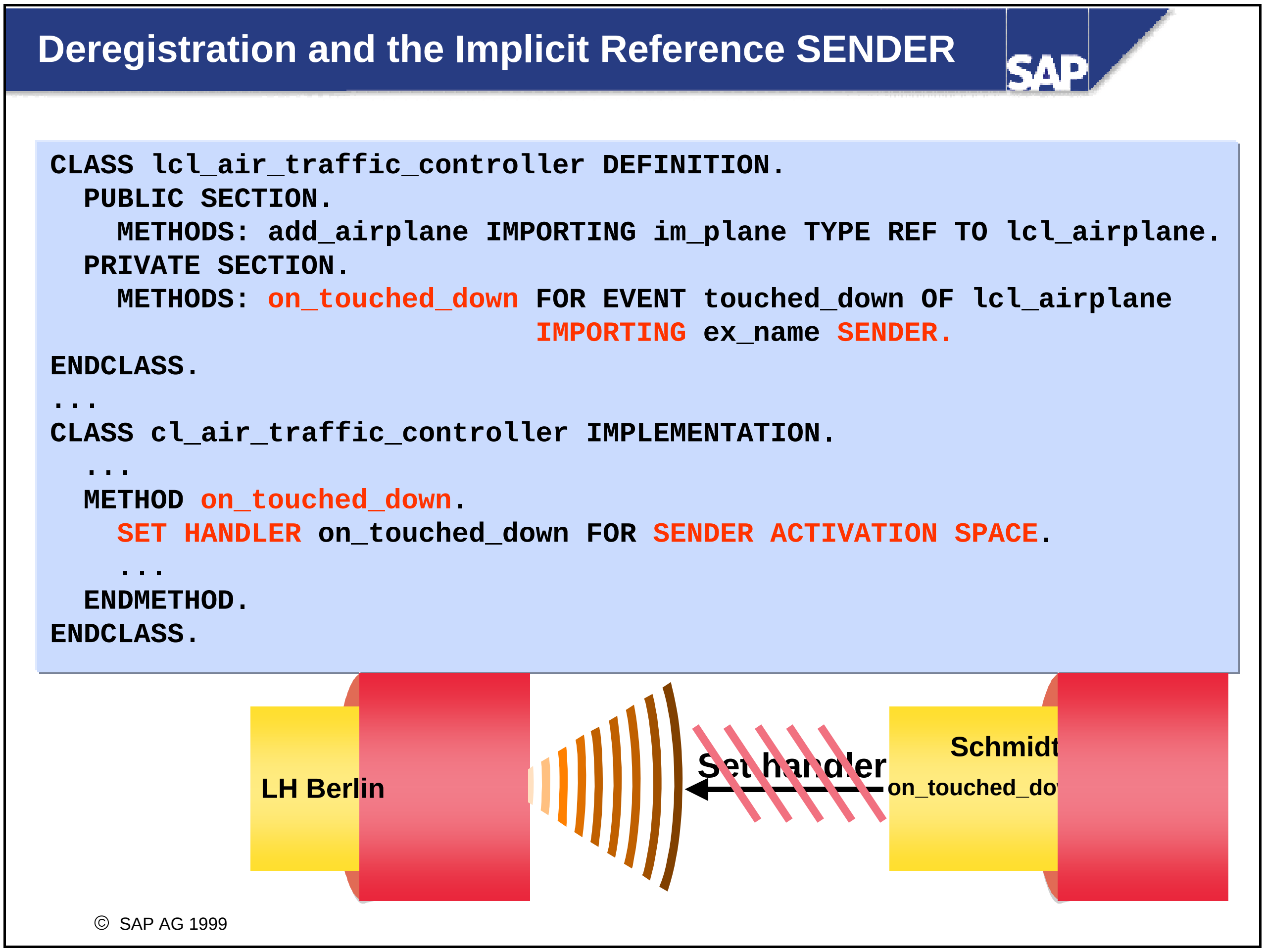
You can register an event using Activation ‘X‘ (see above example), and deregister it using Activation ‘SPACE‘ (see next slide). You can also register and deregister using a variable <var>, which is filled with one of these two values. If you do not specify a value for Activation, then the event is registered (default setting).

You can register several methods in one SET-HANDLER statement:

SET HANDLER <ref_handle1>-><handler_method1> ...

<ref_handlen>-><handler_methodN>

FOR <ref_sender> | FOR ALL INSTANCES.

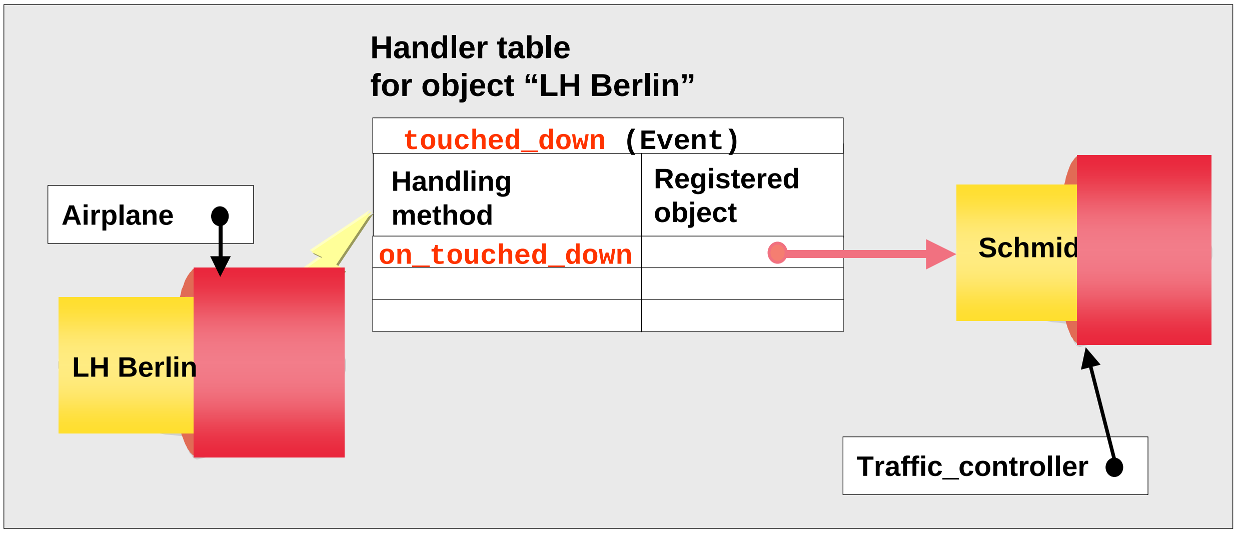


Deregistration and the Implicit Reference SENDER

```
CLASS lcl_air_traffic_controller DEFINITION.
  PUBLIC SECTION.
    METHODS: add_airplane IMPORTING im_plane TYPE REF TO lcl_airplane.
  PRIVATE SECTION.
    METHODS: on_touched_down FOR EVENT touched_down OF lcl_airplane
                            IMPORTING ex_name SENDER.
ENDCLASS.
...
CLASS cl_air_traffic_controller IMPLEMENTATION.
  ...
  METHOD on_touched_down.
    SET HANDLER on_touched_down FOR SENDER ACTIVATION SPACE.
    ...
  ENDMETHOD.
ENDCLASS.
```

In the above example, air-traffic controller Schmidt deregisters himself from the event touch_down for the airplane “LH Berlin” once it has landed, as the next time “LH Berlin” lands (again triggering touch_down) , this will be at a different airport and so of no interest to him.

Registration/Deregistration: Handler Table

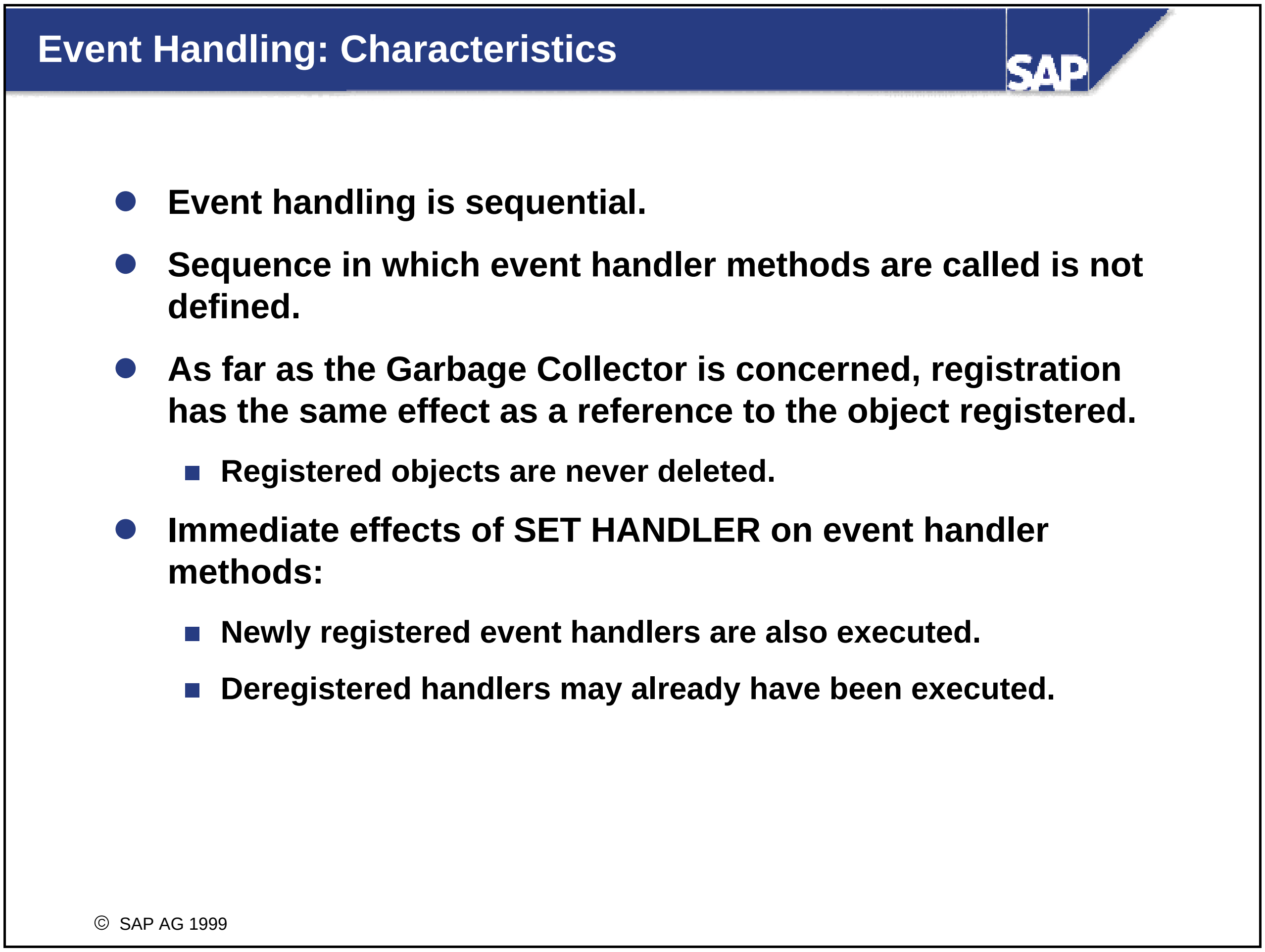


Handler table
for object “LH Berlin”

touched_down (Event)	
Handling method	Registered object
on_touched_down	

Every object that has defined events has an internal table: the handler table. All objects that have registered for events are entered in this table together with their event handler methods.

Objects that have registered themselves for an event that is still “live” also remain “live”. The methods of these objects are called when the event is triggered, even if they can no longer be reached using main memory references.

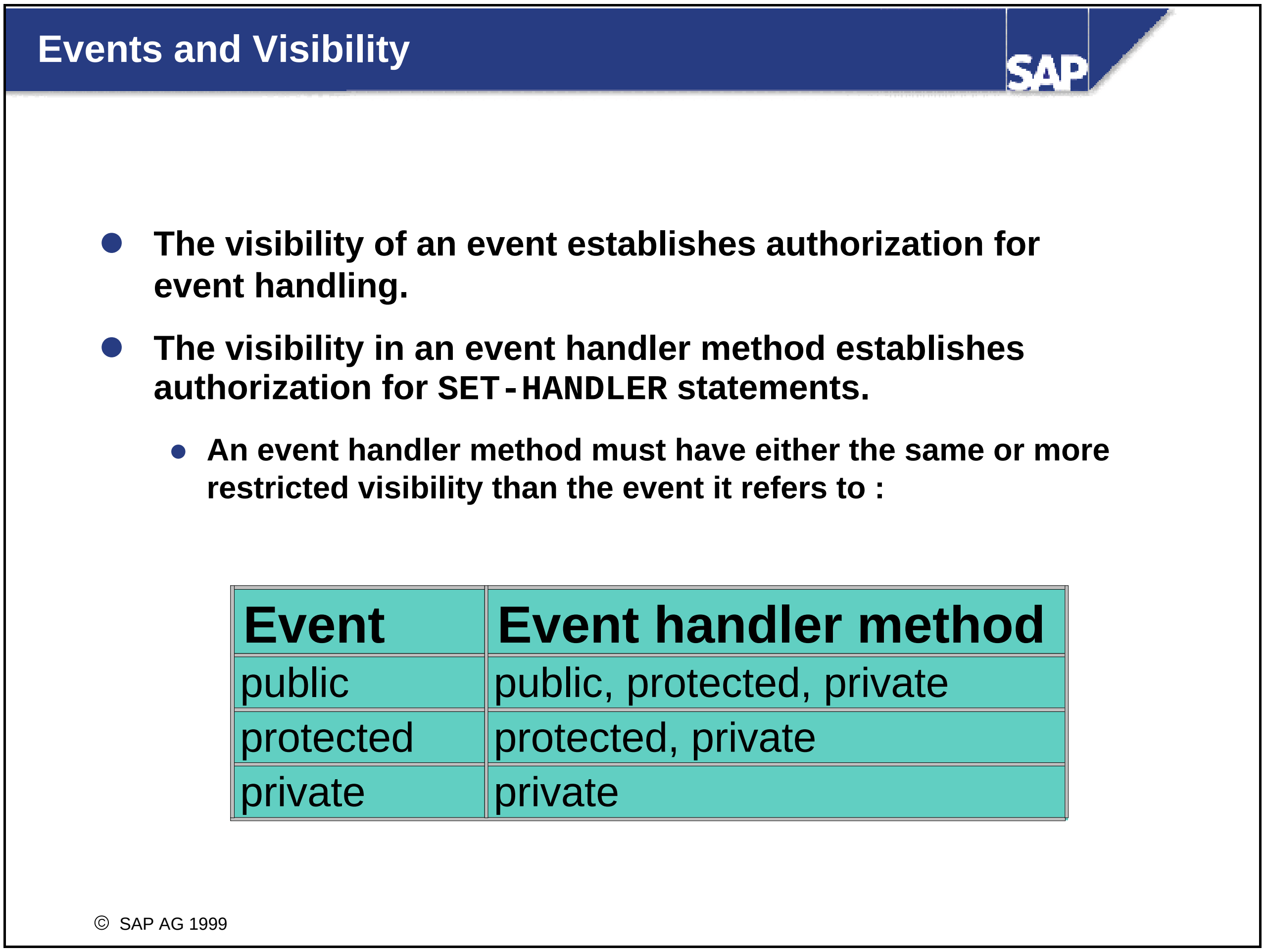


Event Handling: Characteristics

- Event handling is sequential.
- Sequence in which event handler methods are called is not defined.
- As far as the Garbage Collector is concerned, registration has the same effect as a reference to the object registered.
  - Registered objects are never deleted.
- Immediate effects of SET HANDLER on event handler methods:
  - Newly registered event handlers are also executed.
  - Deregistered handlers may already have been executed.

If several objects have registered themselves for an event, then the sequence in which the event handler methods are called is not defined, that is, there is no guaranteed algorithm for the sequence in which the event handler methods are called.

If a new event handler is registered in an event handler method for an event that has just been triggered, then this event handler is added to the end of the sequence and is then also executed when its turn comes. If an existing event handler is deregistered in an event handler method, then this handler is deleted from the event handler method sequence.



Events and Visibility

- **The visibility of an event establishes authorization for event handling.**
- **The visibility in an event handler method establishes authorization for `SET-HANDLER` statements.**
  - **An event handler method must have either the same or more restricted visibility than the event it refers to :**

Event	**Event handler method**
public	public, protected, private
protected	protected, private
private	private

Events are also subject to the visibility concept and can therefore be either public, protected or private. Visibility establishes authorization for event handling :

all users

only users within that class or its subclasses

only users in that class.

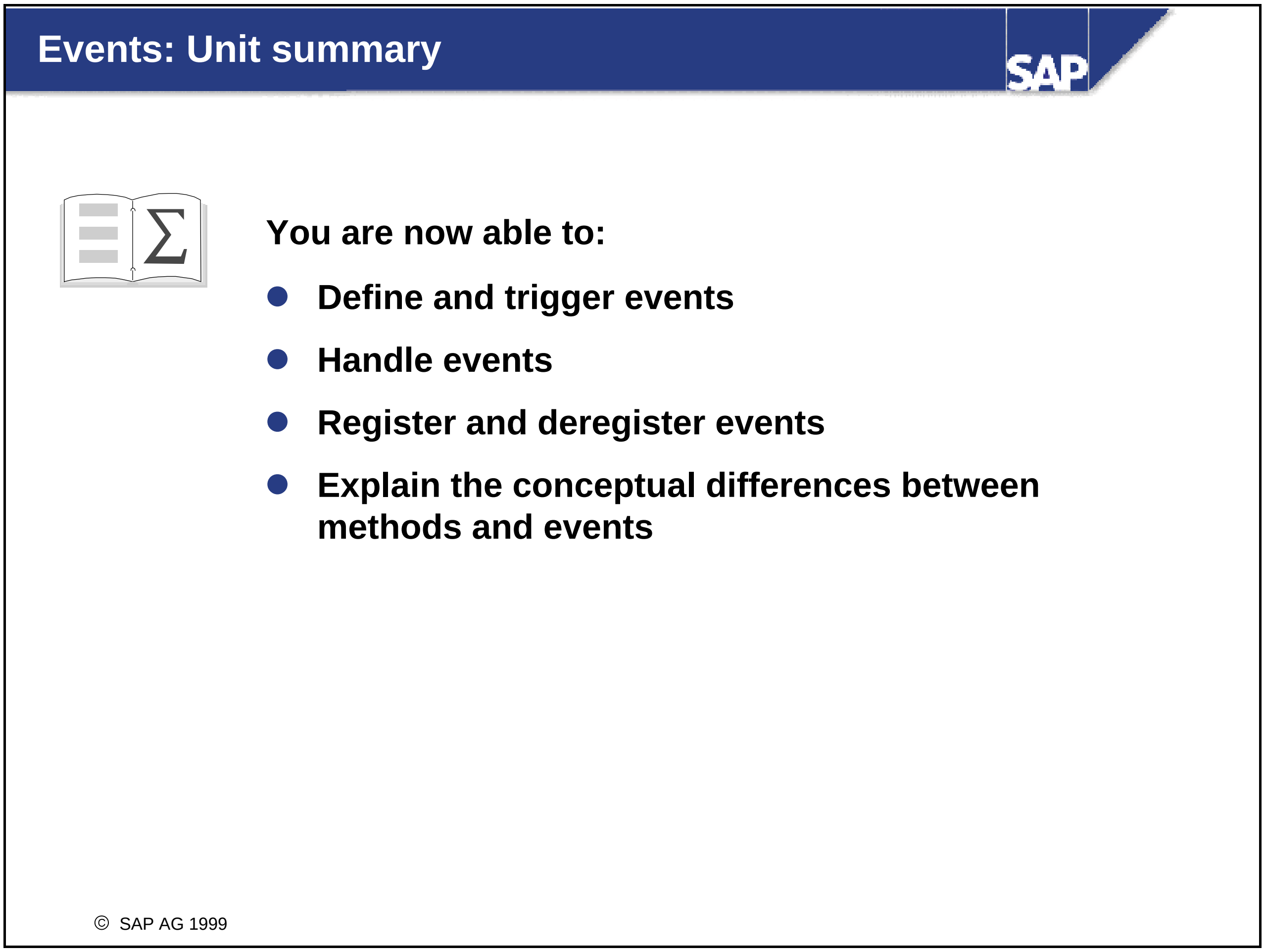
Event handler methods also have visibility characteristics. Event handler methods, however, can only have the same visibility or more restricted visibility than the events they refer to.

The visibility of event handler methods establishes authorization for SET-HANDLER statements: SET HANDLER statements can be made

anywhere

in that class and its subclasses

only in that class



Events: Unit summary

You are now able to:

- Define and trigger events
- Handle events
- Register and deregister events
- Explain the conceptual differences between methods and events

Exercises

Unit: Events

Topic: Events

At the end of this exercise you will be able to:

- Define and trigger events
- Handle events
- Register for events
-

An airline creates new flights and publicizes them in the media. Travel agencies can then include these flights in their offerings.

1-1 Go to the include program **ZBC404_##_LCL_CARRIER** and define an event *flight_created*, that you also trigger in the class.

1-1-1 The event is a public event that has two transfer parameters: *ex_connid* (type: *sflight-connid*) and *ex_fldate* (type: *sflight-fldate*).

1-1-2 The event should be triggered in the existing *create_a_new_flight* method, after the APPEND statement. Consider carefully how to pass the parameters.

1-2 Go to the include program **ZBC404_##_LCL_TRAVEL_AGENCY**, write a handler method for the *flight_created* event and register the travel agency to the *flight_created* event for all airlines that are business partners of the travel agency. The flights created by these airlines should be saved in the travel agency in a flight list .

1-2-1 In class *lcl_travel_agency*, create an internal table *list_of_flights* as a private attribute. The table should have the structure *bc404_flight_list_type*, which is already defined in the Dictionary. Examine the structure definition in the Dictionary.

Define a public instance method *add_a_new_flight* as a handler method for the *flight_created* event in class *lcl_carrier*. Enter the flight number (*ex_connid*), the flight date (*ex_fldate*) and a reference to the event trigger (*sender*) as IMPORTING parameters.

When implementing the *add_a_new_flight* method, enter the airline, the flight number and the flight date in the list of flights (*list_of_flights*). To do this, create a table work area in the method. This table work area must have the same structure as the internal table *list_of_flights*. Use the APPEND statement to fill the table.

Define the public method *subscribe_for_flight_creation* for registering the travel agency to the *flight_created* event. This method does not have any transfer parameters.

Since the travel agency only includes flights provided by airlines that are its business partners in its offering, it can only register itself to the *flight_created* event of these airlines. To do this, you must use a LOOP structure to determine all the business partners that are airlines in the internal table *list_of_business_partners* (since business partners also include hotels).
Warning: If, during registration (SET-HANDLER ...), you enter a reference to an object that has not implemented the corresponding event, you will get a runtime error.

Tip: In the method, define a reference to class *lcl_carrier* and carry out a widening cast check using the interface reference that you need to read from the internal table *list_of_business_partners*. Use the CATCH SYSTEM-EXCEPTIONS statement to catch the *move_cast_error* runtime error. If no runtime error occurs (*sy-subrc = 0*), then you can register the method *add_a_new_flight*.

Define and implement the public instance method *display_list_of_flights* to display the flight list *list_of_flights*. This method does not have any transfer parameters.

1-3 Go to program **ZBC404_##_MAIN.**

Comment out the line with the method call *display_business_partners*.

Register the travel agency to new flights provided by airlines that are its business partners. At the end of the program, call method *subscribe_for_flight_creation*.

An airline creates a new flight (method *create_a_new_flight*). Fill the transfer parameters with your own data.

Display the list of flights of your travel agency (method *display_list_of_flights*).

Unit: Events

Topic: Events

```
REPORT  sapbc404gens_interfaces .

INCLUDE sapbc404gens_lcl_airplane.
INCLUDE sapbc404gens_lcl_passenger_air.
INCLUDE sapbc404gens_lcl_cargo_air.
INCLUDE sapbc404gens_lif_busi_partners.
INCLUDE sapbc404gens_lcl_hotel.
* Definition and raising the event FLIGHT_CREATED
INCLUDE sapbc404eves_lcl_carrier.
* Subscribing and handling the event FLIGHT_CREATED
INCLUDE sapbc404eves_lcl_travel_agency.


DATA: passenger_airplane TYPE REF TO lcl_passenger_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane,
      carrier TYPE REF TO lcl_carrier,
      hotel1 TYPE REF TO lcl_hotel, hotel2 TYPE REF TO lcl_hotel,
      agency TYPE REF TO lcl_travel_agency.



START-OF-SELECTION.

  INCLUDE sapbc404eves_agency_partners.

  CREATE OBJECT agency EXPORTING im_name = 'Happy Holiday'.

  CALL METHOD agency->add_business_partner
                      EXPORTING im_business_partner = carrier.

  CALL METHOD agency->add_business_partner
                      EXPORTING im_business_partner = hotel1.

  CALL METHOD agency->add_business_partner
                      EXPORTING im_business_partner = hotel2.

*  call method agency->display_business_partners.

* Display empty flight list
```

```
  CALL METHOD agency->display_list_of_flights.

* Subscribe for event FLIGHT_CREATED of all business partners
  CALL METHOD agency->subscribe_for_flight_creation.

* One business partner is creating a new flight
  CALL METHOD carrier->create_a_new_flight
                    EXPORTING im_connid   = '815'
                              im_fldate   = '19991231'
                              im_airplane = passenger_airplane.

* Display flight list
  CALL METHOD agency->display_list_of_flights.
```

Include program SAPBC404GENS_LCL_AIRPLANE

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

  PUBLIC SECTION.

    TYPES: name_type(25) TYPE c.
    CONSTANTS: pos_1 TYPE i VALUE 30.

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

    CLASS-METHODS: display_n_o_airplanes.
```

```
* NEW: protected section
  PROTECTED SECTION.

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

  PRIVATE SECTION.

    CLASS-DATA: n_o_airplanes TYPE i.

ENDCLASS.


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

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

  METHOD set_attributes.
    name      = im_name.
    planetype = im_planetype.
  ENDMETHOD.

  METHOD display_attributes.
    WRITE: / 'Name of the airplane: '(001), AT pos_1 name,
           / 'Plane type:           '(002), AT pos_1 planetype.
  ENDMETHOD.




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

ENDCLASS.
```

Include program SAPBC404GENS_LCL_PASSENGER_AIR

```
*---------------------------------------------------------------*
*       CLASS lcl_passenger_airplane DEFINITION                  *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane DEFINITION INHERITING FROM
  lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
   im_name      TYPE name_type
                        im_planetype TYPE saplane-planetype
                        im_n_o_seats TYPE sflight-seatsmax,
             display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: n_o_seats TYPE sflight-seatsmax.

ENDCLASS.
```

```
*---------------------------------------------------------------*
*       CLASS lcl_passenger_airplane IMPLEMENTATION              *
*---------------------------------------------------------------*
CLASS lcl_passenger_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name      = im_name
                                  im_planetype = im_planetype.
    n_o_seats = im_n_o_seats.
  ENDMETHOD.

  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Number of seats:      '(003), 25 n_o_seats, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LCL_CARGO_AIR

```
*---------------------------------------------------------------*
*       CLASS lcl_cargo_airplane DEFINITION                      *
```

```
*---------------------------------------------------------------*
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING
   im_name      TYPE name_type
                        im_planetype TYPE saplane-planetype
                        im_cargo_max TYPE p,
            display_attributes REDEFINITION.

  PRIVATE SECTION.

    DATA: cargo_max TYPE scplane-cargomax.

ENDCLASS.




*---------------------------------------------------------------*
*       CLASS lcl_cargo_airplane IMPLEMENTATION                  *
*---------------------------------------------------------------*
CLASS lcl_cargo_airplane IMPLEMENTATION.

  METHOD constructor.
    CALL METHOD super->constructor EXPORTING
  im_name      = im_name
                              im_planetype = im_planetype.
    cargo_max = im_cargo_max.
  ENDMETHOD.

  METHOD display_attributes.
    CALL METHOD super->display_attributes.
    WRITE: / 'Maximal cargo:      '(004),
           at pos_1 cargo_max left-justified, /.
  ENDMETHOD.

ENDCLASS.
```

Include program SAPBC404GENS_LIF_BUSI_PARTNERS

```
INTERFACE lif_business_partners.

  METHODS: display_company_data.
```

```
ENDINTERFACE.
```

Include program SAPBC404GENS_LCL_HOTEL

```
*---------------------------------------------------------------*
*       CLASS lcl_hotel DEFINITION                               *
*---------------------------------------------------------------*
CLASS lcl_hotel DEFINITION.

  PUBLIC SECTION.

*   Interface declaration
    INTERFACES: lif_business_partners.

    METHODS: constructor IMPORTING im_name      TYPE string
                                   im_city      TYPE string
                                   im_n_o_rooms TYPE i.

  PRIVATE SECTION.

    DATA: name      TYPE string,
          city      TYPE string,
          n_o_rooms TYPE i.
ENDCLASS.


*---------------------------------------------------------------*
*       CLASS lcl_hotel IMPLEMENTATION                           *
*---------------------------------------------------------------*
CLASS lcl_hotel IMPLEMENTATION.

  METHOD constructor.
    name = im_name.
    city = im_city.
    n_o_rooms = im_n_o_rooms.
  ENDMETHOD.

* Interface Implementation
  METHOD lif_business_partners~display_company_data.
    WRITE: / 'Hotel '(h01), name COLOR COL_NEGATIVE, 'in'(h02), city,
           / 'The number of available rooms is'(h03),
             n_o_rooms left-justified, /.
  ENDMETHOD.
```

```
ENDCLASS.
```

Include program SAPBC404EVES_LCL_CARRIER

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

  PUBLIC SECTION.

    TYPES: BEGIN OF flight_list_type,
              connid   TYPE sflight-connid,
              fldate   TYPE sflight-fldate,
              airplane TYPE REF TO lcl_airplane,
              seatsocc TYPE sflight-seatsocc,
              cargo(5) TYPE p DECIMALS 3,
           END OF flight_list_type.

    INTERFACES: lif_business_partners.

    METHODS: constructor IMPORTING im_name TYPE string,
             get_name RETURNING value(ex_name) TYPE string,
             add_a_new_airplane IMPORTING
                        im_airplane TYPE REF TO lcl_airplane,
             create_a_new_flight IMPORTING
                         im_connid   TYPE sflight-connid
                         im_fldate   TYPE sflight-fldate
                         im_airplane TYPE REF TO lcl_airplane
                         im_seatsocc TYPE sflight-seatsocc
                                OPTIONAL
                         im_cargo    TYPE p OPTIONAL,
             display_airplanes.

*   Definition of event FLIGHT_CREATED
    EVENTS: flight_created EXPORTING
                       value(ex_connid) TYPE sflight-connid
                       value(ex_fldate) TYPE sflight-fldate.

  PRIVATE SECTION.

    DATA: name              TYPE string,
          list_of_airplanes TYPE TABLE OF REF TO lcl_airplane,
          list_of_flights   TYPE TABLE OF flight_list_type.
```

```
ENDCLASS.


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

  METHOD constructor.
    name = im_name.
  ENDMETHOD.

  METHOD get_name.
    ex_name = name.
  ENDMETHOD.

  METHOD lif_business_partners~display_company_data.
    DATA: n_o_airplanes TYPE i.
    DESCRIBE TABLE list_of_airplanes LINES n_o_airplanes.
    WRITE: / name COLOR COL_POSITIVE, / 'Number of Airplanes:'(c01),
             n_o_airplanes LEFT-JUSTIFIED, /.
  ENDMETHOD.

  METHOD add_a_new_airplane.
    APPEND im_airplane TO list_of_airplanes.
  ENDMETHOD.

  METHOD create_a_new_flight.
    DATA: wa_list_of_flights TYPE flight_list_type.

    wa_list_of_flights-connid   = im_connid.
    wa_list_of_flights-fldate   = im_fldate.
    wa_list_of_flights-airplane = im_airplane.
    IF im_seatsocc IS INITIAL.
      wa_list_of_flights-cargo = im_cargo.
    ELSE.
      wa_list_of_flights-seatsocc = im_seatsocc.
    ENDIF.
    APPEND wa_list_of_flights TO list_of_flights.
*   Raise event FLIGHT_CREATED
    RAISE EVENT flight_created EXPORTING ex_connid = im_connid
                                         ex_fldate = im_fldate.
  ENDMETHOD.

  METHOD display_airplanes.
    DATA airplane TYPE REF TO lcl_airplane.
    LOOP AT list_of_airplanes INTO airplane.
      CALL METHOD airplane->display_attributes.
    ENDLOOP.
  ENDMETHOD.

ENDCLASS.
```

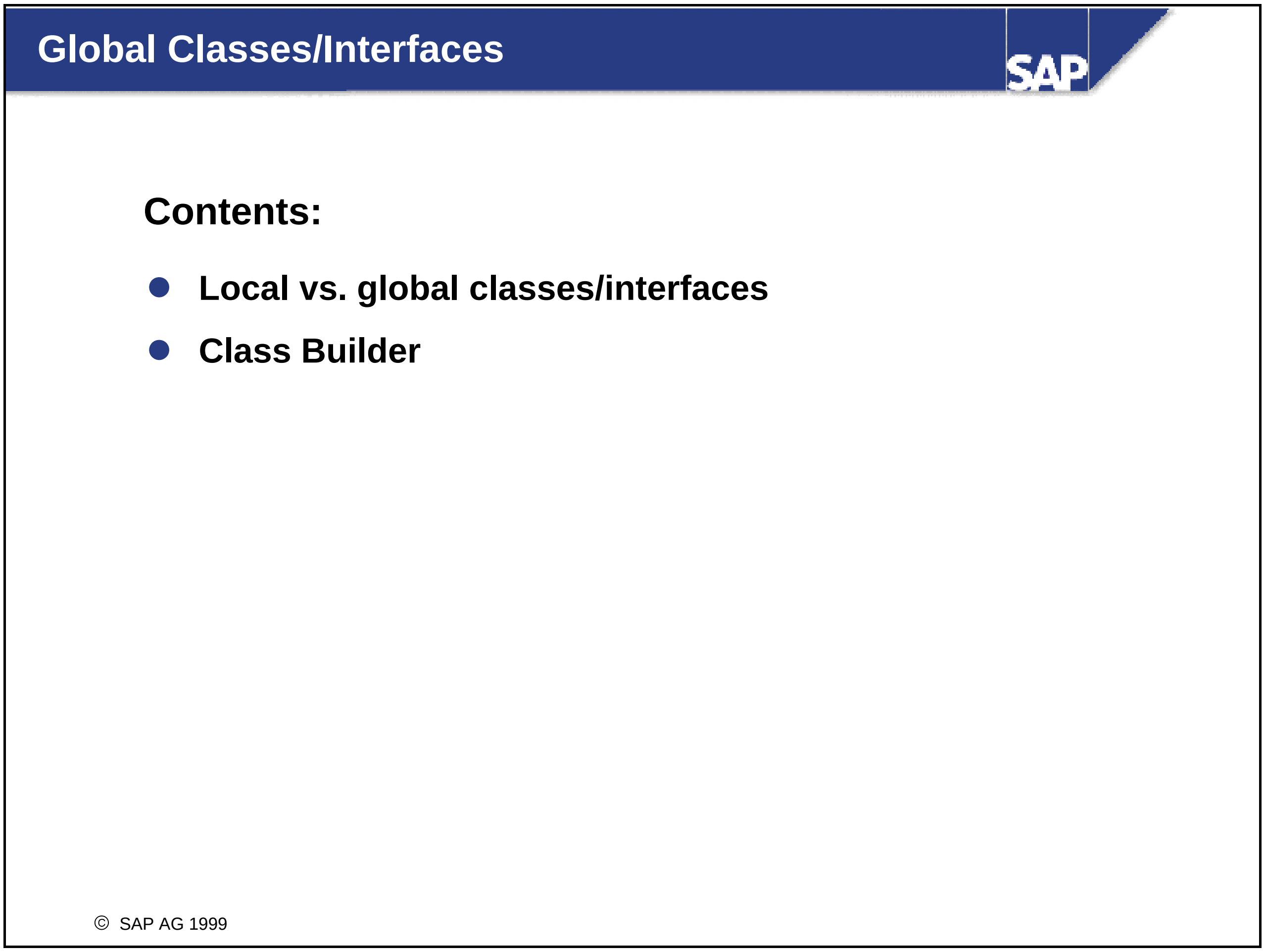
Include program SAPBC404EVES_LCL_TRAVEL_AGENCY

```
*---------------------------------------------------------------------*
*       CLASS lcl_travel_agency DEFINITION                            *
*---------------------------------------------------------------------*
CLASS lcl_travel_agency DEFINITION.

  PUBLIC SECTION.

    METHODS: constructor IMPORTING im_name TYPE string,
             add_business_partner IMPORTING im_business_partner
                          TYPE REF TO lif_business_partners,
             display_business_partners,
*            Subscribe for event FLIGHT_CREATED
             subscribe_for_flight_creation,
*            Handler method for event FLIGHT_CREATED
             add_a_new_flight FOR EVENT flight_created
   OF lcl_carrier
                       IMPORTING ex_connid ex_fldate sender,
*            Display flight list
             display_list_of_flights.

  PRIVATE SECTION.

    DATA: name TYPE string,
*         Internal table for flight list
          list_of_flights TYPE TABLE OF bc404_flight_list_type,
          list_of_business_partners TYPE TABLE OF REF TO
                                   lif_business_partners.

ENDCLASS.


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

```
  METHOD constructor.
    name = im_name.
  ENDMETHOD.

* Implementation of subscribe method
  METHOD subscribe_for_flight_creation.
    DATA: partner TYPE REF TO lif_business_partners,
          carrier type ref to lcl_carrier.
    LOOP AT list_of_business_partners INTO partner.
*     Attention: widening cast ...
      CATCH SYSTEM-EXCEPTIONS move_cast_error = 4.
        carrier ?= partner.
      ENDCATCH.
      IF sy-subrc = 0.
*     ... in SET HANDLER command
        SET HANDLER add_a_new_flight FOR partner.
      ENDIF.
    ENDLOOP.
  ENDMETHOD.
```

```
* Implementation of handler method
  METHOD add_a_new_flight.
    DATA: flight TYPE bc404_flight_list_type.
    flight-carrier = sender->get_name( ).
    flight-connid  = ex_connid.
    flight-fldate  = ex_fldate.
    APPEND flight TO list_of_flights.
  ENDMETHOD.

* Implementation of display flight list
  METHOD display_list_of_flights.
    DATA: flight TYPE bc404_flight_list_type.
    WRITE: / 'flight list of the travel agency'(ev1), name, /.
    LOOP AT list_of_flights INTO flight.
      WRITE: / flight-carrier, flight-connid, 30 flight-fldate.
    ENDLOOP.
    write: /, / 'created at'(ev2), sy-datum.
    skip 5.
  ENDMETHOD.

  METHOD add_business_partner.
    APPEND im_business_partner TO list_of_business_partners.
  ENDMETHOD.

  METHOD display_business_partners.
    DATA business_partner TYPE REF TO lif_business_partners.
    WRITE: / 'Travel Agency:'(ta1), name COLOR COL_HEADING,
           / 'Business partners:'(ta2), /.
    LOOP AT list_of_business_partners INTO business_partner.
      CALL METHOD business_partner->display_company_data.
    ENDLOOP.
  ENDMETHOD.

ENDCLASS.
```

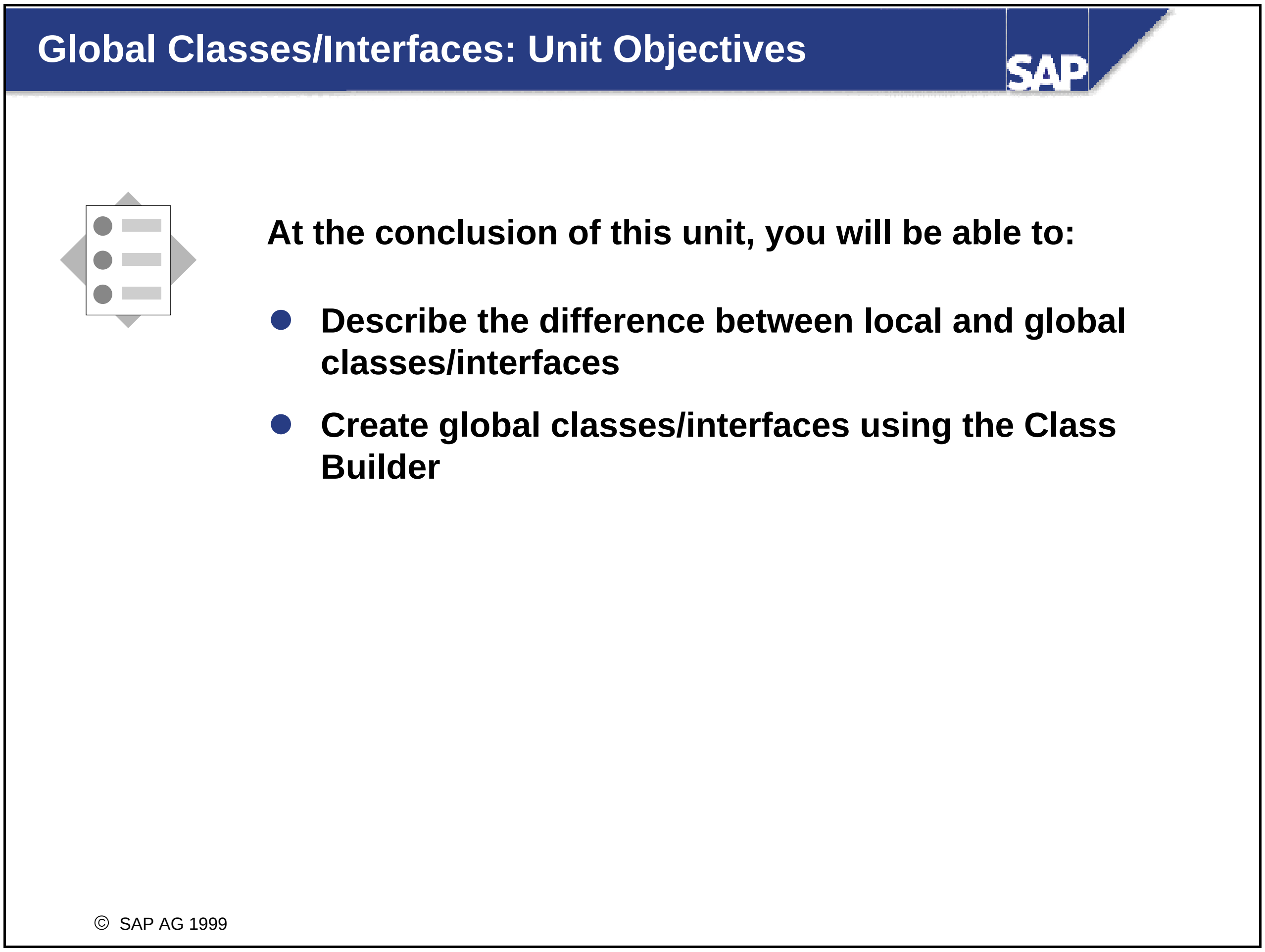
```
Include program SAPBC404EVES_AGENCY_PARTNERS
CREATE OBJECT passenger_airplane EXPORTING
im_name      = 'LH Berlin'
                  im_planetype = '747-400'
                  im_n_o_seats = 580.


 CREATE OBJECT cargo_airplane EXPORTING im_name = 'US Hercules'
                      im_planetype = 'Galaxy'
                      im_cargo_max = 30000.

 CREATE OBJECT carrier EXPORTING im_name = 'Lufthansa'.

 CALL METHOD carrier->add_a_new_airplane EXPORTING
                   im_airplane = passenger_airplane.

 CREATE OBJECT passenger_airplane EXPORTING
 im_name      = 'LH München'
                   im_planetype = 'A310-300'
                   im_n_o_seats = 280.

 CALL METHOD carrier->add_a_new_airplane EXPORTING
                   im_airplane = passenger_airplane.


 CALL METHOD carrier->add_a_new_airplane EXPORTING
                     im_airplane = cargo_airplane.


 create object hotel1 exporting im_name      = 'Budget Inn'
                  im_city      = 'Washington'
                  im_n_o_rooms = 112.

 create object hotel2 exporting im_name      = 'Ambassador'
                  im_city      = 'Frankfurt'
                  im_n_o_rooms = 85.
```



Global Classes/Interfaces

Contents:

- **Local vs. global classes/interfaces**
- **Class Builder**

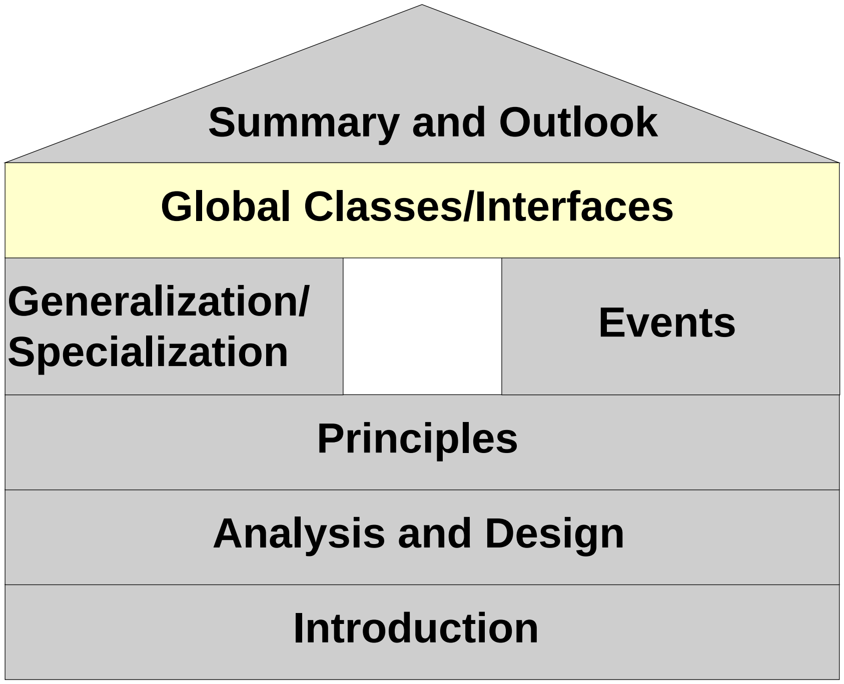


Global Classes/Interfaces: Unit Objectives

At the conclusion of this unit, you will be able to:

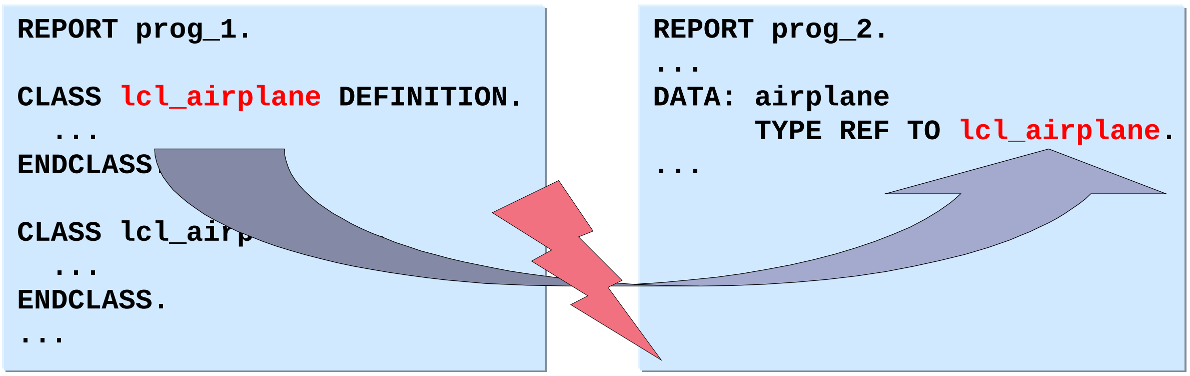
- Describe the difference between local and global classes/interfaces
- Create global classes/interfaces using the Class Builder

Global Classes/Interfaces: Course Overview Diagram



Local Classes/Interfaces

- **Local in program**
  - **Local classes are only valid in the program they were defined in**
- **Not stored in the Repository**
  - **No global access**



```
REPORT prog_1.

CLASS lcl_airplane DEFINITION.
  ...
ENDCLASS.

CLASS lcl_airp
  ...
ENDCLASS.
...
```

```
REPORT prog_2.
...
DATA: airplane
        TYPE REF TO lcl_airplane.
...
```

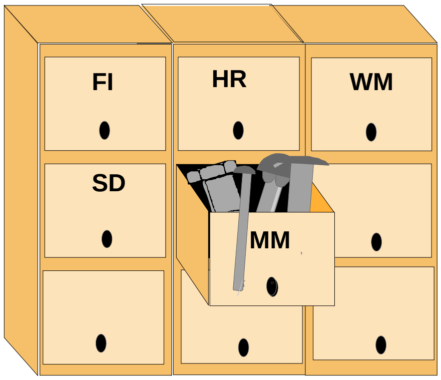
Local classes/interfaces are only known within the program in which they are defined and implemented.

Local classes/interfaces are not stored in the Repository (no TADIR entry). There is no “global” access to these classes/interfaces (for example, from other programs).

If a local class is implemented in an include which is then embedded in two different programs, then references to the “same” class still cannot be exchanged at runtime. Two classes that do not conform to type are created at runtime.

Global Classes/Interfaces

- Stored in Repository
  - Access from all programs using `TYPE REF TO`
- Class and interface names governed by the SAP namespace concept
  - Customer namespace: Y* or Z*
- Where-used list available
- Own maintenance tool
  - Transaction SE24 - Class Builder

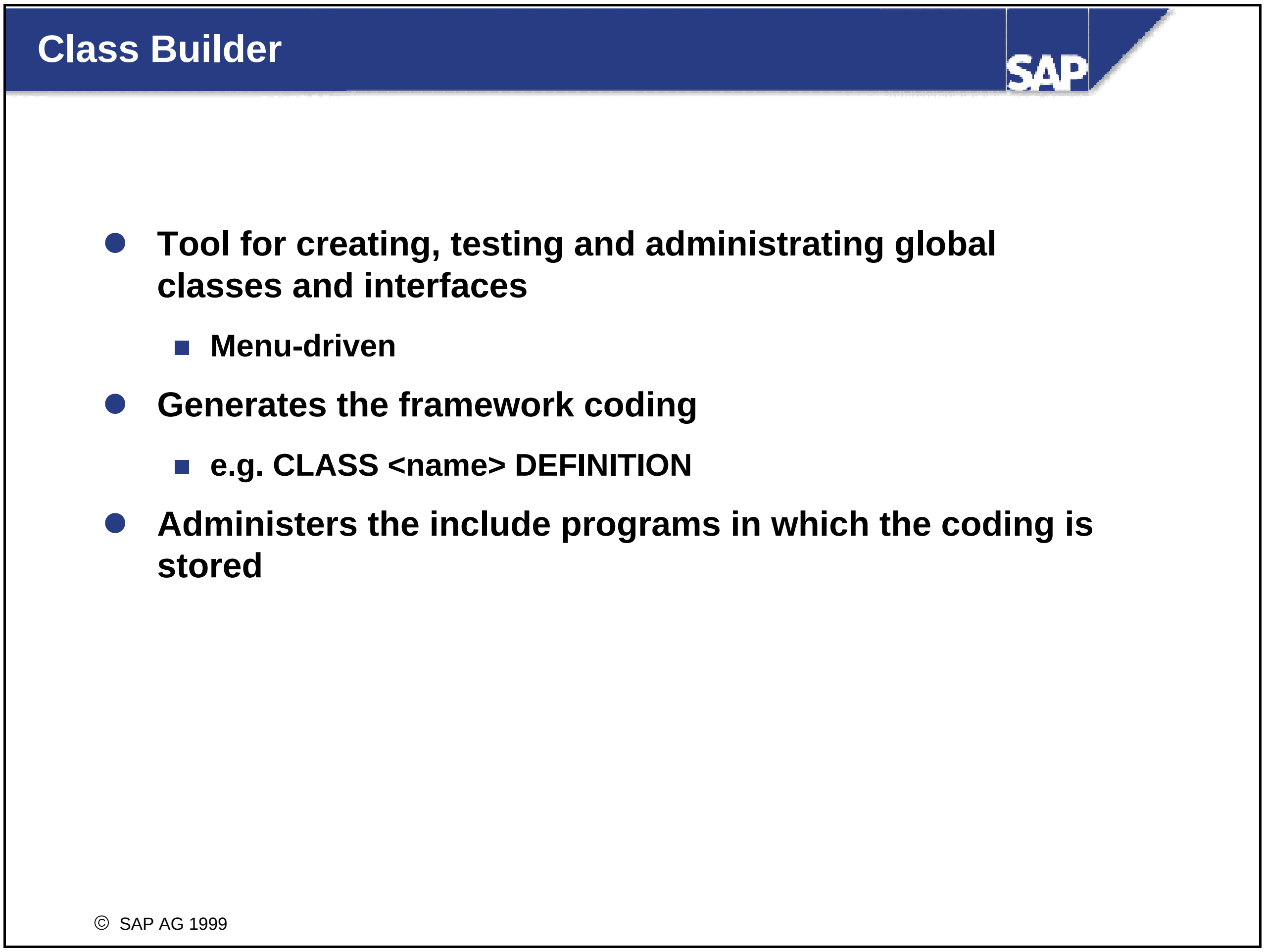


Unlike local in program classes/interfaces, global classes/interfaces can be created and implemented using the ABAP Workbench Tool *Class Builder* or transaction SE24. These classes/interfaces are then available to all developers.

Global class and interface names share the same namespace.

Global classes/interfaces have a TADIR entry: R3TR CLASS <name>

The smallest transport unit is method LIMU METH.

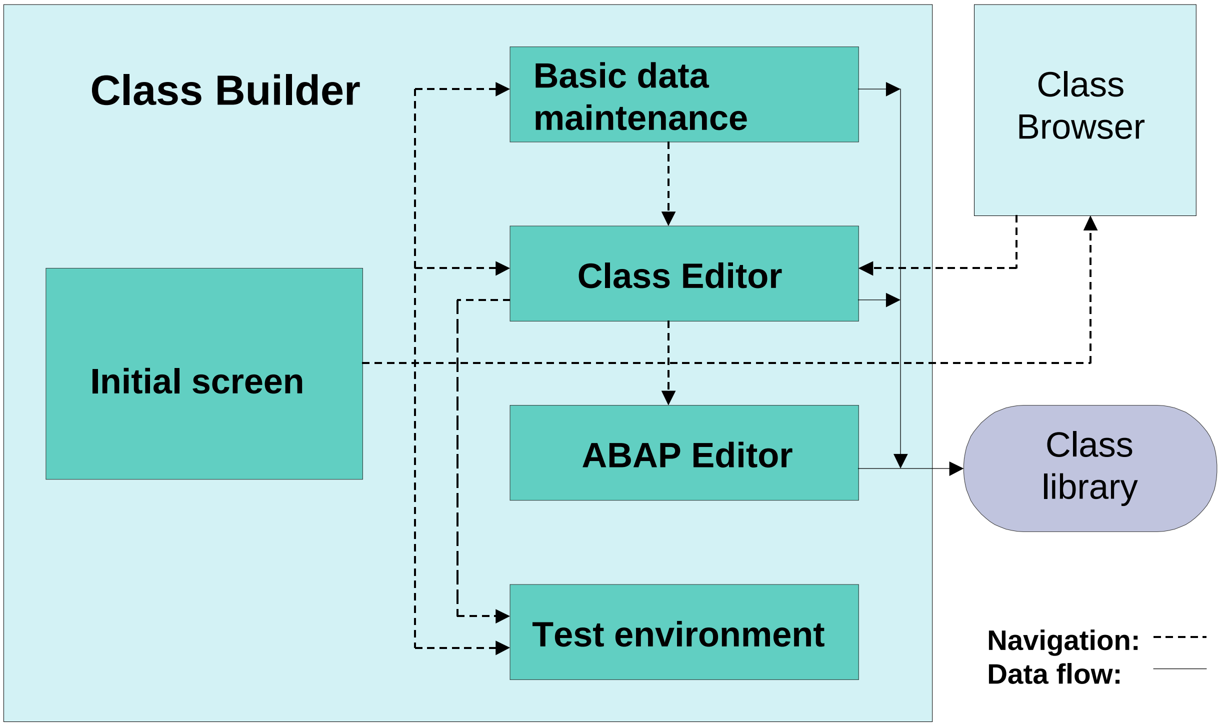


Class Builder

- Tool for creating, testing and administrating global classes and interfaces
  - Menu-driven
- Generates the framework coding
  - e.g. CLASS <name> DEFINITION
- Administers the include programs in which the coding is stored

The *Class Builder* is a tool in the ABAP Workbench that is used to create, define and test global ABAP classes and interfaces.

Class Builder: Structure

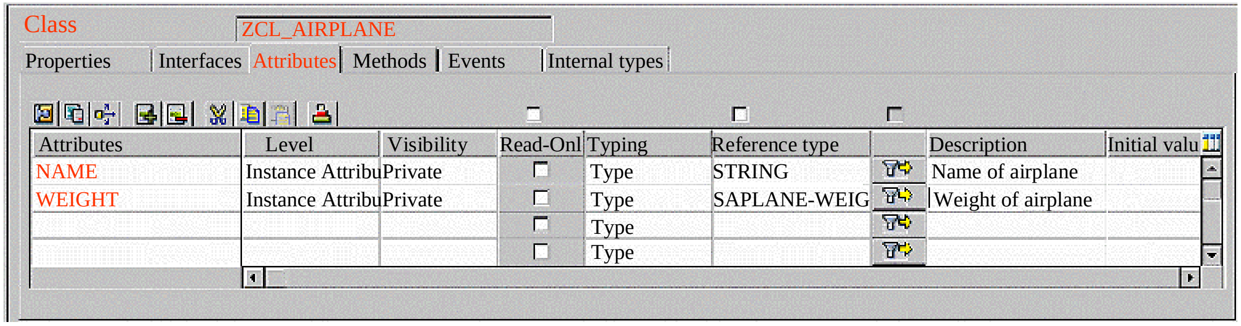


In the initial screen, select the object type you want to work with - class or interface. Then choose one of the display, change, create or test functions.

In the initial screen you have the choice of viewing the contents of the R/3 class library using the *Class Browser* or going straight to basic data maintenance of the object types and the *Class Editor,* where you can define the object types and their components. The object type definition can be immediately followed by method implementation in the ABAP Editor. You can also access the test environment from the initial screen or from the Class Editor.

Class Builder: Global Classes

Class Editor



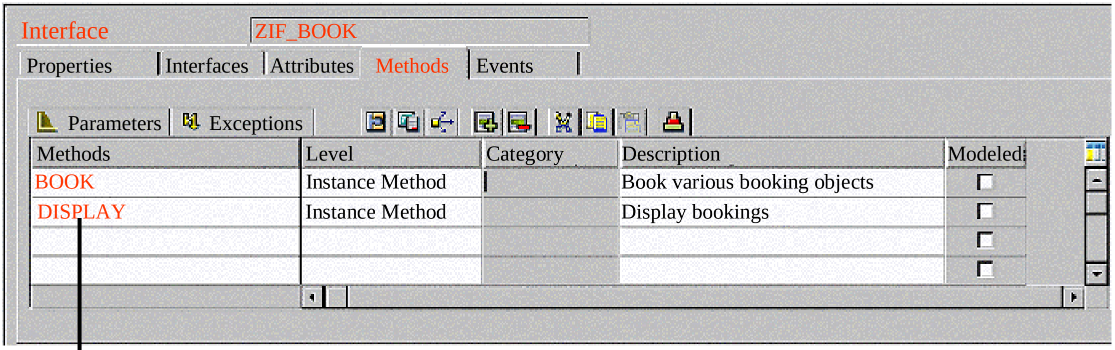
Class ZCL_AIRPLANE

Attributes	Level	Visibility	Read-Onl	Typing	Reference type		Description	Initial valu
NAME	Instance Attribu	Private	□	Type	STRING		Name of airplane	
WEIGHT	Instance Attribu	Private	□	Type	SAPLANE-WEIG		Weight of airplane	
			□	Type				
			□	Type				

In the Class Builder you have the same options for creating a global class as for creating a local class.

Class Builder: Global Interfaces

Class Editor



Interface ZIF_BOOK

Methods	Level	Category	Description	Modeled
BOOK	Instance Method		Book various booking objects	□
DISPLAY	Instance Method		Display bookings	□
				□
				□

Double click

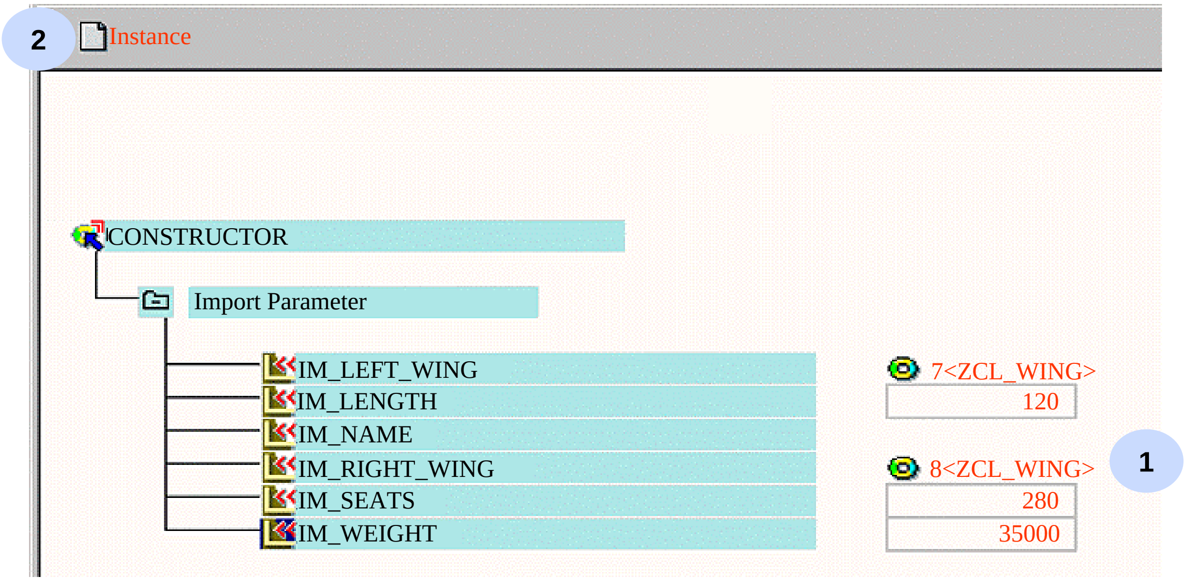
ABAP Editor

In the Class Builder you have the same options for creating a global interface as for creating a local interface.

Class Builder: Testing Classes (1)

- Create objects

Test environment

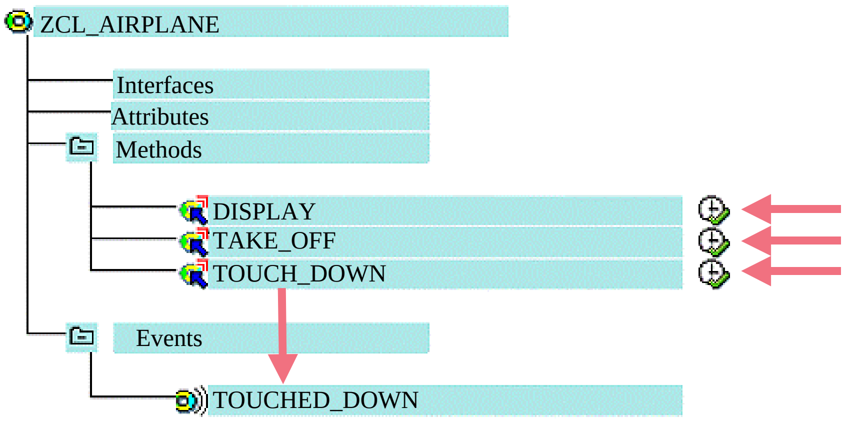


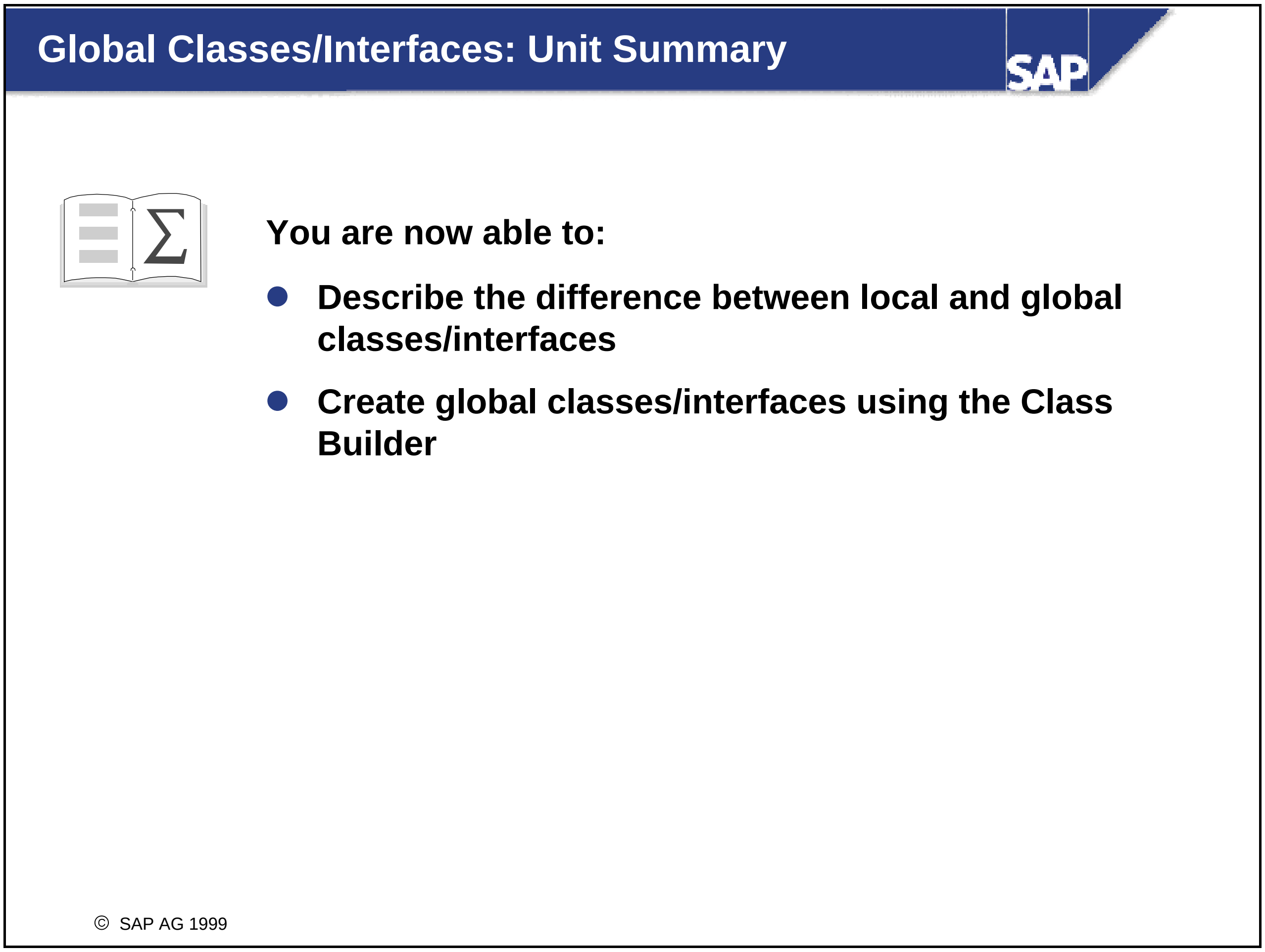
© SAP AG 1999

Class Builder: Testing Classes (2)

- Test methods
- Trigger events

Test environment





Global Classes/Interfaces: Unit Summary

You are now able to:

- Describe the difference between local and global classes/interfaces
- Create global classes/interfaces using the Class Builder

Global Classes/Interfaces Exercises

Unit: Global Classes/Interfaces

Topic: Global Classes

At the end of this exercise you will be able to:

- Create a global class in the Class Builder
- Test a global class in the Class Builder
- Use global classes in programs
-

An airline needs to manage its airplanes.

1-1 Create global class *zcl_##_airplane* in the Class Builder. Define the global class similarly to the local class *lcl_airplane* (include program **ZBC404_##_LCL_AIRPLANE**).

1-1-1 Attributes:
- *name* (protected instance attribute, type *string*)
– planetyp (protected instance attribute, type *saplane-planetyp*)
– *n_o_airplanes* (private static attribute, type *i*)

1-1-2 Methods:
- *constructor* (parameters: im_name, im_planetype)
– *display_attributes* (no parameters)
– *display_n_o_airplanes* (no parameters)
You can copy the source code for the methods from your definition of the local class lcl_airplane (program **ZBC404_##_LCL_AIRPLANE**).

1-2 Test your global class *zcl_##_airplane* in the Class Builder.

1-2-1 Create an instance.

1-2-2 Call methods *display_attributes* and *display_n_o_airplanes*.

1-3 Go to program **ZBC404_##_MAINTAIN_AIRPLANES.**

Comment out the INCLUDE statement with **ZBC404_##_LCL_AIRPLANE**.

Replace *lcl_airplane* throughout the program with *zcl_##_airplane*.

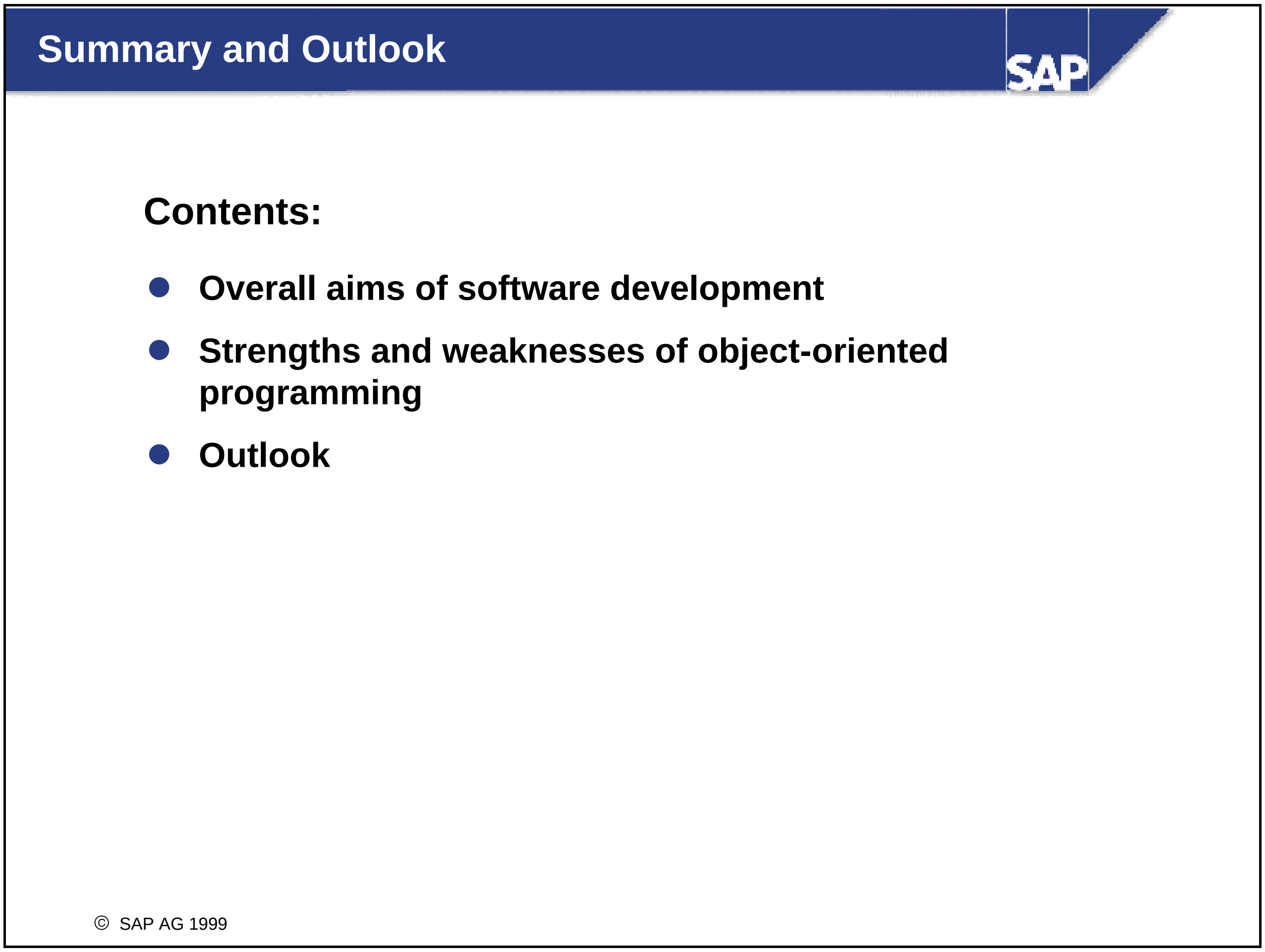
Start the program.

Global Classes/Interfaces Solutions

Unit: Global Classes/Interfaces

Topic: Global Classes

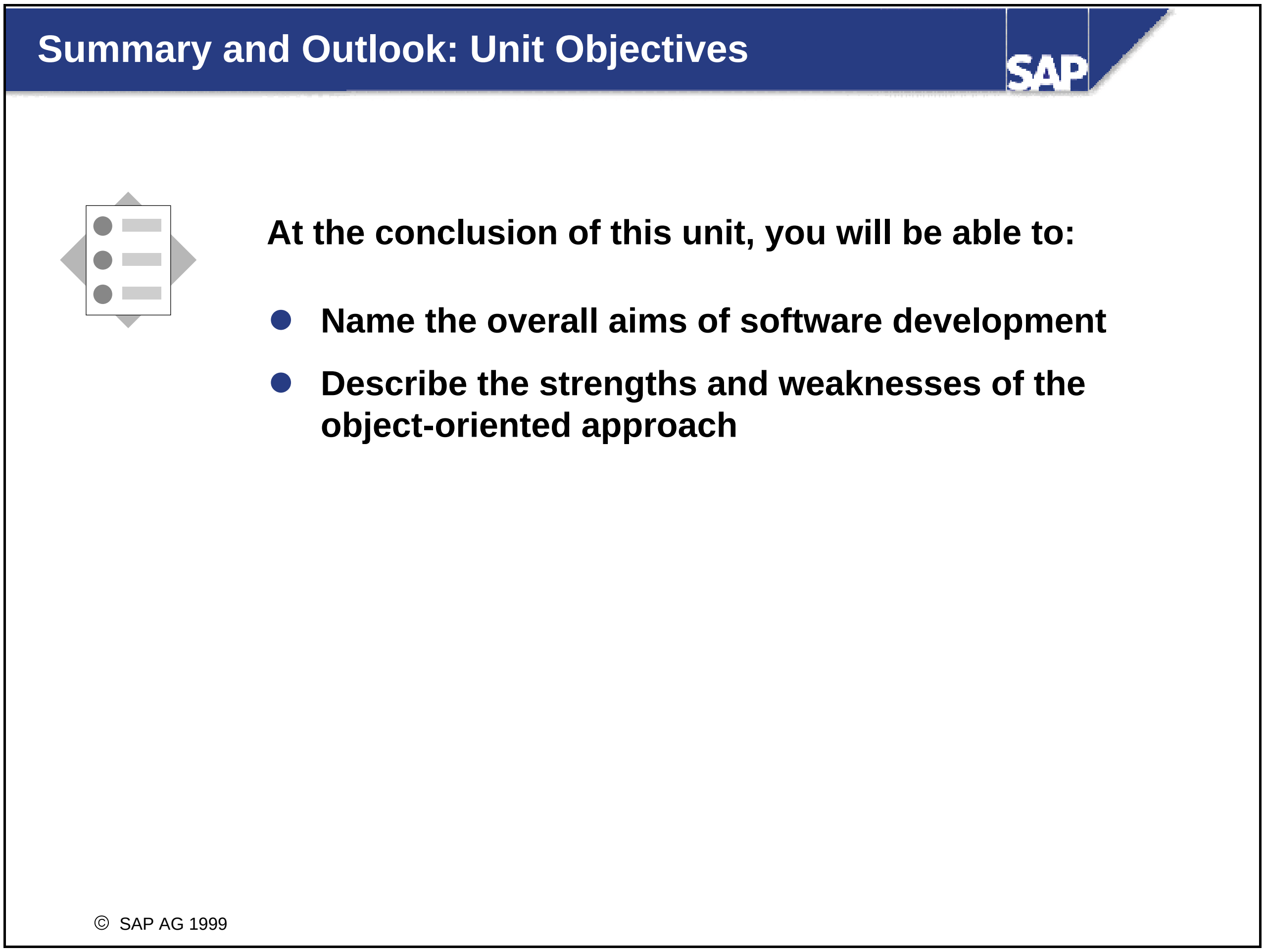
```
REPORT  sapbc404glos_cl_airplane      .

* No use of the local implementation
* include sapbc404bass_lcl_airplane_2.

* Use of global class BC404_CL_AIRPLANE
DATA: airplane TYPE REF TO bc404_cl_airplane.


START-OF-SELECTION.

* Use of global class BC404_CL_AIRPLANE
  CALL METHOD bc404_cl_airplane=>display_n_o_airplanes.

  CREATE OBJECT airplane EXPORTING im_name      = 'LH Berlin'
                        im_planetype = '747-400'.

  CALL METHOD airplane->display_attributes.

* Use of global class BC404_CL_AIRPLANE
  CALL METHOD bc404_cl_airplane=>display_n_o_airplanes.
```



Summary and Outlook

Contents:

- Overall aims of software development
- Strengths and weaknesses of object-oriented programming
- Outlook

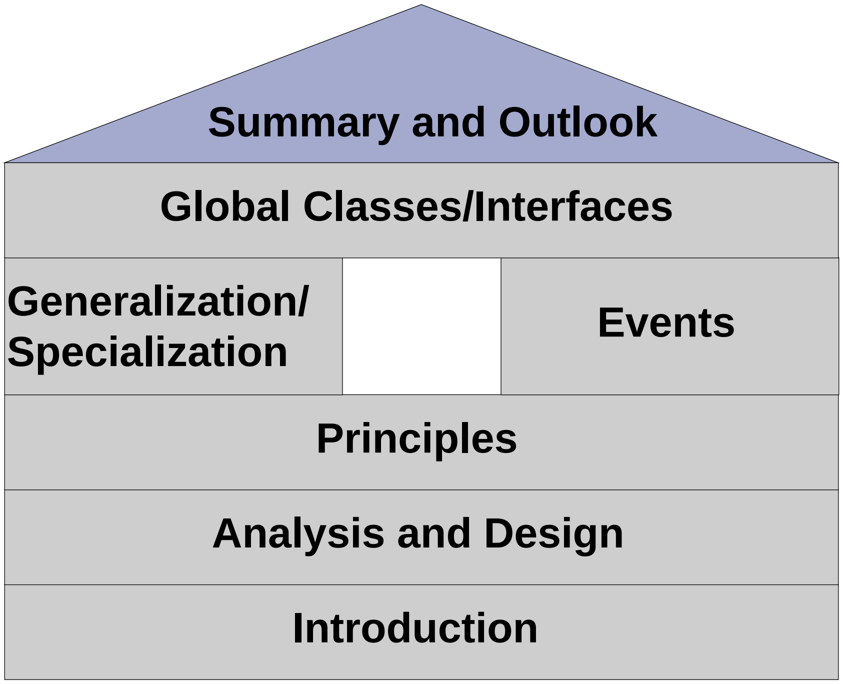


Summary and Outlook: Unit Objectives

At the conclusion of this unit, you will be able to:

- Name the overall aims of software development
- Describe the strengths and weaknesses of the object-oriented approach

Summary and Outlook: Course Overview Diagram



Summary and Outlook

Global Classes/Interfaces

Generalization/ Specialization

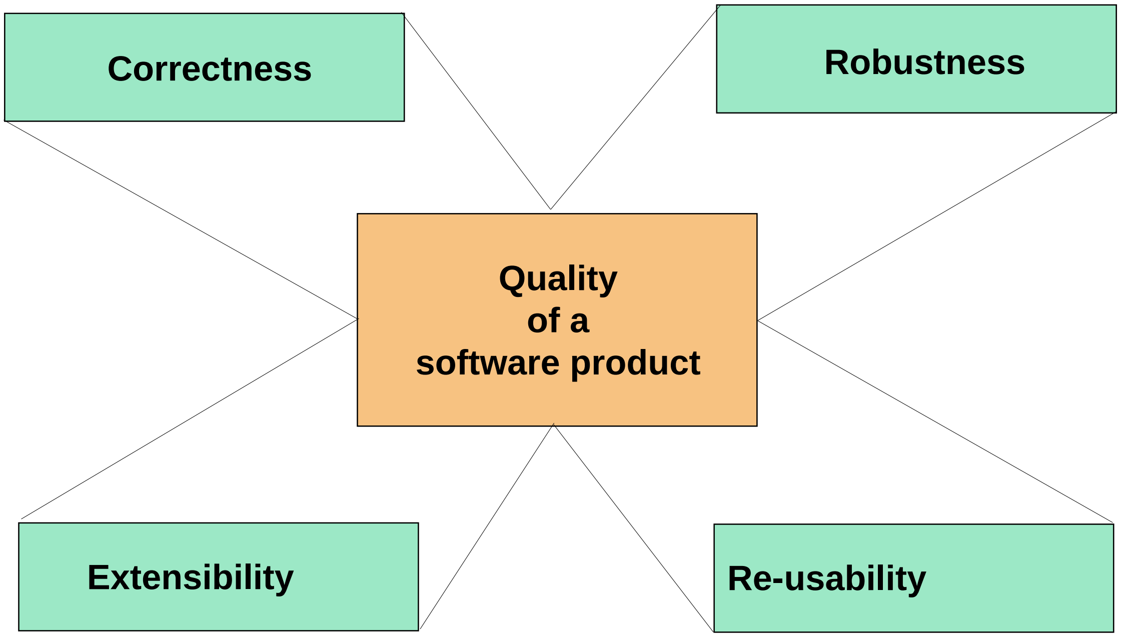
Events

Principles

Analysis and Design

Introduction

Overall Aims of Software Development



Correctness

Robustness

Quality of a software product

Extensibility

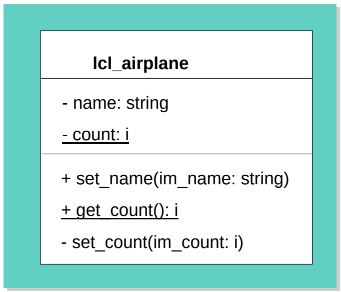
Re-usability

In the early stages of programming history, in the 1970s and 1980s, the principle aim was to write programs that were correct and robust. A program is considered correct if it does exactly what is said in the program specification. A program is considered robust if it can react appropriately to (user) errors and does not just crash immediately.

As programs grew in scope and complexity, more attention began to be paid to the possibilities of extensibility and re-usability, in order to avoid constantly having to re-invent the wheel. Extensibility is the facility to enhance an existing program by adding new functions, while still using it in the same context. Re-usability, on the other hand, is when a program or part of a program is taken out of its own context and recycled in another context, that is, as part of another program that has different tasks.

Characteristics of Object-Oriented Programming (1)

- Classes
  - Summarization of data and functionality into an “independent” software unit
  - Simple re-use of coding



lcl_airplane

- name: string

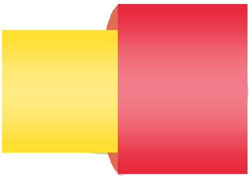
- count: i

+ set_name(im_name: string)

+ get_count(): i

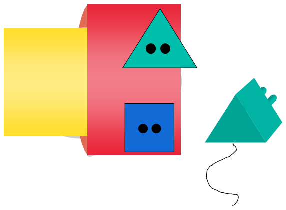
- set_count(im_count: i)

- Encapsulation
  - Communication only using interfaces (public components)
  - No dependency on actual implementation

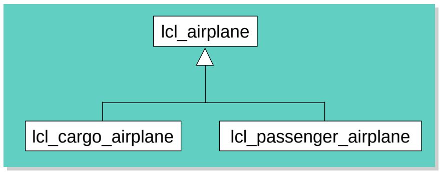


Characteristics of Object-Oriented Programming (2)

- **Polymorphism**
  - **Programs can be extended with minimum effort**

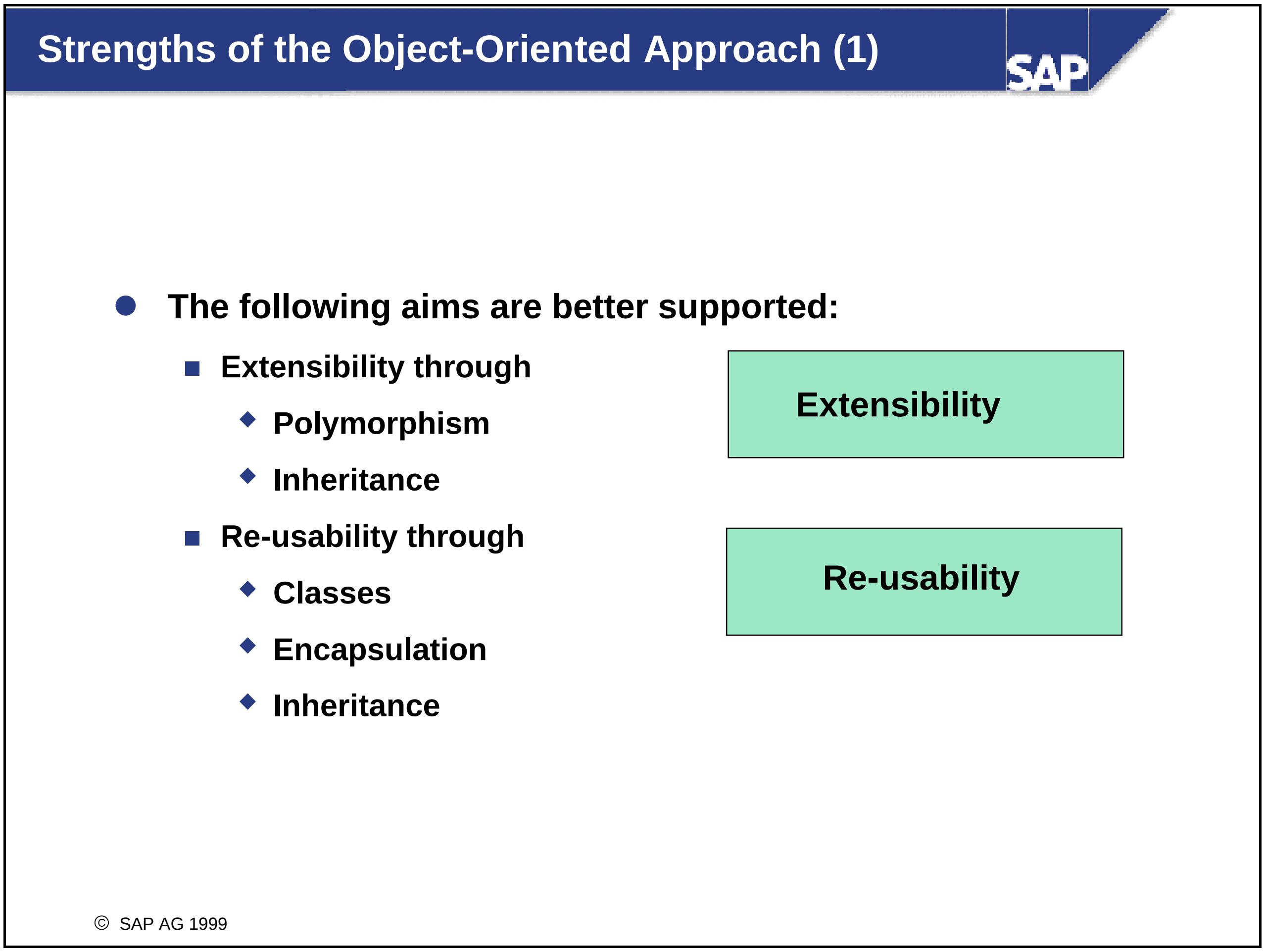


- **Inheritance**
  - **Re-use of implementations**



- **Think in terms of responsibilities**
  - **Question: which class is responsible?**
  - **Avoid redundancies**

© SAP AG 1999



Strengths of the Object-Oriented Approach (1)

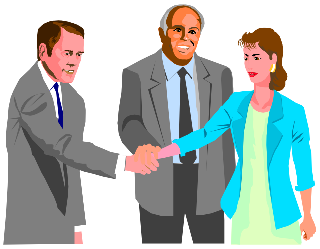
- The following aims are better supported:
  - Extensibility through
    - Polymorphism
    - Inheritance
  - Re-usability through
    - Classes
    - Encapsulation
    - Inheritance

Extensibility

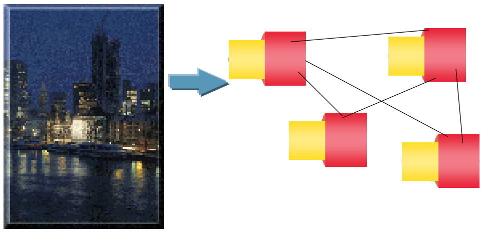
Re-usability

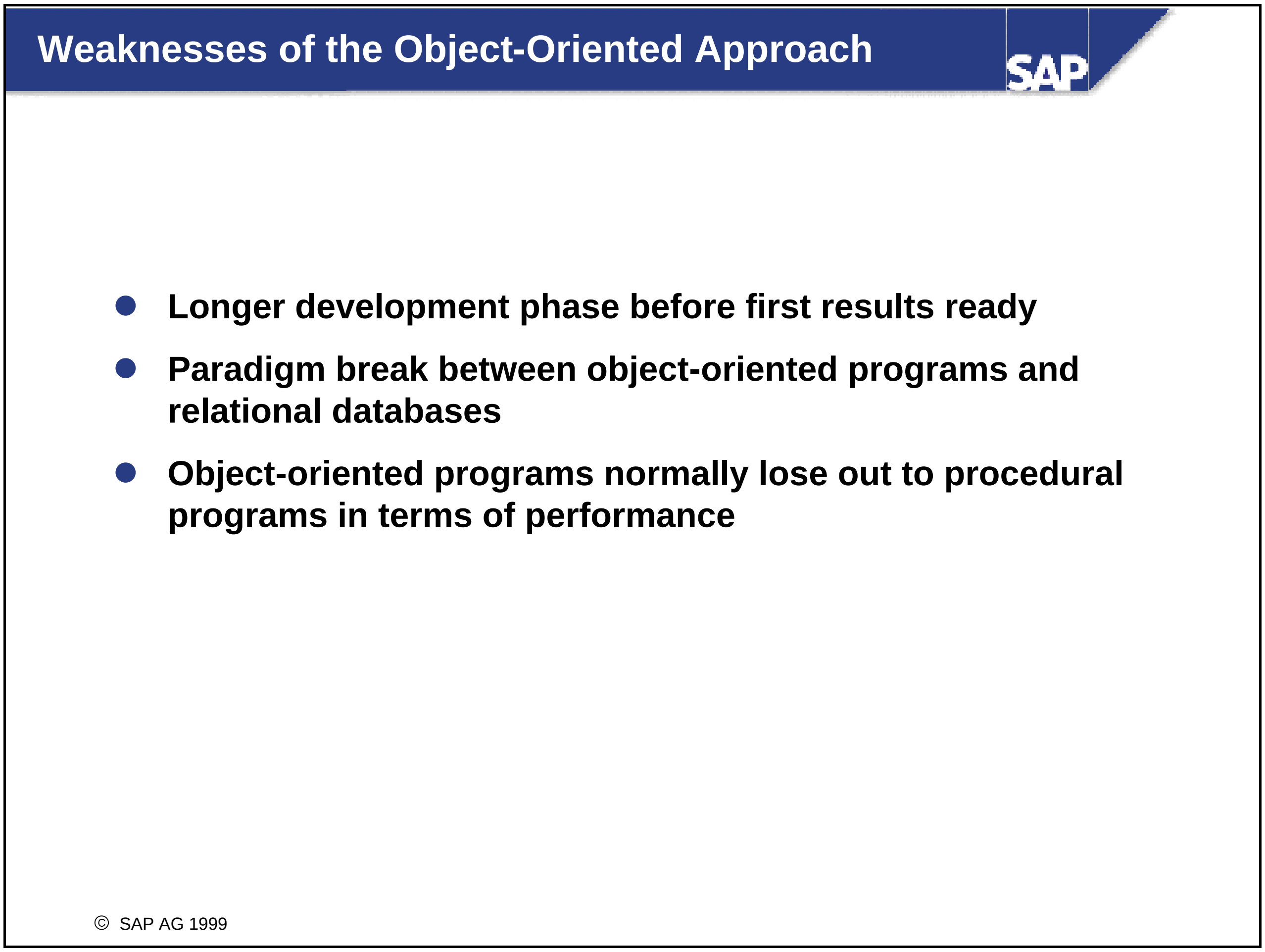
Strengths of the Object-Oriented Approach (2)

- Uniform language throughout the development process
  - All participants
  - In all phases



- Reality is reflected in appropriate software concepts
  - Objects -> objects
  - Their state -> attributes
  - Their functions -> methods

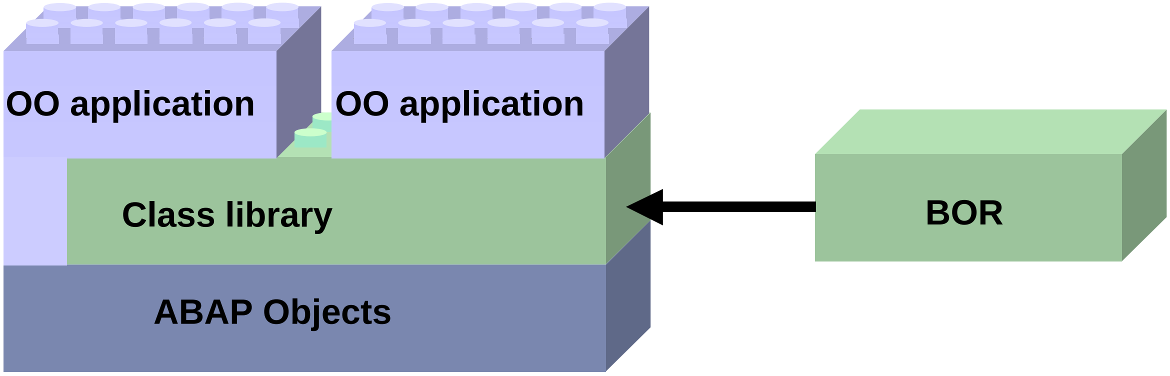




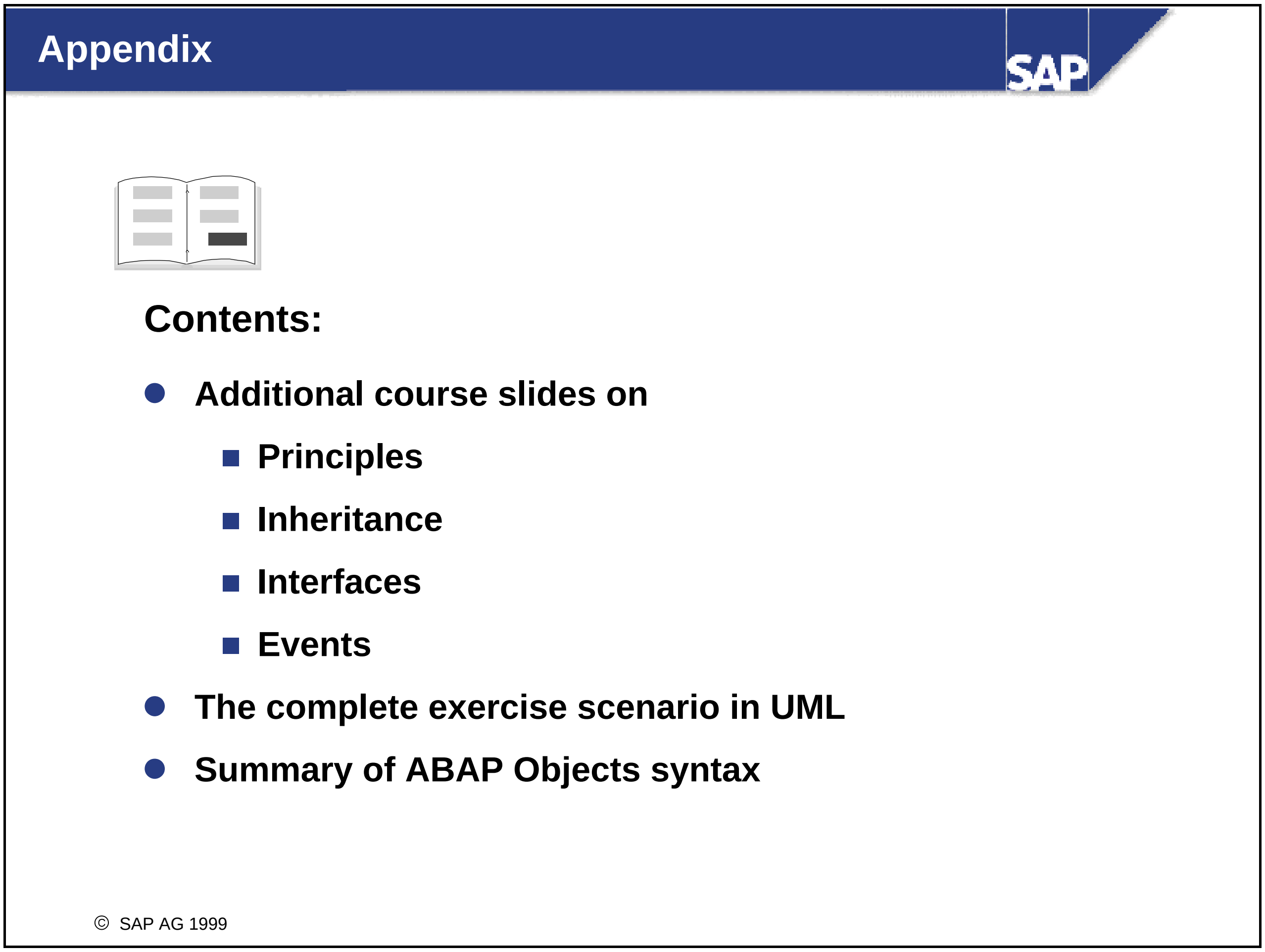
Weaknesses of the Object-Oriented Approach

- Longer development phase before first results ready
- Paradigm break between object-oriented programs and relational databases
- Object-oriented programs normally lose out to procedural programs in terms of performance

Outlook



The Business Object Repository, the object-oriented view of the R/3 System, will be migrated to the Class Library by Rel5.0. Then classes, such as customer or invoice, will be available globally in the system for use by any application.



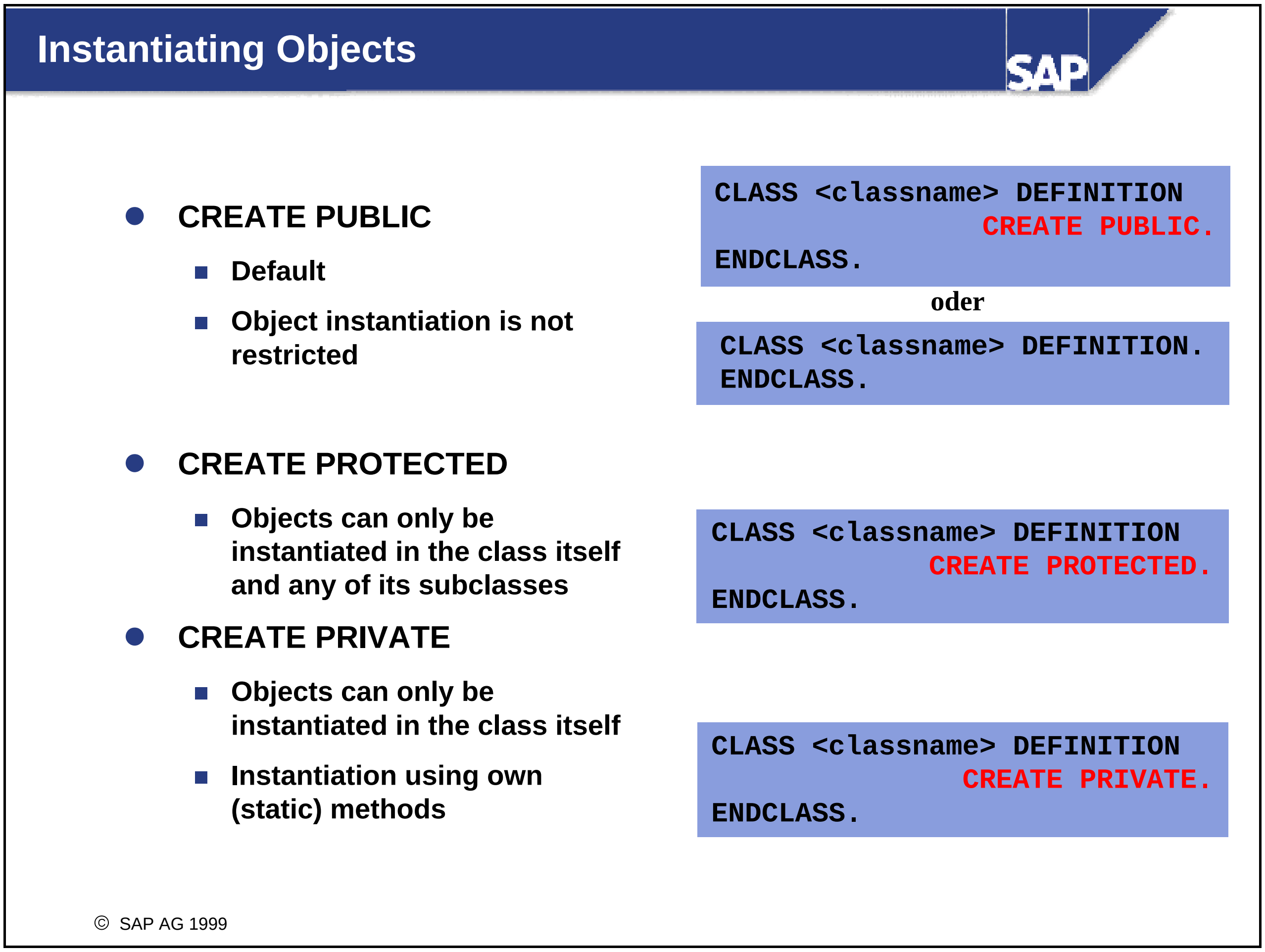
Appendix

Contents:

- Additional course slides on
  - Principles
  - Inheritance
  - Interfaces
  - Events
- The complete exercise scenario in UML
- Summary of ABAP Objects syntax

Appendix: Overview (1)

- Principles
- Inheritance
- Interfaces
- Events
- Exercise Scenario
- Summary of Syntax



Instantiating Objects

- CREATE PUBLIC
  - Default
  - Object instantiation is not restricted

```
CLASS <classname> DEFINITION
                     CREATE PUBLIC.
ENDCLASS.
```

oder

```
CLASS <classname> DEFINITION.
ENDCLASS.
```

- CREATE PROTECTED
  - Objects can only be instantiated in the class itself and any of its subclasses

```
CLASS <classname> DEFINITION
                 CREATE PROTECTED.
ENDCLASS.
```

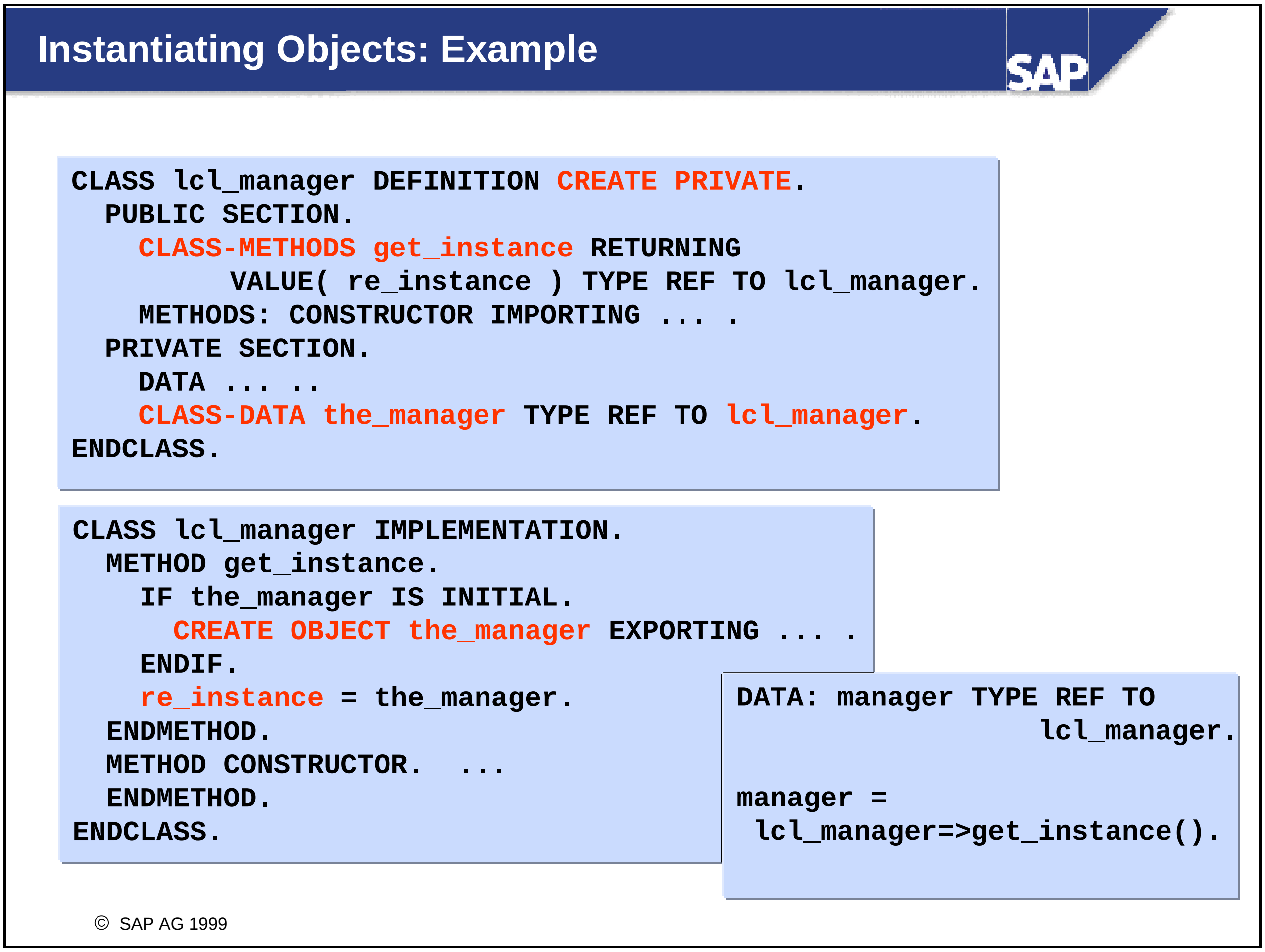
- CREATE PRIVATE
  - Objects can only be instantiated in the class itself
  - Instantiation using own (static) methods

```
CLASS <classname> DEFINITION
                   CREATE PRIVATE.
ENDCLASS.
```

CREATE PUBLIC, the optional default supplement, allows unrestricted instantiation of objects in a class, that is, instances in a class can be created in any part of this program/class.

CREATE PROTECTED only allows objects in a class to be instantiated in that class itself and in any of its subclasses.

CREATE PRIVATE only allows objects in a class to be instantiated in that class itself. This is then done using static methods (known as Factory Methods).



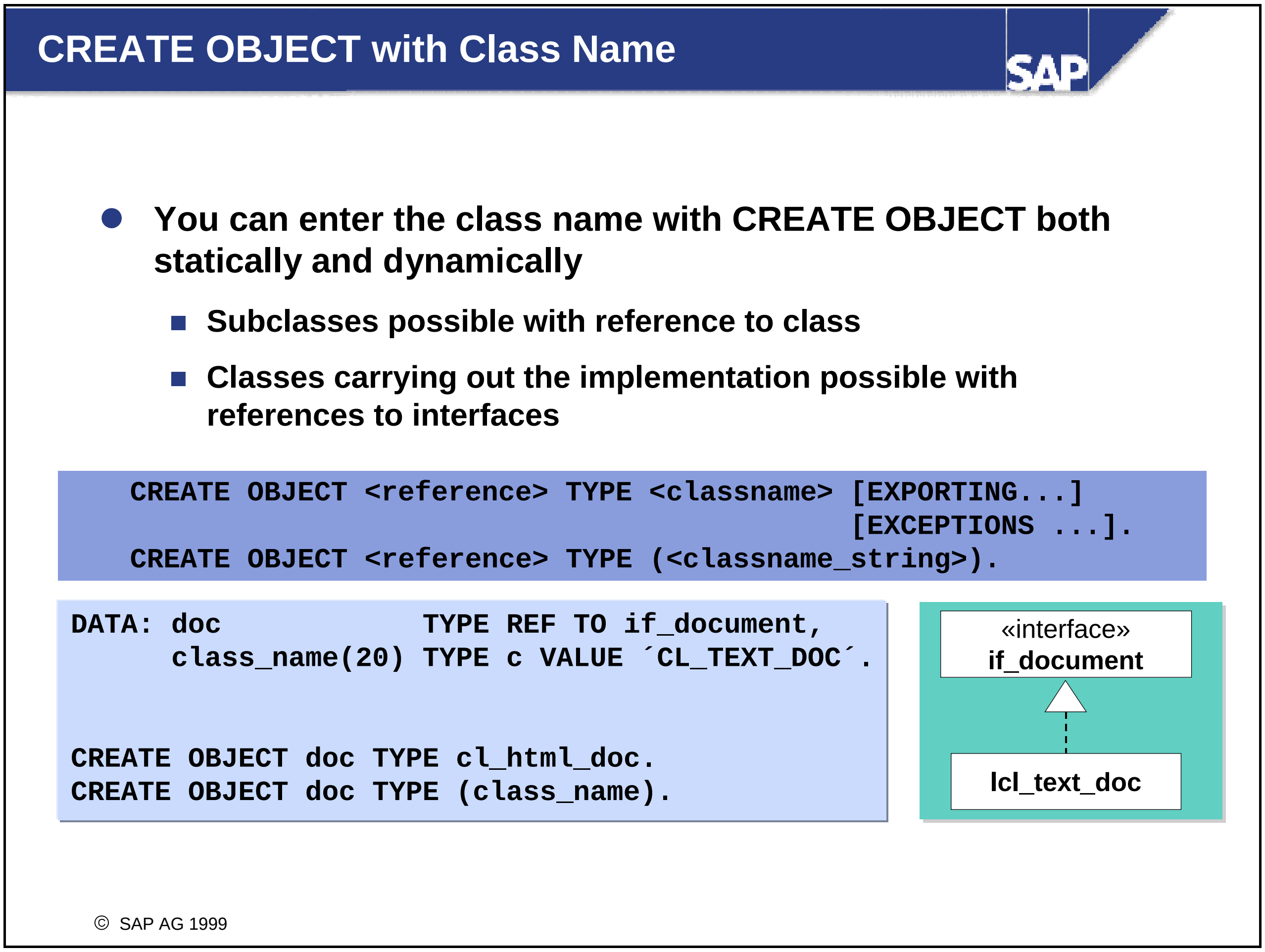
Instantiating Objects: Example

```
CLASS lcl_manager DEFINITION CREATE PRIVATE.
  PUBLIC SECTION.
    CLASS-METHODS get_instance RETURNING
          VALUE( re_instance ) TYPE REF TO lcl_manager.
    METHODS: CONSTRUCTOR IMPORTING ... .
  PRIVATE SECTION.
    DATA ... ..
    CLASS-DATA the_manager TYPE REF TO lcl_manager.
ENDCLASS.
```

```
CLASS lcl_manager IMPLEMENTATION.
  METHOD get_instance.
    IF the_manager IS INITIAL.
      CREATE OBJECT the_manager EXPORTING ... .
    ENDIF.
    re_instance = the_manager.
  ENDMETHOD.
  METHOD CONSTRUCTOR.  ...
  ENDMETHOD.
ENDCLASS.
```

```
DATA: manager TYPE REF TO
                  lcl_manager.

manager =
 lcl_manager=>get_instance().
```

Example using the addition CREATE PRIVATE (see above):
Class lcl_manager wants to prevent several objects of this class existing at runtime. Only one instance is to be instantiated.
Therefore the class defines the instantiation of objects as private and provides in its place the static method get_instance, which a potential client can use to get a reference to the sole object.



CREATE OBJECT with Class Name

- You can enter the class name with CREATE OBJECT both statically and dynamically
  - Subclasses possible with reference to class
  - Classes carrying out the implementation possible with references to interfaces

```
    CREATE OBJECT <reference> TYPE <classname> [EXPORTING...]
                                               [EXCEPTIONS ...].
    CREATE OBJECT <reference> TYPE (<classname_string>).
```

```
DATA: doc             TYPE REF TO if_document,
      class_name(20)  TYPE c VALUE ´CL_TEXT_DOC´.


CREATE OBJECT doc TYPE cl_html_doc.
CREATE OBJECT doc TYPE (class_name).
```

The CREATE OBJECT statement is extended by the introduction of inheritance and interfaces: you can enter the class of the instance to be created either statically, using the class name, or dynamically, using a variable containing the class name. Once the statement has been executed (successfully), a (runtime) instance of the class entered will have been created and the reference variable entered points to this instance.

There are two possible situations:

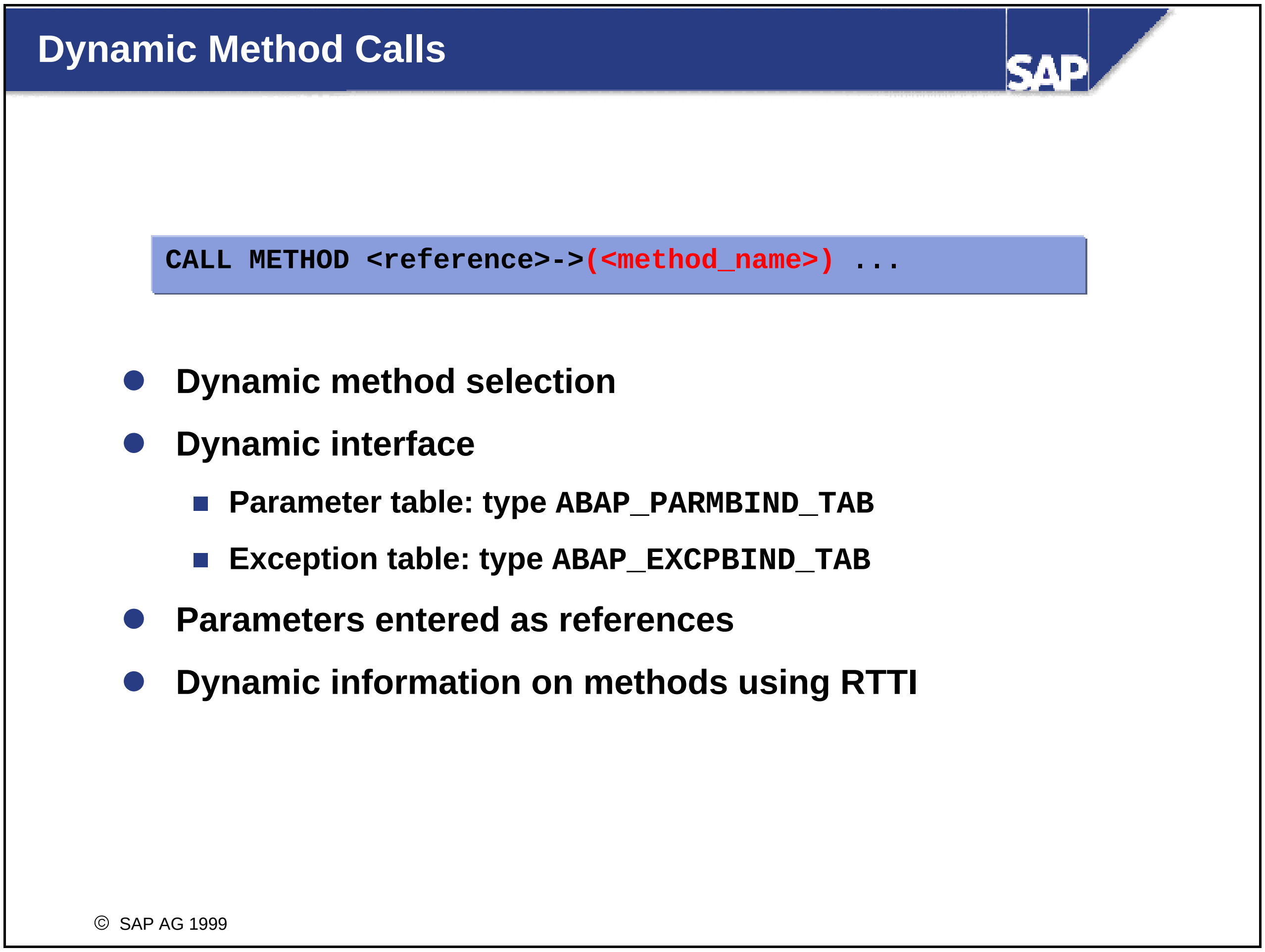
For a reference variable referring to a class, enter the name of a subclass (or of the class itself).

For a reference variable referring to an interface, enter the name of the class carrying out the implementation.

A check can be carried out in the static form “... TYPE <classname>…” to see if one of the two situations above has occurred. If it has not, a syntax error will occur.

In the dynamic form “...TYPE (<classname_string>).” the classname_string field provides the class name. A check can be carried out at runtime to ensure that the reference variable type is compatible with the class entered. If this is not the case, a runtime error occurs.

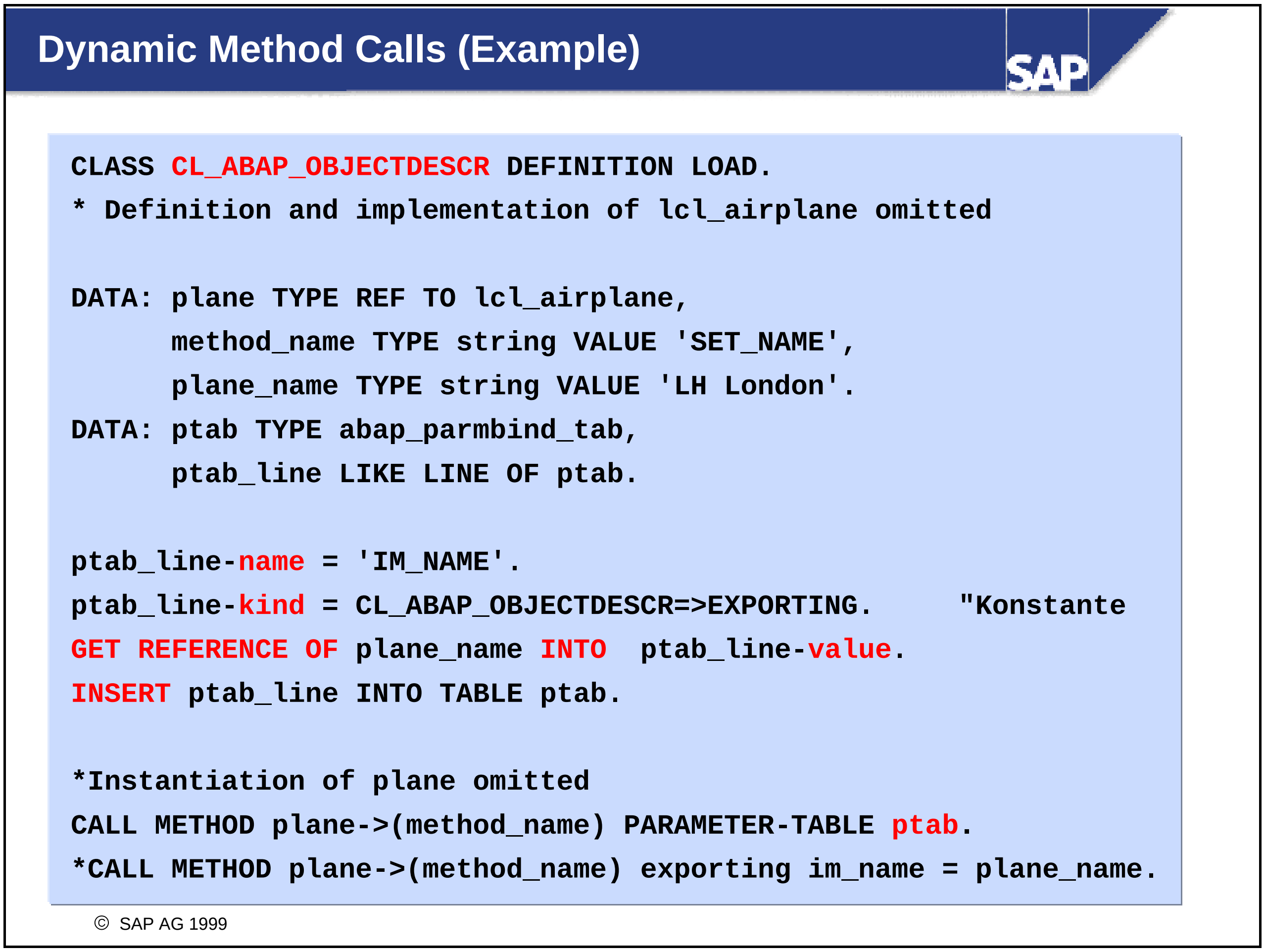
In the dynamic form, you can only enter the names of classes whose (instance) constructor either has no parameters or only optional parameters.



Dynamic Method Calls

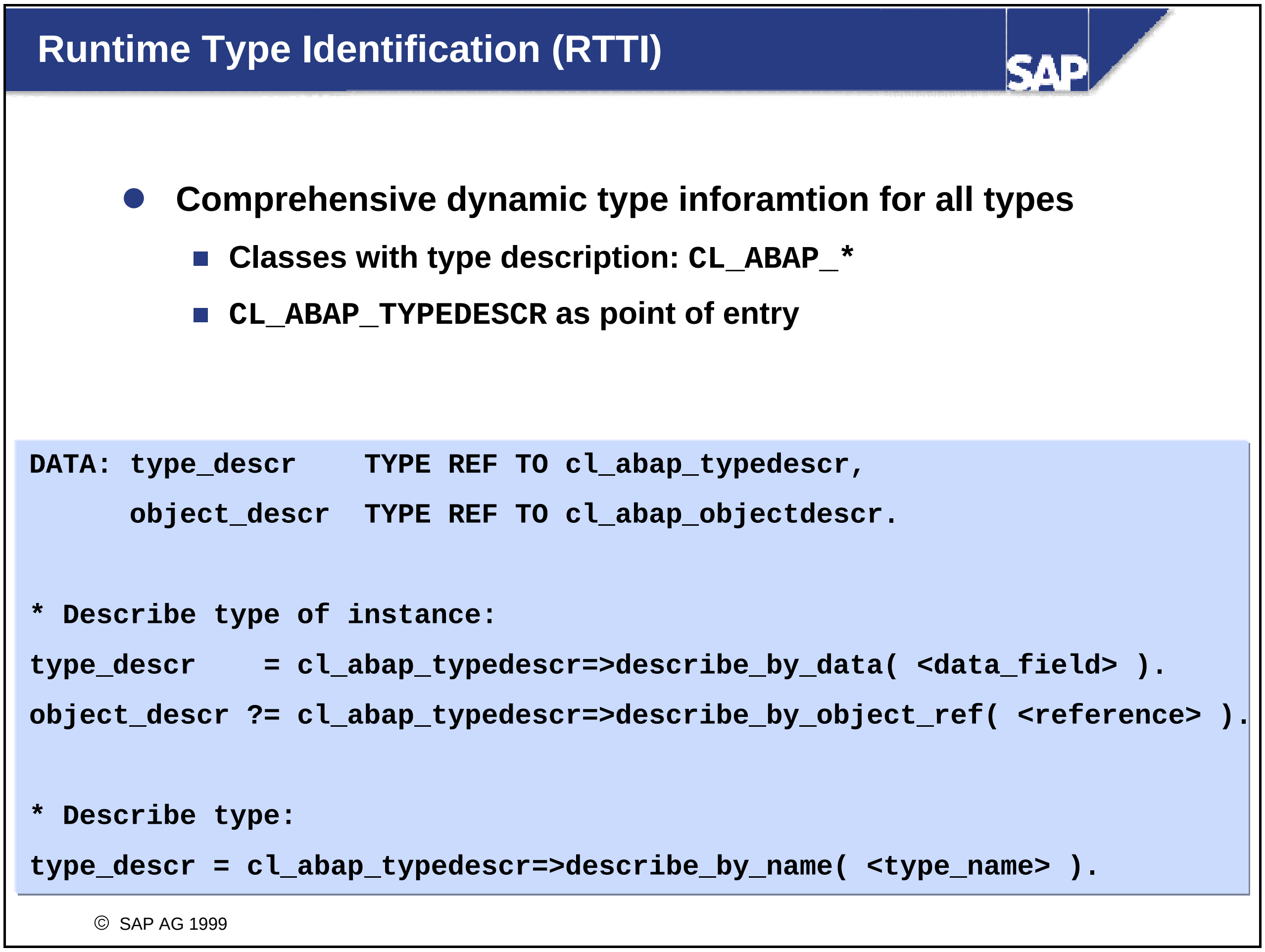
```
CALL METHOD <reference>->(<method_name>) ...
```

- Dynamic method selection
- Dynamic interface
  - Parameter table: type `ABAP_PARMBIND_TAB`
  - Exception table: type `ABAP_EXCPBIND_TAB`
- Parameters entered as references
- Dynamic information on methods using RTTI



Dynamic Method Calls (Example)

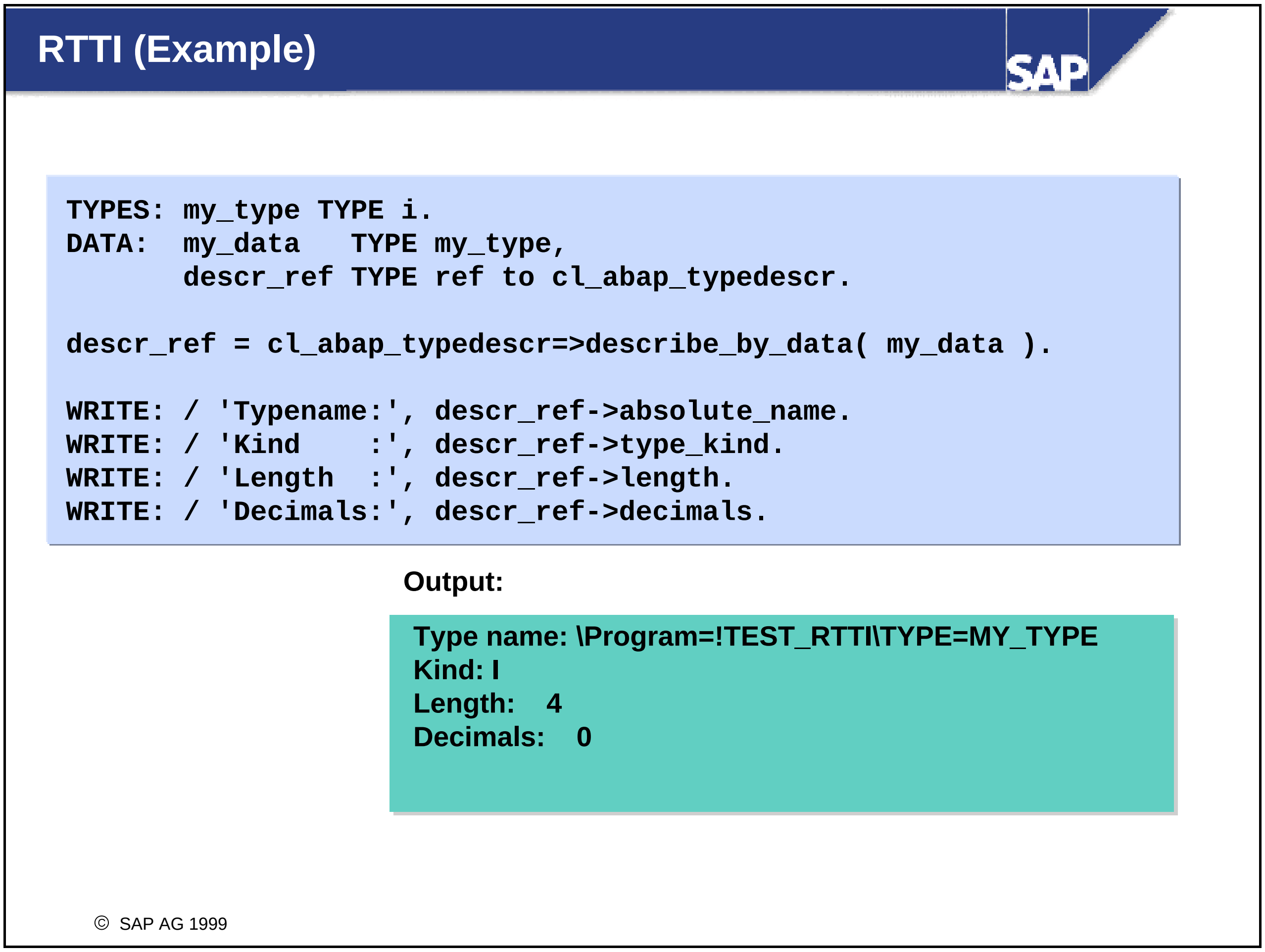
```
CLASS CL_ABAP_OBJECTDESCR DEFINITION LOAD.
* Definition and implementation of lcl_airplane omitted

DATA: plane TYPE REF TO lcl_airplane,
      method_name TYPE string VALUE 'SET_NAME',
      plane_name TYPE string VALUE 'LH London'.
DATA: ptab TYPE abap_parmbind_tab,
      ptab_line LIKE LINE OF ptab.

ptab_line-name = 'IM_NAME'.
ptab_line-kind = CL_ABAP_OBJECTDESCR=>EXPORTING.     "Konstante
GET REFERENCE OF plane_name INTO  ptab_line-value.
INSERT ptab_line INTO TABLE ptab.

*Instantiation of plane omitted
CALL METHOD plane->(method_name) PARAMETER-TABLE ptab.
*CALL METHOD plane->(method_name) exporting im_name = plane_name.
```



Runtime Type Identification (RTTI)

- Comprehensive dynamic type inforamtion for all types
  - Classes with type description: `CL_ABAP_*`
  - `CL_ABAP_TYPEDESCR` as point of entry

```
DATA: type_descr    TYPE REF TO cl_abap_typedescr,
      object_descr  TYPE REF TO cl_abap_objectdescr.

* Describe type of instance:
type_descr    = cl_abap_typedescr=>describe_by_data( <data_field> ).
object_descr ?= cl_abap_typedescr=>describe_by_object_ref( <reference> ).

* Describe type:
type_descr = cl_abap_typedescr=>describe_by_name( <type_name> ).
```

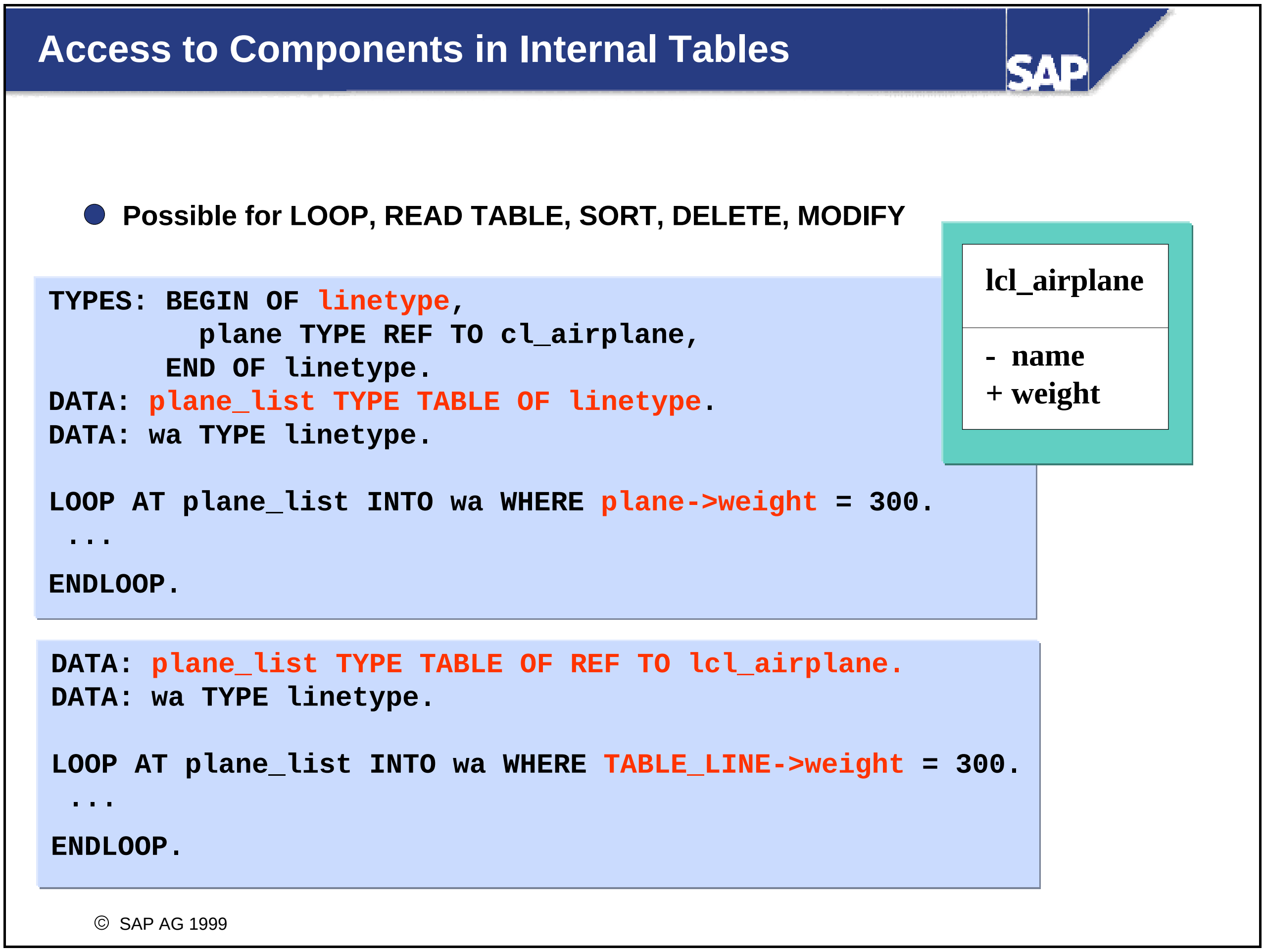


RTTI (Example)

```
TYPES: my_type TYPE i.
DATA:  my_data    TYPE my_type,
       descr_ref TYPE ref to cl_abap_typedescr.

descr_ref = cl_abap_typedescr=>describe_by_data( my_data ).

WRITE: / 'Typename:', descr_ref->absolute_name.
WRITE: / 'Kind    :', descr_ref->type_kind.
WRITE: / 'Length  :', descr_ref->length.
WRITE: / 'Decimals:', descr_ref->decimals.
```

Output:

```
Type name: \Program=!TEST_RTTI\TYPE=MY_TYPE
Kind: I
Length:   4
Decimals:   0
```



Access to Components in Internal Tables

- Possible for LOOP, READ TABLE, SORT, DELETE, MODIFY

```
TYPES: BEGIN OF linetype,
         plane TYPE REF TO cl_airplane,
       END OF linetype.
DATA: plane_list TYPE TABLE OF linetype.
DATA: wa TYPE linetype.

LOOP AT plane_list INTO wa WHERE plane->weight = 300.
 ...
ENDLOOP.
```

lcl_airplane

- name
+ weight

```
DATA: plane_list TYPE TABLE OF REF TO lcl_airplane.
DATA: wa TYPE linetype.

LOOP AT plane_list INTO wa WHERE TABLE_LINE->weight = 300.
 ...
ENDLOOP.
```

If the line type of an internal table contains references variables in the component comp, their attributes can accessed in the following statements:

LOOP AT itab ... WHERE comp->attr ...

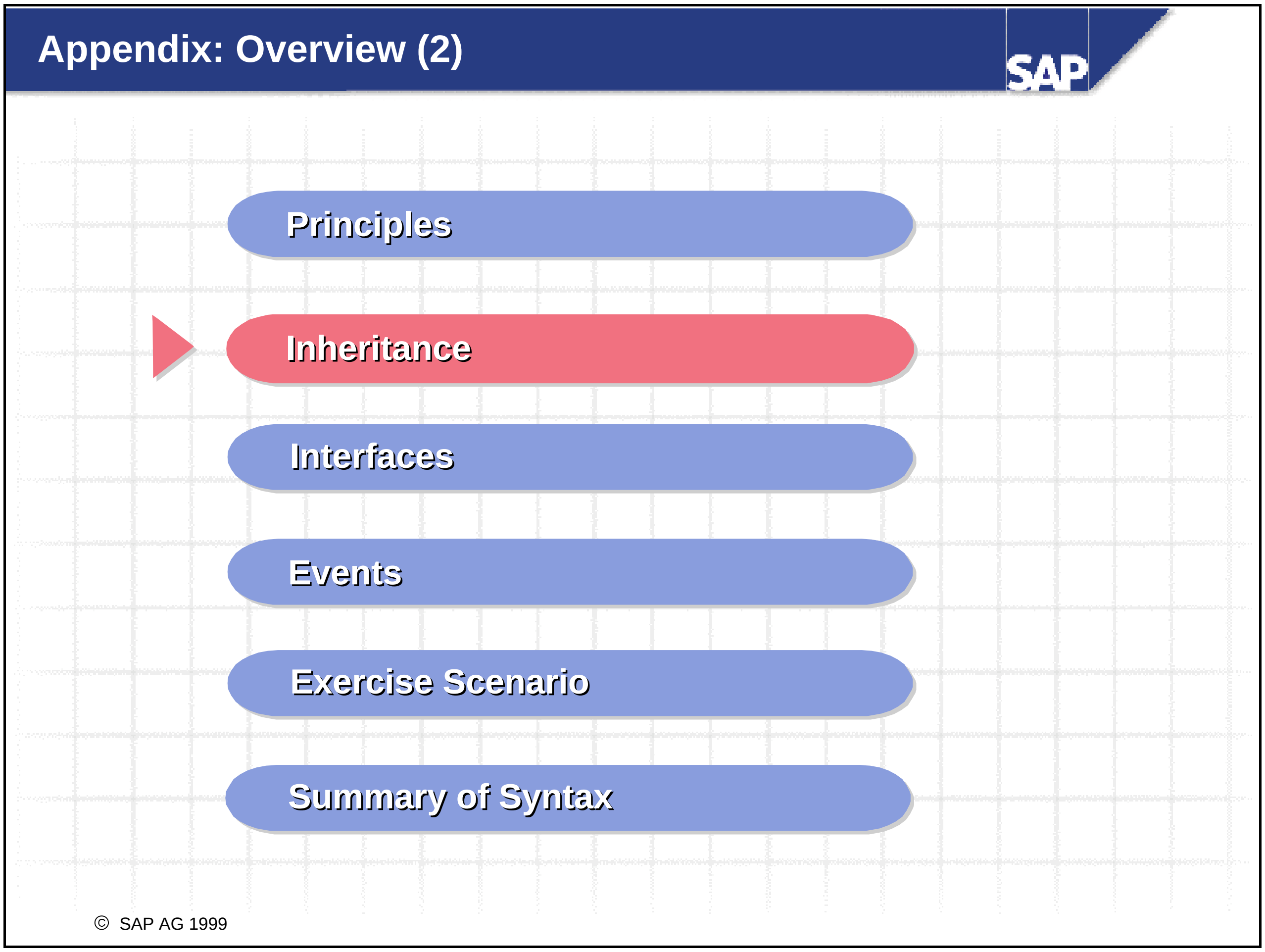
READ TABLE itab ... WITH [TABLE] KEY comp->attr ...

SORT itab BY comp->attr ...

DELETE itab WHERE comp->attr ...

MODIFY itab ... TRANSPORTING .. WHERE comp->attr ...

If an internal table has unstructured lines of the reference variable type, then the attributes of the object that the line points to can be addressed using TABLE_LINE->attr.

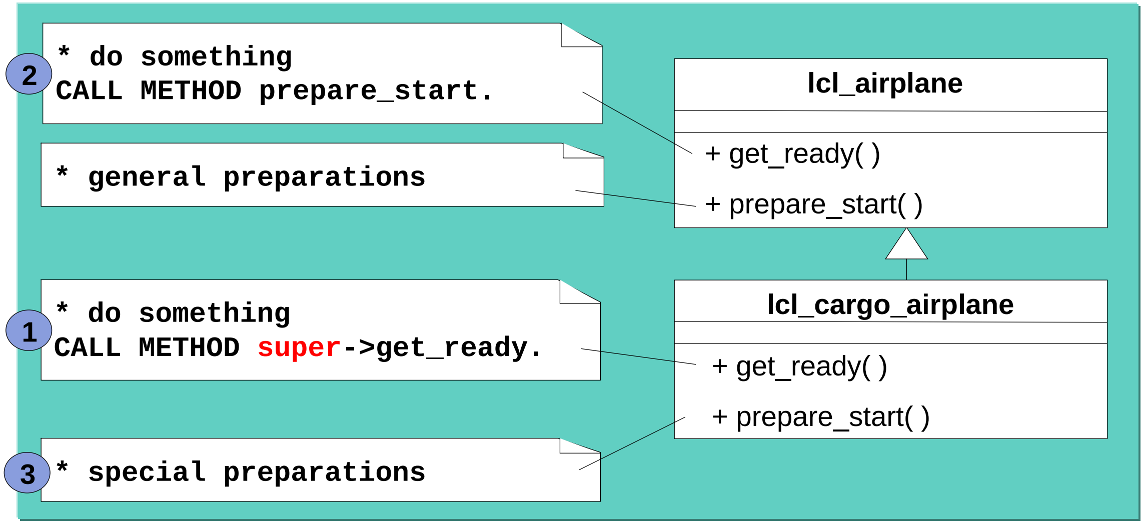


Appendix: Overview (2)

- Principles
- Inheritance
- Interfaces
- Events
- Exercise Scenario
- Summary of Syntax

Polymorphism in Methods

```
CALL METHOD cargo_airplane->get_ready.
```



```
* do something
CALL METHOD prepare_start.
```

```
* general preparations
```

```
* do something
CALL METHOD super->get_ready.
```

```
* special preparations
```

In ABAP Objects you can call a method from the superclass using the pseudo-reference **super**: CALL METHOD super->method_name ...

You can only do this in the implementation of the redefined method method_name in a subclass.

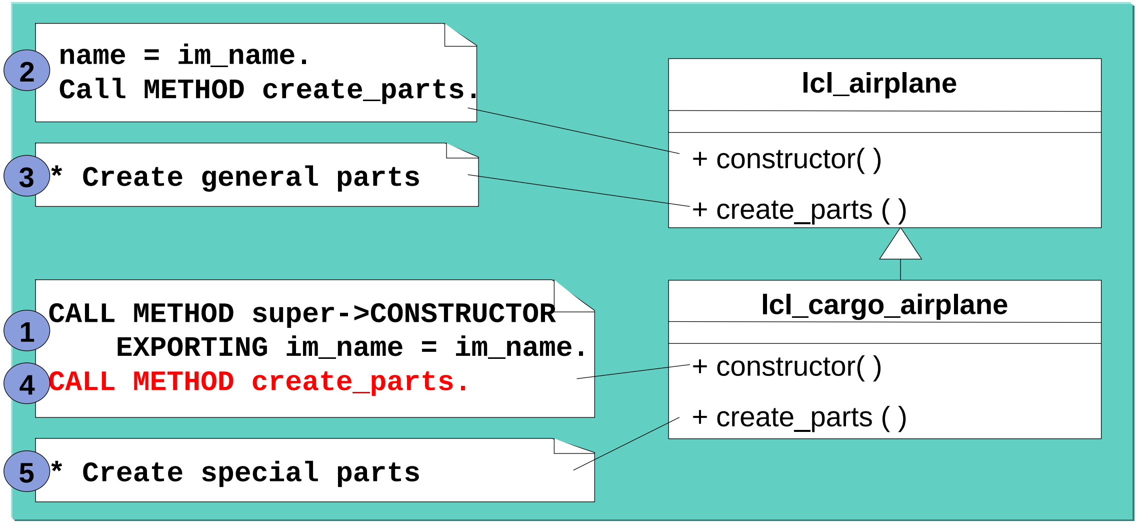
In the above example, the reference variable cargo_airplane calls the method get_ready, in which the superclass method get_ready is called. This calls the method prepare_start, which is redefined in the subclass. As the dynamic type of the calling reference variable is the subclass cl_cargo_airplane, the call is polymorphic, that is, the implementation of the subclass is carried out.

In the above example, the implementation of the superclass method prepare_start cannot be accessed from the superclass method get_ready.

Polymorphism in the (Instance) Constructor

- **Within the instance constructor, methods from the same class cannot be called polymorphically!**

```
CREATE OBJECT cargo_airplane EXPORTING ... .
```



```
name = im_name.
Call METHOD create_parts.
```

```
* Create general parts
```

```
CALL METHOD super->CONSTRUCTOR
     EXPORTING im_name = im_name.
CALL METHOD create_parts.
```

```
* Create special parts
```

What happens when a cargo plane instance is created in the above example? Firstly, the constructor of superclass lcl_airplane is called in the constructor, which in turn calls the method create_parts. As this method is (implicitly) carried out on an instance in the lcl_cargo_airplane class, the call was, as usual, polymorphic, that is, the implementation for class lcl_cargo_airplane was carried out. This causes a problem: an instance method is running for an object whose constructor is not yet finished!

Therefore ABAP Objects works according to a different model: within the (instance) constructor, (instance) methods from that class cannot be polymorphic! In other words: within the constructor, subclass method implementations are ignored.

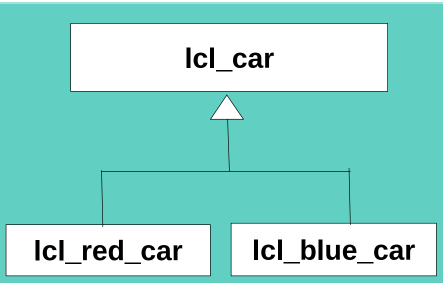
In the above situation, in which the constructor in the lcl_cargo_airplane class calls the superclass constructor, which in turn calls create_parts, the implementation in the superclass lcl_airplane is used. Therefore, once the superclass constructor has been executed, the create_parts method is called again in the subclass constructor, so that the specialized parts of the cargo plane are also created.

The consequences of this are: the create_parts method in the superclass must be able to cope with several calls for one and the same object. Why? Normally, the subclass method would call the method of the same name in the immediate superclass. However, during the constructor, the superclss implementation of create_parts is called twice.

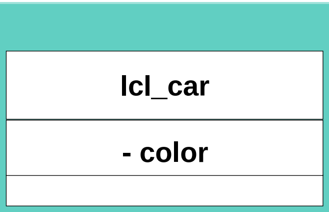
Incorrect Use of Inheritance: Example (1)

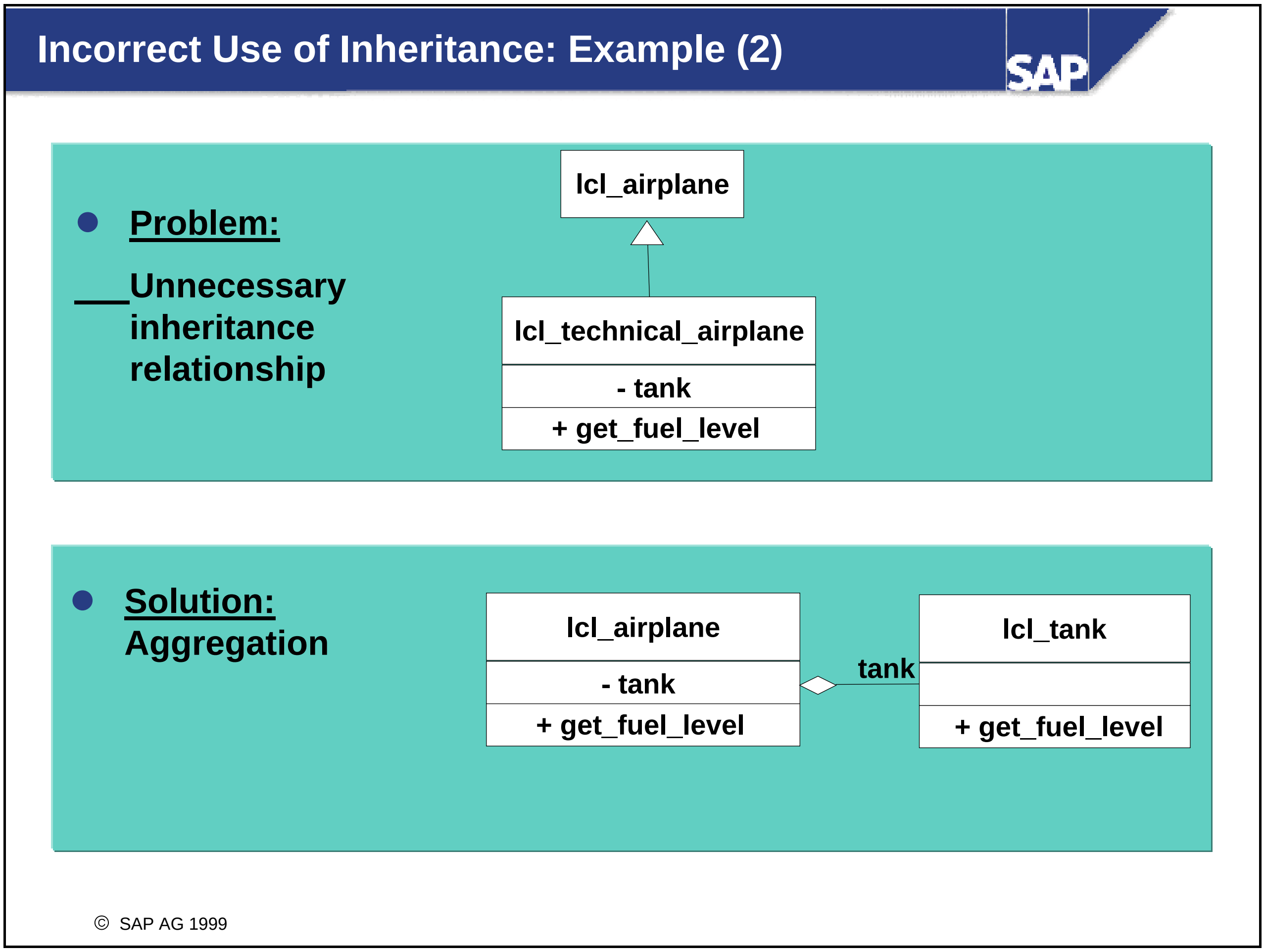
- **Problem:**

  Unnecessary inheritance relationship



- **Solution:** Additional attribute or aggregation





Incorrect Use of Inheritance: Example (2)

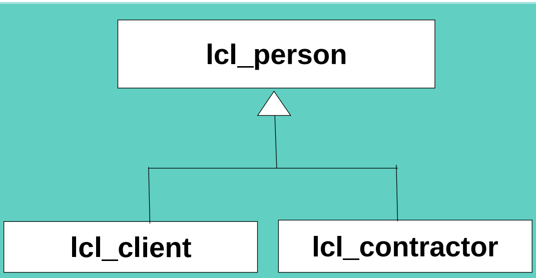
- **Problem:**

  Unnecessary inheritance relationship

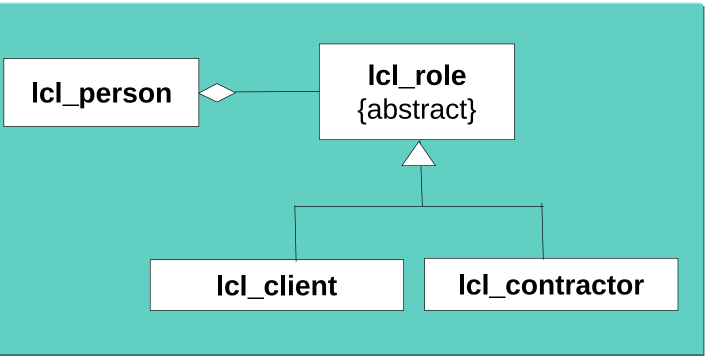
- **Solution:** Aggregation

Incorrect Use of Inheritance: Example (3)

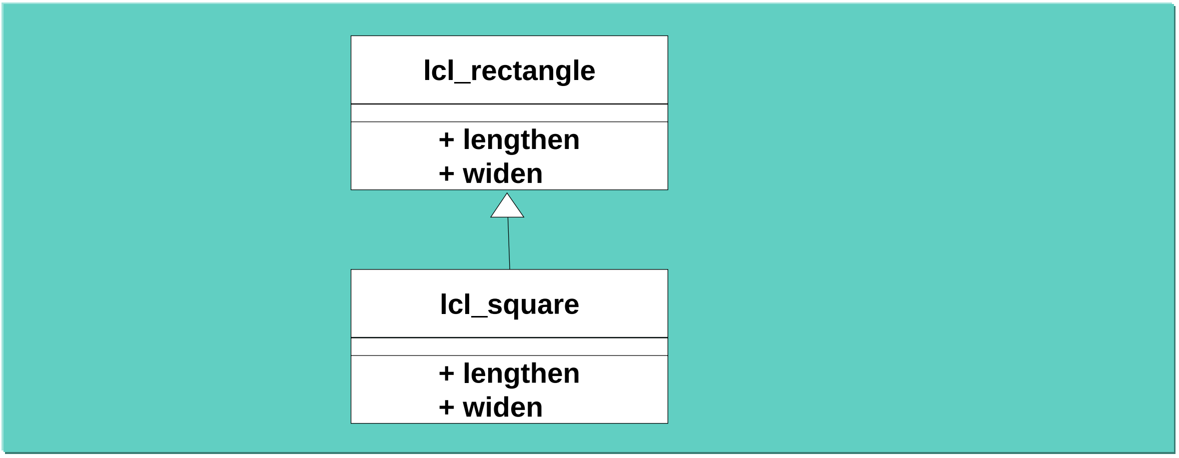
- **Problem:**
  One person can be both client and vendor at different times



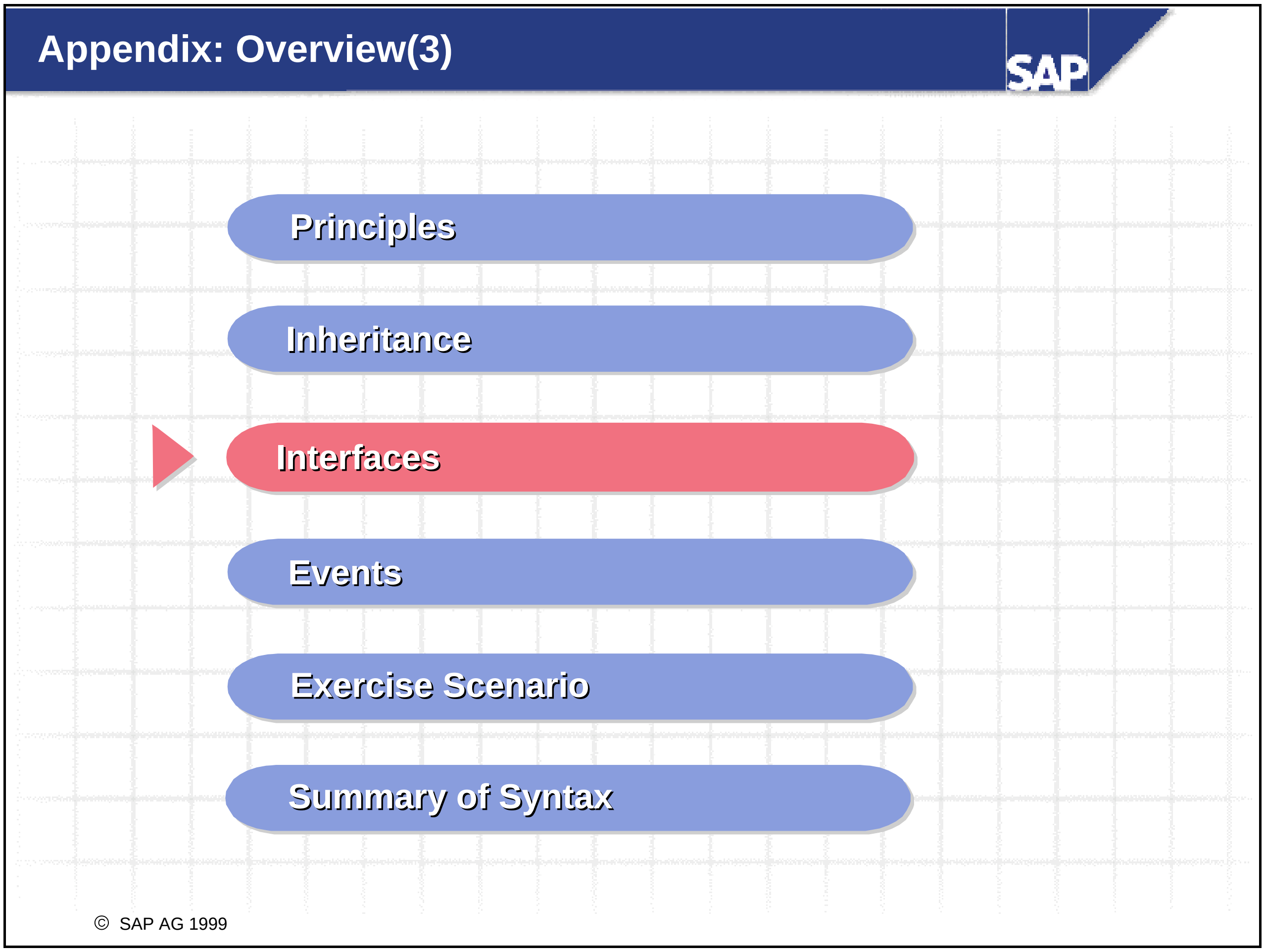
- **Solution:**
  Roles



Problematic Use of Inheritance: Example

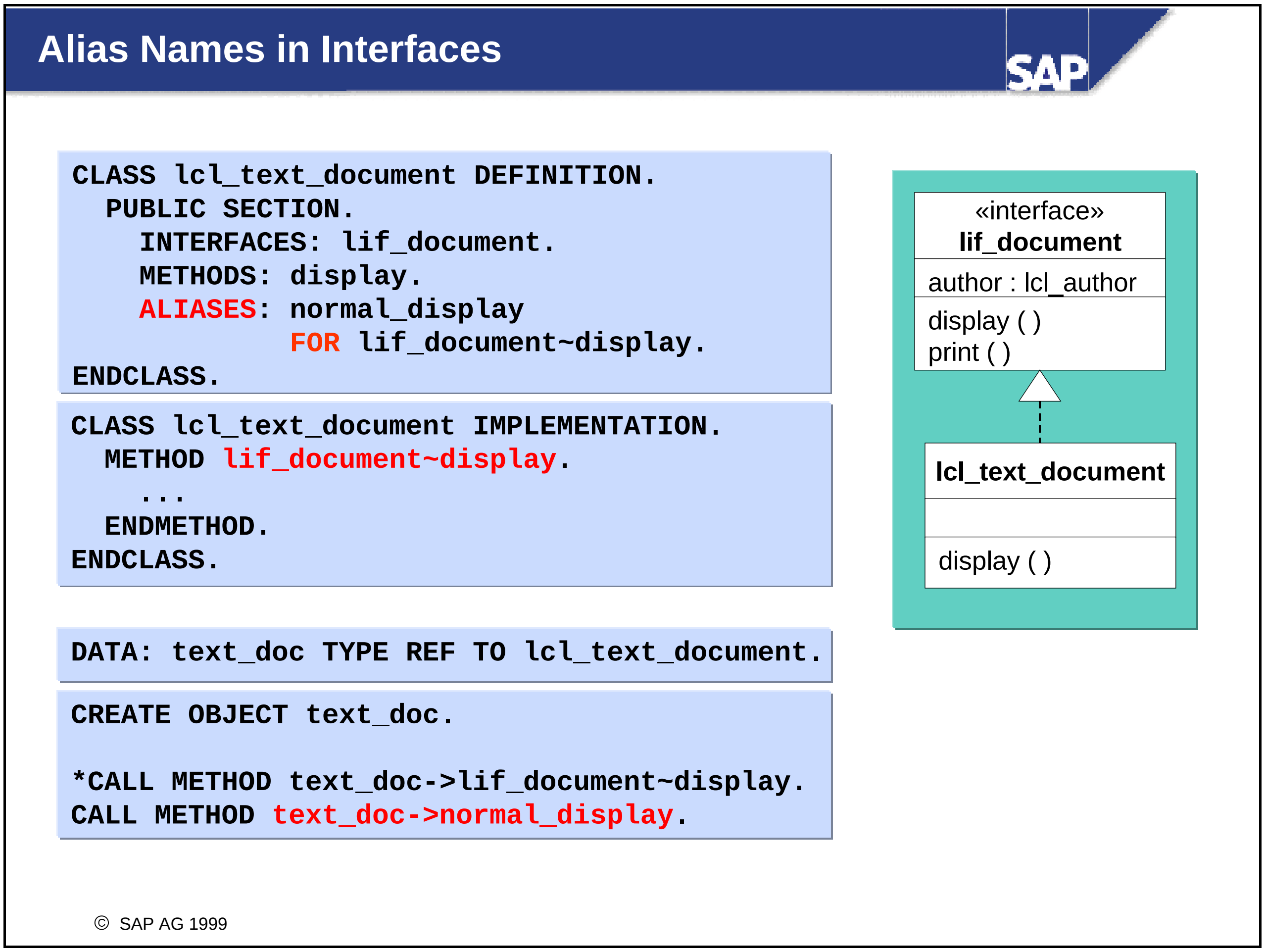


- **Problem:** **the inheritance relationship does not correspond to the real world: it does not make sense to have 2 methods to increase the side length of a square.**



Appendix: Overview(3)

- Principles
- Inheritance
- Interfaces
- Events
- Exercise Scenario
- Summary of Syntax



Alias Names in Interfaces

```
CLASS lcl_text_document DEFINITION.
  PUBLIC SECTION.
    INTERFACES: lif_document.
    METHODS: display.
    ALIASES: normal_display
              FOR lif_document~display.
ENDCLASS.
```

```
CLASS lcl_text_document IMPLEMENTATION.
  METHOD lif_document~display.
    ...
  ENDMETHOD.
ENDCLASS.
```

```
DATA: text_doc TYPE REF TO lcl_text_document.
```

```
CREATE OBJECT text_doc.

*CALL METHOD text_doc->lif_document~display.
CALL METHOD text_doc->normal_display.
```

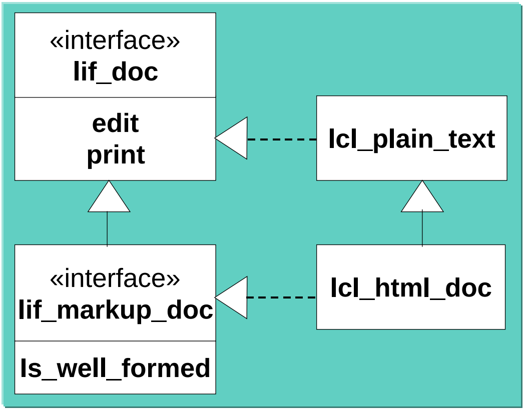
In the class carrying out the implementation, you can assign an **alias name** to an interface component using the ALIASES statement for the class itself and all its users. This is however only an abbreviation of the long name. Even if an alias is assigned for an interface method, the method only needs to be implemented once as <interfacename>~<methodname>. Alias names are subject to the usual visibility rules.

The most important use for alias names is in nested interfaces. In the definition of a nested interface, the components of the component interfaces cannot be addressed directly, but **only** using their alias names.

Alias names can be used in classes to enable class-specific components that have been replaced by components from interfaces during the course of the development cycle to continue to be addressed by their old names. This means that the users of the class to not need to be adjusted in accordance with the new names.

Alias names cannot be used in the IMPLEMENTATION part of the class itself.

Inheritance and Supported Interfaces



```
CLASS lcl_plain_text IMPLEMENTATION.
 METHOD lif_doc~edit.
 ENDMETHOD.
 METHOD lif_doc~print.
 ENDMETHOD.
ENDCLASS.
```

```
CLASS lcl_html_doc DEFINITION
      INHERITING FROM lcl_plain_text.
 PUBLIC SECTION.
  INTERFACES lif_markup_doc.
  METHODS lif_doc~print REDEFINITION.
ENDCLASS.
```

```
CLASS lcl_html_doc IMPLEMENTATION.
 METHOD lif_doc~print.
 ENDMETHOD.
 METHOD lif_markup_doc~is_well_formed.
 ENDMETHOD.
ENDCLASS.
```

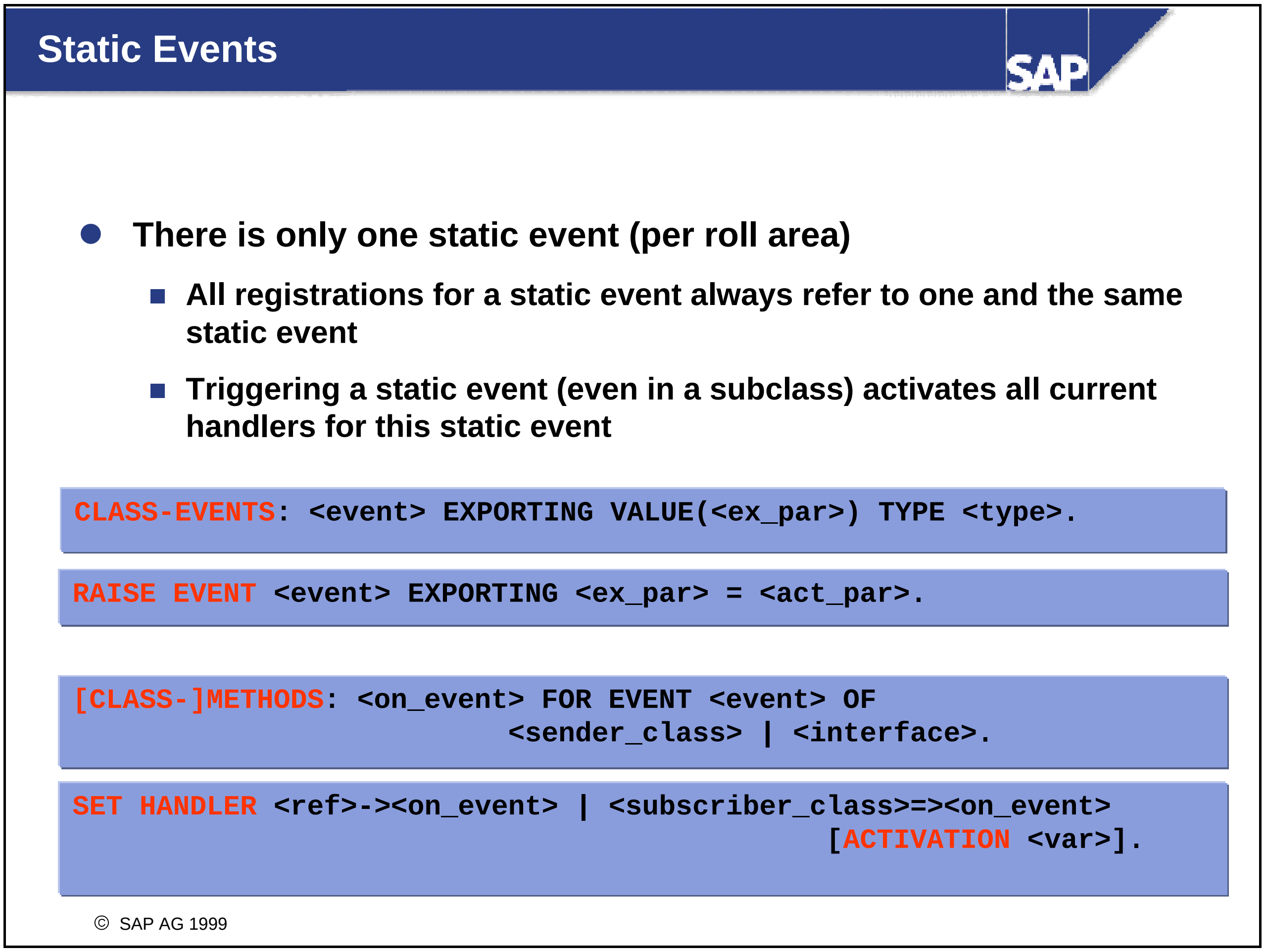
A subclass always co-inherits the supported interfaces from its superclass, but does have the options of implementing additional interfaces and redefining inherited interface methods.

If the subclass supports a compound interface, one of whose component interfaces is already implemented in the superclass, then the subclass does not need to do anything about the implementation of the component interface, but simply inherits its implementation (as long as there are no ABSTRACT constructs involved). In this case it would only need to implement the additional methods of the compound interface, although it could also redefine methods from the component interface.

The principle that interface components are only present once in any one class or interface is still valid. In the situation described above, it is therefore irrelevant, whether the subclass is supporting the interface method because it is implementing a compound interface, or because the superclass is implementing a component interface. The unique <interfacename>~<componentname> names of interface components ensure that interface components that are ‘inherited’ in a variety of ways can always be correctly identified and distinguished from one another.

Appendix: Overview(4)

- Principles
- Inheritance
- Interfaces
- Events
- Exercise Scenario
- Summary of Syntax



Static Events

- There is only one static event (per roll area)
  - All registrations for a static event always refer to one and the same static event
  - Triggering a static event (even in a subclass) activates all current handlers for this static event

```
CLASS-EVENTS: <event> EXPORTING VALUE(<ex_par>) TYPE <type>.
```

```
RAISE EVENT <event> EXPORTING <ex_par> = <act_par>.
```

```
[CLASS-]METHODS: <on_event> FOR EVENT <event> OF
                           <sender_class> | <interface>.
```

```
SET HANDLER <ref>-><on_event> | <subscriber_class>=><on_event>
                                         [ACTIVATION <var>].
```

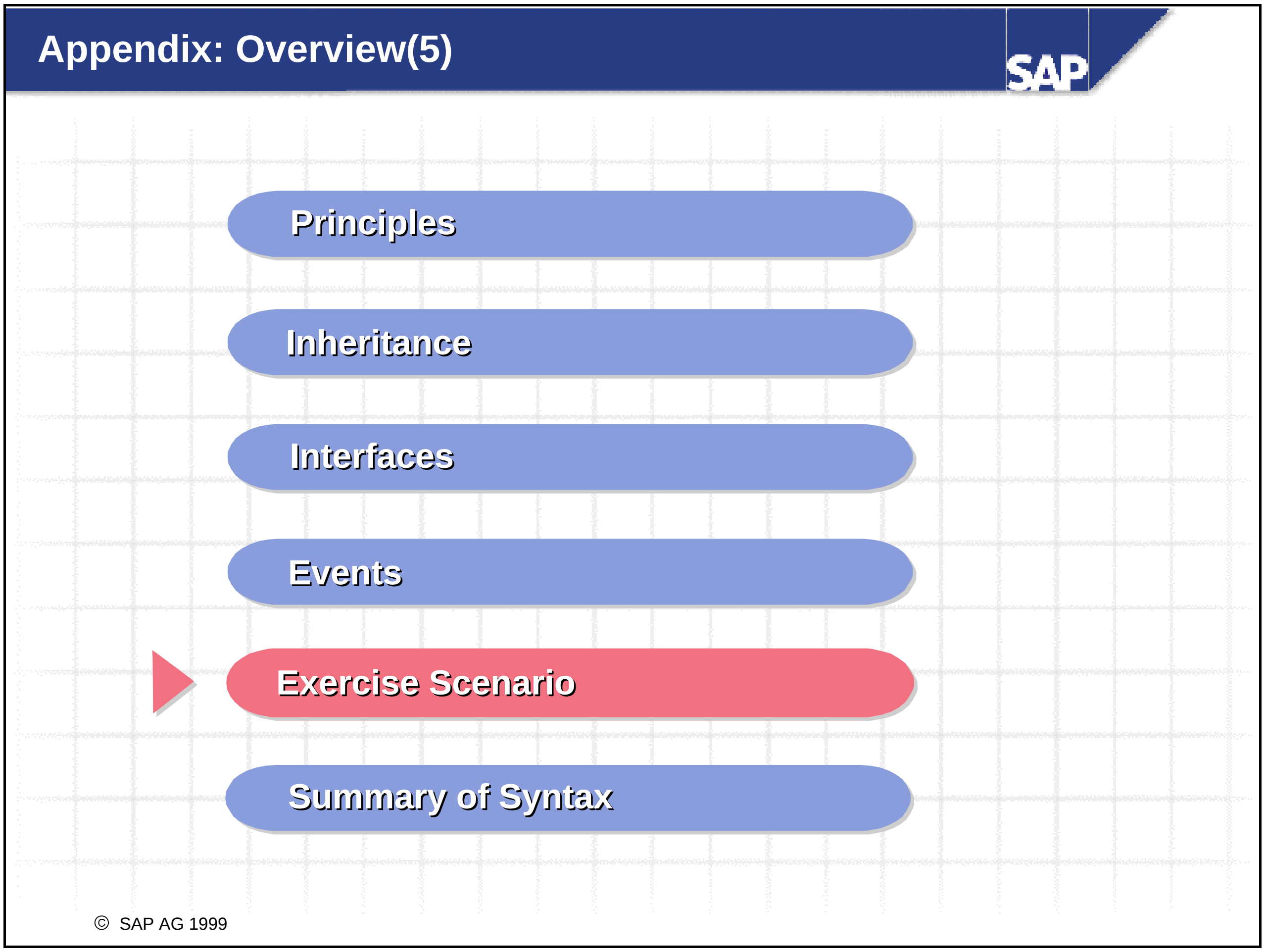
Static events can be triggered in instance methods and in static methods.

Classes or their instances that want to receive a message if an event is triggered and react to this event define event handler methods
Statement: (CLASS-)METHODS <handler_method> FOR EVENT <event> OF <classname>.

This class or its instances register themselves for one or more events at runtime.
Statement: SET HANDLER <handler_method>.

At runtime a class/instance can trigger a static event using the RAISE EVENT statement.

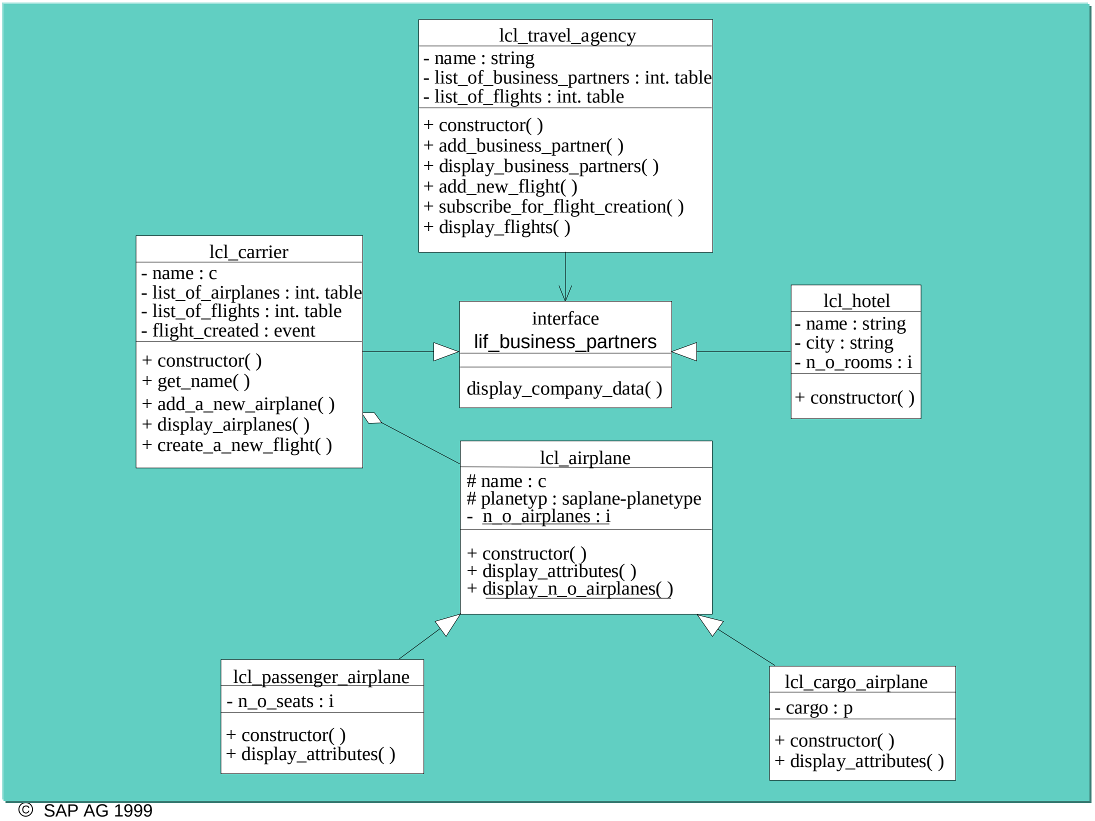
Static events, like attributes, only exist once per roll area. It is not the case that every subclass has its own copy of the static event.
All registrations for an event therefore refer to a single event, even if the event handler method registered is defined with reference to the inherited static event of a subclass.
Consequently, triggering a static event, be it in the defining class or in a subclass, activates all current handlers for this event, and not just those that are defined with reference to a specific class.



Appendix: Overview(5)

- Principles
- Inheritance
- Interfaces
- Events
- ▶ Exercise Scenario
- Summary of Syntax

Exercise Scenario for UML Notation



lcl_travel_agency
- name : string
- list_of_business_partners : int. table
- list_of_flights : int. table

+ constructor( )
+ add_business_partner( )
+ display_business_partners( )
+ add_new_flight( )
+ subscribe_for_flight_creation( )
+ display_flights( )

lcl_carrier
- name : c
- list_of_airplanes : int. table
- list_of_flights : int. table
- flight_created : event

+ constructor( )
+ get_name( )
+ add_a_new_airplane( )
+ display_airplanes( )
+ create_a_new_flight( )

interface
lif_business_partners

display_company_data( )

lcl_hotel
- name : string
- city : string
- n_o_rooms : i

+ constructor( )

lcl_airplane
# name : c
# planetyp : saplane-planetype
- n_o_airplanes : i

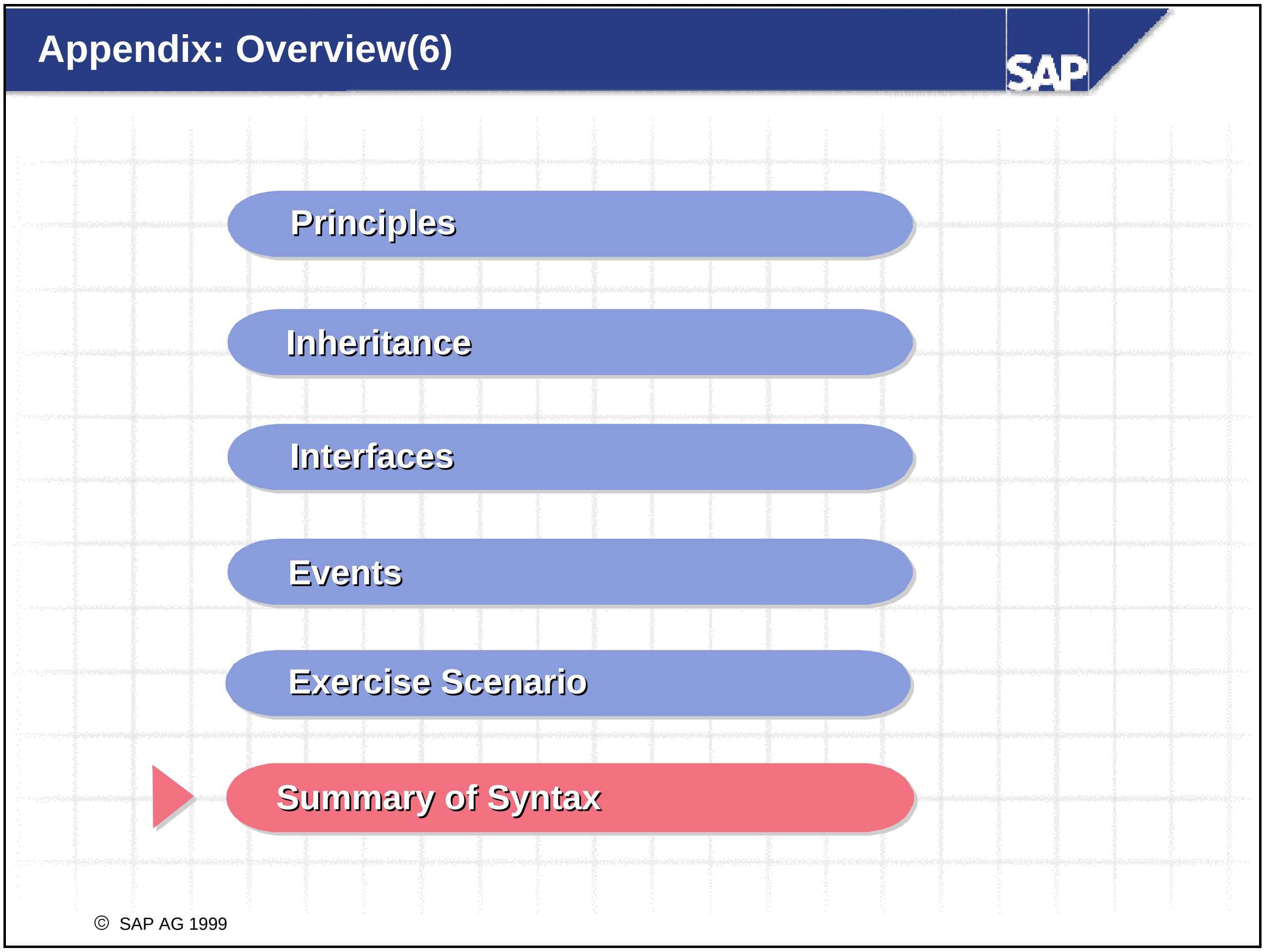
+ constructor( )
+ display_attributes( )
+ display_n_o_airplanes( )

lcl_passenger_airplane
- n_o_seats : i

+ constructor( )
+ display_attributes( )

lcl_cargo_airplane
- cargo : p

+ constructor( )
+ display_attributes( )



Appendix: Overview(6)

- Principles
- Inheritance
- Interfaces
- Events
- Exercise Scenario
- Summary of Syntax

ABAP Objects – Summary of Syntax

Availability: There is a brief comment on availability under each syntax summary. If there is no specific comment, then these components are available as of Rel. 4.5A.

Global class definition (in the CLASS-POOL):

```
CLASS-POOL.
  TYPES: …                                      " local types
  TYPE-POOLS: …                        " refer to type-pools (global types)

  CLASS c … .                          " local helper classes
    …
  ENDCLASS

  CLASS pc DEFINITION PUBLIC.  " the public class of the class pool
    …
  ENDCLASS.

  CLASS pc IMPLEMENTATION.
    …
  ENDCLASS.
```

Availability: all available as of 4.5A

Class definition:

```
CLASS c DEFINITION
  [PUBLIC]
  [ABSTRACT]
  [FINAL]
  [INHERITING FROM superclass]
  [CREATE {PUBLIC | PROTECTED | PRIVATE}].

[PUBLIC SECTION.
    … <definition of public components>]

[PROTECTED SECTION.
    … <definition of protected components>]

[PRIVATE SECTION.
    … <definition of private components>]

ENDCLASS.
```

```
CLASS c IMPLEMENTATION.
  …
ENDCLASS.
```

```
*--- forward definition of class for mutual recursive references
CLASS c DEFINITION DEFERRED.
```

Availability: all available as of 4.5A

Class components:

```
CLASS c DEFINITION.
{PUBLIC|PROTECTED|PRIVATE} SECTION.

  TYPES … .

  CONSTANTS … .

  [CLASS-]DATA    a TYPE t [READ-ONLY].

  METHODS m REDEFINITION.

  [CLASS-]METHODS m [ABSTRACT | FINAL]
                  [IMPORTING …] [EXPORTING …] [CHANGING …]
                               [RETURNING VALUE(result) TYPE t ] [EXCEPTIONS …] .

  [CLASS-]METHODS m [ ABSTRACT | FINAL]
                   FOR EVENT e OF {c|i}
                  [IMPORTING fp1 fp2 … fpn] .

  [CLASS-]EVENTS  e [EXPORTING … ].

  ALIASES alias FOR i~a.

  INTERFACES  i [VALUE a1 = v1 …].

ENDCLASS.
```

Availability:

as of 4.5A:	all components, except
as of 4.6A:	METHODS: REDEFINITION, ABSTRACT, FINAL

Implementation of classes/methods:

```
CLASS c IMPLEMENTATION.

  "--- implementation of methods for class, interfaces, events:
  METHOD m1.
    …
  ENDMETHOD.

ENDCLASS.
```

Interface definition :

```
INTERFACE i.

*  --> like public components of classes <--

ENDINTERFACE.
```

```
*--- forward definition of interface
INTERFACE i DEFERRED.

*---
INTERFACE i SUPPORTING REMOTE INVOCATION.
```

Availability:
as of 4.5A: all components, except
as of 4.6A: compound interfaces

Using OO:

Create objects:

```
CREATE OBJECT objvar  [ TYPE class | TYPE (classname) ]
                      [ EXPORTING arg = val … ].
```

Availability:
as of 4.5A: CREATE OBJECT … [EXPORTING …]
as of 4.6A: TYPE …

Call methods:

```
CALL METHOD o->m [EXPORTING …] [IMPORTING …] [CHANGING …]
                 [RECEIVING …] [EXCEPTIONS …].
CALL METHOD o->m( … [IMPORTING …] [CHANGING …][EXCEPTIONS …]).
a = o->m( [IMPORTING …] ).          "functional method call
a = o->m( [arg =] val ).            "ditto, with only one parameter
```

Availability:
as of 4.5A: CALL METHOD …
as of 4.6A: functional method call

Examples of functional method calls:

```
IF o->m( 4711 ) = TRUE. … ENDIF.
CASE order->status( ). WHEN … WHEN … ENDCASE.
LOOP AT itab WHERE name = o->hostname( ). … ENDLOOP.
len = strlen( adr1->get_name( ) ) + strlen( adr2->get_name ) ).
MOVE o->m( p1 = val1 p2 = val2) TO dest.
```

Trigger event:

```
RAISE EVENT e   [EXPORTING arg = val … ].
RAISE EVENT I~e [EXPORTING arg = val … ].
```

Activate/deactivate event handler:

```
*--- general form
SET HANDLER h1 h2 …  FOR … <see below> …  [ACTIVATION cval].

*--- standard registration
SET HANDLER h1 …  FOR i|o.       "- register instance event for instance
SET HANDLER h1 … .               "- handler for class event

*--- group registration
SET HANDLER h1 … FOR ALL INSTANCES.
                "- for "e of C" event: all instances of C
                "- for "e OF I" event: all instances of all C implementing I
```

Availability:
as of 4.5A: all components

Widening Cast:

```
"--- CAST
i1 ?= o1.           "--- does o1 implement interface of i1?
MOVE o1 ?TO i1.     "--- same as above
```

Special compiler statements:

```
*--- make class definition known to compiler before accessing class comp.
CLASS ddic_class DEFINITION LOAD.

*--- make interface known to compiler before accessing interface comp.
INTERFACE LOAD.
```

Component selection:

Symbol		**Meaning**
->	o->a i->a	Component access using object or interface reference
-	x-a	Component access for structures (and embedded objects)
~	I~a	Compound names for interface components
=>	c=>a	Access to the static components of class c