Foundation of Object Orientation

Course duration 3 days

1. Introduction

1.1. Explaining the Object-Oriented Model

Goal: Describe the difference between the procedural and object oriented programming models.

Background: ABAP is a hybrid programming language that supports both a procedural and an object-oriented programming model.

Definition: ***Procedural programming model***
The procedural programming model is based on the modularization of programs in classic processing blocks (event blocks, dialog modules, function modules, and subroutines).

Definition: ***ABAP Objects***
In ABAP Objects, the class conceptually supersedes the classic program, and modularization is implemented using methods. From a technical point of view, classes are still declared and implemented in programs.

Rule: Use ABAP objects wherever possible for new and further developments.

Difference: ***Procedural Approach vs. Object-Oriented Approach***

FEATURES	PROCEDURAL APPROACH	OBJECT-ORIENTED APPROACH
STRCUTURE	In procedural programming in SAP ABAP, the code is organized around procedures and functions.	OOP in SAP ABAP is organized around classes and objects. Classes define both data (attributes) and behavior (methods), and objects are instances of classes.
CODE REUSABILITY	In procedural programming in SAP ABAP, codes are less reusable.	OOP allows for inheritance, where new classes can be derived from existing classes, inheriting their attributes and methods. This leads to code reuse and the creation of specialized classes.

FUNCTION NAME	In procedural programming in SAP ABAP, we cannot have more than one function with same name.	OOP Polymorphism allows different classes to have methods with the same name but different behaviors.
MAINTENANCE	In procedural Programming Language in SAP ABAP, maintenance of codes can be more challenging.	OOP can make maintenance easier because classes are self-contained and changes to one class are less likely to affect other parts of the codebase. It leads to modular and reusable code.
REAL WORLD	Procedural programming in SAP ABAP is based on the unreal world.	Object-oriented approach is based on the real world.

Benefits: Using object-oriented ABAP, developers may represent real-world things such as clients, commodities, and orders as objects (Real-World Entity Modeling). Therefore, the code is more understandable and aligned with the business domain.

Benefits: Encapsulation ensures that information stored within objects may only be accessed and altered through certain methods. Consequently, there is less likelihood of unintentional data change and better data security.

Benefits: By establishing classes and using inheritance, programmers may reuse code in several regions of an application (*Code Reusability*). Therefore, redundancy is reduced, and the development process is made more efficient.

Background: ABAP is a hybrid programming language that supports both a procedural and an object-oriented programming model.

1.2. Analyzing and Designing with Unified Modeling Language (UML)

Goal 1: Classify objects.
Goal 2: Understanding the UML syntax.

Definition: ***Unified modeling language UML***
The unified modeling language (UML) is a general-purpose visual modeling language that is intended to provide a standard way to visualize the design of a system.

Note: UML provides a standard notation for many types of diagrams which can be roughly divided into three main

groups: behavior diagrams, interaction diagrams, and structure diagrams.

Note: We will use a form of Structural Diagram Types: class diagrams.

Definition: ***Class diagram***
In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

Key Points: ***Class***
A class is a blueprint for an object. It specifies what properties (attributes) and behaviors (methods) an object will have. Classes serve as the foundation for creating objects.

Attributes
Attributes represent properties of a class. They are depicted in the second partition of the class and correspond to member variables in code.

Operations (Methods)
Operations are the services a class offers. They appear in the third partition of the class and correspond to methods in code.

Visibility
Symbols like +, -, and # preceding an attribute or operation indicate visibility. + denotes public, - denotes private, and # denotes protected.

Parameters
Each parameter of an operation can be marked as in, out, or in out to indicate its direction with respect to the caller.

Note: We will use class diagrams according to UML notation in the following learning units. These examples will be used to learn and demonstrate the required notation.

1.3. ABAP 7.5x and Clean ABAP

Goal 1: A brief comparison of old and new syntax.
Goal 2: Understand why we use Clean ABAP.

Note: The following is a brief overview of the language changes in ABAP 7.4 and ABAP 7.5, which do not affect the course objectives. However, they are required to understand the examples.

Note: This overview does not claim to be exhaustive. It only serves to make the examples easier to understand.

Difference: **Before 7.4 vs "new" ABAP**

	BEFORE 7.4	**NEW**

DATA	`DATA text TYPE string.`
	
	`text = 'ABC'.`
		`DATA(text) = 'ABC'.`

LOOP AT	`DATA wa LIKE LINE OF itab.`
	
	`LOOP AT itab INTO wa.`
	`…`
	`ENDLOOP.`
		`LOOP AT itab INTO DATA(wa).`
		`…`
		`ENDLOOP.`

LOOP AT ASSIGNING	`FIELD-SYMBOLS: <line> TYPE …`
	
	`LOOP AT itab ASSIGNING <line>.`
	`…`
	`ENDLOOP.`
		`LOOP AT itab ASSIGNING FIELD-SYMBOL(<line>).`
		`…`
		`ENDLOOP.`

Note: For consistency with the general pattern of object oriented ABAP using references you may wish to use references as loop targets whenever possible.

Additionally, data access via field symbols is slightly faster than data access via references. This is only noticable when loops make up a significant part of the runtime of the program and is often not relevant, e.g. when database operations or other input/output processes dominate the runtime.

For these reasons, there are two possible consistent styles depending on the specific application context:

If the context mostly uses objects and references otherwise, and if small performance hits in loops are not generally relevant, use references instead of field symbols as loop targets whenever possible.

If the context performs a lot of manipulation of plain data and not references or objects, or if small performance hits in

loops are generally relevant, use field symbols to read and manipulate data in loops.

Differences: **Before 7.4 vs "new" ABAP**

	BEFORE 7.4	NEW
READ TABLE INDEX	`READ TABLE itab INDEX idx INTO wa.`	`wa = itab[ idx ].`
READ TABLE USING KEY	`READ TABLE itab INDEX idx USING KEY key INTO wa.`	`wa = itab[ KEY key INDEX idx ].`
READ TABLE WITH KEY	`READ TABLE itab WITH KEY col1 = … col2 = … INTO wa.`	`wa = itab[ col1 = … col2 = … ].`
READ TABLE WITH KEY COMPONENTS	`READ TABLE itab WITH TABLE KEY key COMPONENTS col1 = … col2 = … INTO wa.`	`wa = itab[ KEY key col1 = … col2 = … ].`

DOES RECORD EXIST?	`READ TABLE itab … TRANSPORTING NO FIELDS.`
	
	`IF sy-subrc = 0.`
	`...`
	`ENDIF.`
		`IF line_exists( itab[ … ] ).`
		`…`
		`ENDIF.`

GET TABLE INDEX	`DATA idx TYPE sy-tabix.`
	
	`READ TABLE … TRANSPORTING NO FIELDS.`
	
	`idx = sy-tabix.`
		`DATA(idx) = line_index( itab[ … ] ).`

Rule: ***Avoid unnecessary table reads***
In case you expect a row to be there, read once and react to the exception, use try … catch cx_sy_itab_line_not_found instead of littering and slowing down the main control flow with a double read "IF NOT line_exists(my_table[key = input])".

But if no row is expected, line_exists( ) can be more performant.

Note: The new table expressions always take the fastest option based on the table type.

Language elements:

Conversion Operator CONV
CONV dtype|#(...)

Value Operator VALUE
VALUE dtype|#((...) (...) ...) ...

For operator
FOR wa|<fs> IN itab [INDEX INTO idx] [cond]

Conditional operator COND
... COND dtype|#(WHEN log_exp1 THEN result1
[WHEN log_exp2 THEN result2]
...
[ELSE resultn]) ...

Conditional operator SWITCH
... SWITCH dtype|#(operand
WHEN const1 THEN result1
[WHEN const2 THEN result2]
...
[ELSE resultn]) ...

Corresponding Operator
... CORRESPONDING type([BASE (base)] struct|itab [mapping|except])

Definition:

Clean ABAP
Clean ABAP refers to the application of Clean Code principles in ABAP development. It aims to create readable, maintainable, and extensible ABAP code. The Clean ABAP Guideline standardizes coding style and defines what modern ABAP code should look like. The goal is not only to produce functional code but also to ensure it is easy to understand and maintain.

Link: Styleguide Clean ABAP

2. Fundamental Object-Oriented Syntax

2.1. Creating Local Classes

Goal 1: Define local classes.
Goal 2: Define attributes.
Goal 3: Create methods.

Definition:

Local classes
Local classes and interfaces can only be used in the program in which they are defined.

Code: ***Structure of a local class***

```
REPORT <report_name>.
"Defintion Part
"Subdivided into up to three visibility sections
CLASS <class_name> DEFINITION.
   "Components of class
   PUBLIC SECTION.

   PROTECTED SECTION.

   PRIVATE SECTION.
ENDCLASS.

"Implementation part
CLASS <class_name> IMPLEMENTATION.
…
ENDCLASS.
```

Definition: ***Definition Part***
The definition part of a class contains the definition and declaration of all of the elements in the class, that is, the types, the constants, the attributes, and the methods. It begins with CLASS <class_name> DEFINITION. and ends with ENDCLASS.

Definition: ***Implementation Part***
The implementation part of a class contains the executable code of the class, namely the implementation of its methods. It begins with CLASS <class_name> IMPLEMENTATION. and ends with ENDCLASS. The implementation part of a class is optional. It becomes mandatory as soon as the class definition contains executable methods.

Definition: ***Public Section***
All components declared in the public visibility section defined using PUBLIC SECTION are accessible to all consumers as well as in the methods of all inheritors and the class itself. The public components of the class form the interface between the class and its consumers.

Definition: ***Protected Section***
All components declared in the protected visibility section defined with PROTECTED SECTION are accessible in the methods of all inheritors and in the class itself. Protected

components form a special interface between a class and its subclasses.

Definition: ***Private Section***
All components declared in the private visibility section defined with PRIVATE SECTION are only accessible in the class itself and are also not visible to the inheritors. The private components therefore do not form an interface to the consumers of the class.

Summary: ***Visibility of a class***

VISIBLE FOR	PUBLIC SECTION	PROTECTED SECTION	PRIVATE SECTION
SAME CLASS AND ITS FRIENDS	X	X	X
ANY SUBCLASSES	X	X	-
ANY REPOSITORY OBJECTS	X	-	-

Rule: Use for classes that require only local relevance.

Code: Local class attributes declaration

```
REPORT <report_name>.
"Defintion Part
"Subdivided into up to three visibility sections
CLASS <class_name> DEFINITION.
   "Components of class
   PUBLIC SECTION.

      TYPES: …
      CONSTANTS: …

      DATA: <data_name> TYPE <type>,
            <data_name> TYPE <type> READ-ONLY.

      CLASS-DATA <class_data_name> type <typ>.

   PROTECTED SECTION.

   PRIVATE SECTION.
ENDCLASS.

"Implementation part
CLASS <class_name> IMPLEMENTATION.
```

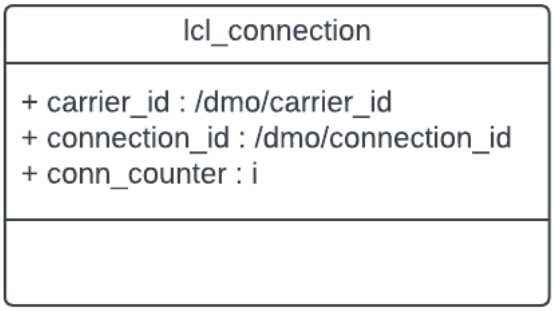
```
ENDCLASS.
```

Definition: ***Data statement***
The contents of instance attributes define the instance-specific state of an object. You declare them using the DATA statement.

Definition: ***Class data***
The contents of static attributes define the state of the class that is valid for all instances of the class. Static attributes exist once for each class. You declare them using the CLASS-DATA statement. They are accessible for the entire runtime of the class.

Task 1: ***Step 1***
Create report Z*CONNECTIONS.

Step 2
Declare the following attributes:



Solution: The solution to the task can be found in a separate document.

2.2 Creating objects

Goal 1: Create objects.
Goal 2: Understand the memory strategy for references.
Goal 3: Call instance and static attributes.

Code: ***Create a class object with new #( )***

```
REPORT <report_name>.
"Defintion Part
"Subdivided into up to three visibility sections
CLASS <class_name> DEFINITION.
   "Components of class
   PUBLIC SECTION.

      TYPES: …
      CONSTANTS: …

      DATA: <data_name> TYPE <type>,
            <data_name> TYPE <type> READ-ONLY.

      CLASS-DATA <class_data_name> type <type>.

   PROTECTED SECTION.

   PRIVATE SECTION.
ENDCLASS.

"Implementation part
CLASS <class_name> IMPLEMENTATION.

ENDCLASS.

START-OF-SELECTION.
   "Use a static attribute
   <class_name>=><class_data_name> = 3.

   "Create an object
   DATA(<class_object>) = NEW <class_name>( ).

   "Use an instance attribute
   <class_object>-><data_name> = 1.
```

Rule: Prefer *NEW* to CREATE OBJECT.

Task 2:

Step 1
In report Z*CONNECTIONS create an object “connection1” of your local class.

Step 2
Create an object “connection2” as copy of “connection1” (DATA(connection2) = connection1.)

Step 3
Create an object “connection3” with new #( ).

Step 4
Set the static attributes of your local class.

Step 5
Set the instance attributes of your object “connection1”.

Step 6
Set the instance attributes of your object “connection3”.

Step 7
Check the report behavior in relation to the instance variables of the three objects.

Solution: The solution to the task can be found in a separate document.

Task 3: ***Step 1***
In report Z*CONNECTIONS create a standard table with reference to your local class (TYPE TABLE OF REF TO).

Step 2
Fill each line of the table with random entries.

Step 3
Feel free to debug.

Solution: The solution to the task can be found in a separate document.

2.3 Accessing Methods and Attributes

Goal 1: Call instance methods.
Goal 2: Call static methods.
Goal 3: Call functional methods.

Definition: **Instance methods**
Instance methods are methods which can be only called using the object reference. Instance methods can access instance attributes and instance events.

Definition: ***Static methods***
Static methods are methods which can be called irrespective to the class instance. You can access only static attributes and static events within the Static method.

Definition: ***Functional methods***
Method with precisely one return value declared using RETURNING and any number of formal parameters.

Code: ***Different class definitions and implementations***

```
REPORT <report_name>.
"Defintion Part
"Subdivided into up to three visibility sections
CLASS <class_name> DEFINITION.
   "Components of class
 <any> SECTION.

  METHODS <method_name>
    IMPORTING <input_1>  TYPE <type>
              <input_2>  TYPE <type> DEFAULT <val>
              …
    EXPORTING <output_1> TYPE <type>
              …
    CHANGING  <inout_1>  TYPE <type> OPTIONAL
              …
    RETURNING VALUE(<result>) TYPE <type>.
ENDCLASS.

"Implementation part
CLASS <class_name> IMPLEMENTATION.
  METHOD <method_name>.
    …
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  "Create an object
  DATA(<class_object>) = NEW <class_name>( ).

  DATA(return) = <class_object>-><class_name>
                           ( input_1 = …
                             Input_2 = … ).
```

Definition: ***Importing parameters***
Importing parameters are values that the method receives from the caller. A method can have any number of importing parameters.

Definition: ***Exporting parameters***
Exporting parameters are results that are returned by the method. A method can have any number of exporting parameters. All exporting parameters are optional - a calling program only uses the values that it needs.

Definition: ***Changing parameters***
Changing parameters are values that the method receives from the caller. Unlike importing parameters, the method can change the values of these parameters. They are then returned to the caller under the same name. A method can have any number of changing parameters. Changing

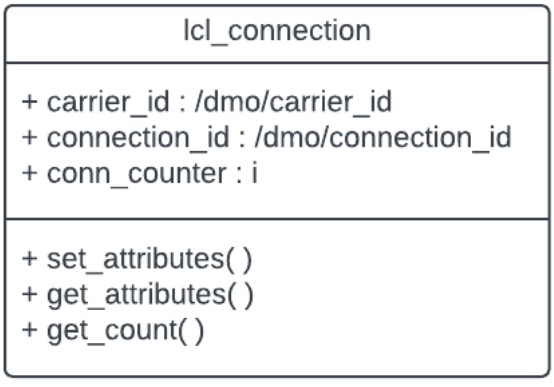
parameters are mandatory by default; you can make them optional in the same way as importing parameters.

Definition: ***Returning parameters***
A returning parameter is a method result that can be used directly in an expression. A method can only have one returning parameter. Returning parameters must use a special form of parameter passing which is called pass-by-value. This form of parameter passing is defined by surrounding the parameter name in brackets (no blanks!) and preceding it with keyword VALUE.

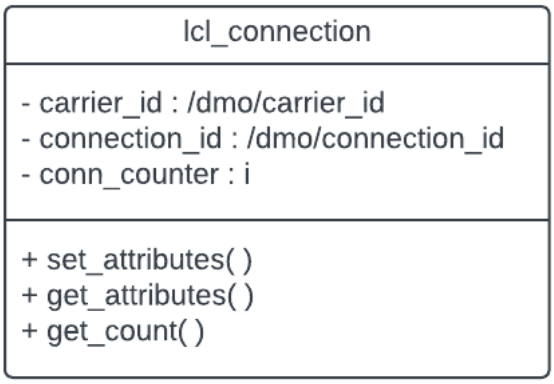
Definition: ***Optional addition***
Using the OPTIONAL addition. The parameter is optional, and its default value is the initial value appropriate to the type of the parameter.

Definition ***Default <val>***
Using the DEFAULT <val> addition. The parameter is optional, and its default value is the value that you specified as <val>.

Task 4: ***Step 1***
In the public section of your local class create three methods to set all attributes, get all attributes and get the counter.



Step 2
Copy all attributes to the private section.



Step 3
Call all methods.

Solution: The solution to the task can be found in a separate document.

Task 5:

Step 1
Change the attributes of your class to get a private table of your instance attribute.

Step 2
Set_attributes( ) will insert a new line of attributes to your table.

Step 3
With get_attributes( ) you will return the table.

Step 4
Loop over get_attributes( ) and make a output of each line.

Solution: The solution to the task can be found in a separate document.

2.4 Implementing Constructors in Local Classes

Goal 1: Create and use constructors.
Goal 2: Understand the difference between static and instance constructs.

Definition: ***Instance constructor***
Instance constructors are called once for each instance.

Definition: **Static constructor**
The static constructor is called once per class and internal session.

Code: ***Instance constructor***

```
CLASS <class_name> DEFINITION.
 PUBLIC SECTION.

  METHODS constructor
    IMPORTING <input_1>  TYPE <type>
              <input_2>  TYPE <type> DEFAULT <val>
              … .
ENDCLASS.
```

Code: ***Static Constructor***

```
CLASS <class_name> DEFINITION.
 PUBLIC SECTION.

  CLASS-METHODS class_constructor.

ENDCLASS.
```

Code: ***Call a static constructor***

```
DATA(<object>) = NEW <cl_name>( <input_1> = …
                                <input_2> = … ).
```

Task 6: ***Step 1***
Implement a constructor to initial set your attribute table.

Solution: The solution to the task can be found in a separate document.

Task 7: ***Step 1***
Test when the static and when the instance constructor is called in your report. What is the difference between a static and an instance constructor.

Solution: The solution to the task can be found in a separate document.

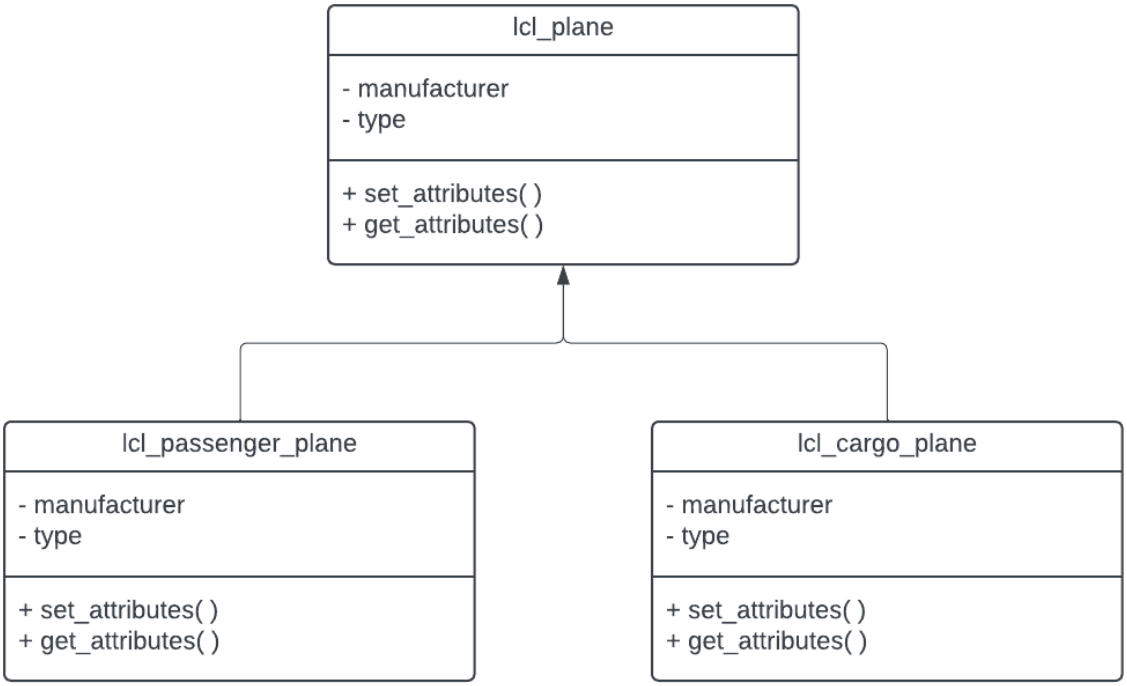
3. Inheritance and casting

3.1. Implementing inheritance

Goal 1: Understand generalization and specialization.
Goal 2: Implement inheritance.
Goal 3: Access elements of classes in inheritance.

Note: Inheritance “is a Relationship”.

Definition: ***Specialization***
The implementation of this relationship in a programming language uses the concept of inheritance, in which a new class (the subclass) is derived from an existing one (the superclass).



Code: ***Defining Inheritance***

```
CLASS lcl_plane DEFINITION.
 PUBLIC SECTION.
   …
ENDCLASS.

CLASS lcl_passenger_plane DEFINITION
 INHERITING FROM lcl_plane.
   …
ENDCLASS.
```

Note: The relationship between superclass and subclass is 1: n.

Note: The subclass knows its superclass but does not know about any sibling classes (other classes that are derived from the

same superclass). In our example, the passenger plane class has no knowledge of the cargo plane class.

Task 8: Implement lcl_plane with its two subclasses using the UML diagram shown, build an object for each and call the methods of the objects.

Solution: The solution to the task can be found in a separate document.

Code: ***How to extend a subclass***

```
CLASS lcl_plane DEFINITION.
 PUBLIC SECTION.

   METHODS constructor
     IMPORTING iv_manufacturer TYPE string
               type            TYPE string
               seats           TYPE i.

   METHODS get_attributes
     EXPORTING et_attributes TYPE tt_attributes.
   …
ENDCLASS.

CLASS lcl_passenger_plane DEFINITION
 INHERITING FROM lcl_plane.

   PUBLIC SECTION.
     METHODS constructor …

     METHODS get_attributes …

   PRIVATE SECTION.

      DATA seats TYPE i.


ENDCLASS.

CLASS lcl_passenger_plane IMPLEMENTATION.

      METHODS constructor.

        super->constructor(
         iv_manufacturer = iv_manufacturer
         iv_type = iv_type ).

        seats = iv_seats.

      ENDMETHODS.

      METHODS get_attributes REDEFINITION.
```

```
        super->get_attributes( ).
      ENDMETHODS.

...
ENDCLASS.
```

Note: ***Three ways to extend a subclass***
There are three ways to extend a subclass.
- add new components.
- redefine methods.
- add a new constructor.

Note: ***Add new components***
You can declare new attributes, types, constants, and methods. Their names must not clash with other names that have already declare components in the superclass.

Note: ***Redefine methods***
When you call an inherited method, the runtime system executes the implementation of the method from the superclass. This implementation cannot, however, consider any new components that you have declared in the subclass. In this case, you may redefine the method. This means that you can assign it a new implementation that is relevant to the subclass.

Note: ***Add a new constructor***
Constructors ensure that new instances of the class are properly initialized. When you create an instance of a subclass, the runtime system always executes the constructor of the superclass. This ensures, for example, that a passenger plane has a manufacturer and a type in just the same way as any other plane. However, the constructor of the superclass cannot initialize attributes of the subclass, as it does not know that they exist. For this reason, you can define a new constructor for the subclass.

Task 9: Implement each way to extend your subclass lcl_passenger_plane. Check the visibility of the attributes.

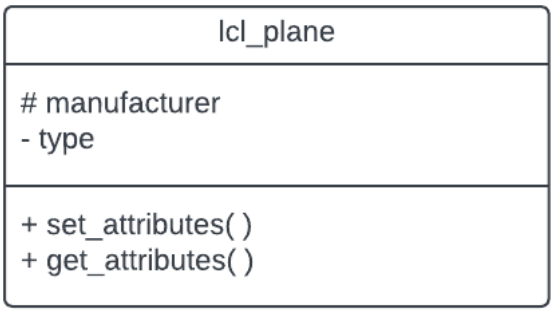
Question: What happens if the super->constructor( ) is not called in the constructor of the subclass?

Solution: The solution to the task can be found in a separate document.

Note: ***Protected Components***

Even though a subclass contains all the attributes of its superclass, it is not allowed to access the private components directly by itself. Sometimes this is necessary, but there are other times when a superclass needs to let its subclasses access attributes or methods without making them fully public. It can do so by declaring them in the protected section. Protected components are visible within the class itself but also to all subclasses.

Task 10: Change lcl_plane and test the visibility



Solution: The solution to the task can be found in a separate document.

3.2. Implementing upcasts using inheritance

Goal: Implement upcasts using inheritance.

Definition: ***Casting***
Assigning object references to a reference variable of a different type is called casting.

Definition: ***Up-cast***
Assigning a reference that points to a subclass to a reference variable with the type of a superclass is called an up-cast, because you are making the assignment to a type higher up in the inheritance hierarchy.

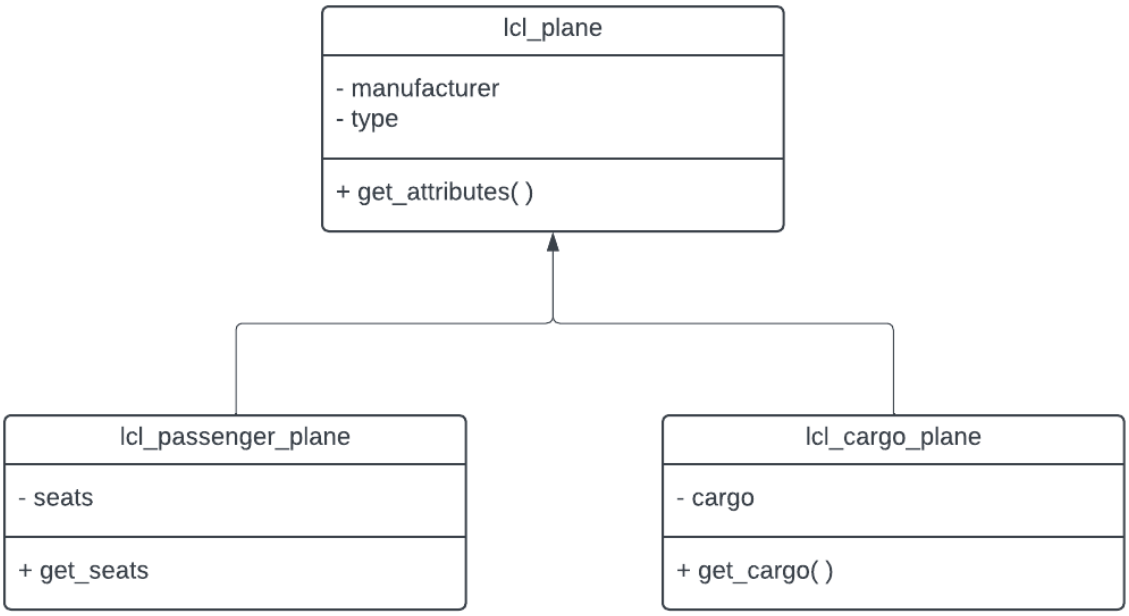
Code: ***Using Superclass References***

```
DATA(passenger_plane) = NEW lcl_passenger_plane( ).
DATA(plane) = CAST lcl_plane( passenger_plane ).

plane->get_seats( ).
plane->get_attributes( ).

DATA(cargo_plane) = NEW lcl_cargo_plane( ).
plane = cargo_plane.
```

Task 11:

Step 1
Implement the code snippet and the changes from the following UML diagram. If there is an error, correct it.

Step 2
Check the objects with the debugger. Feel free to change attributes.



Solution: The solution to the task can be found in a separate document.

Code: ***Using Superclass References II***

```
DATA(passenger_plane) = NEW lcl_passenger_plane( ).
DATA(plane) = CAST lcl_plane( passenger_plane ).

plane->get_seats( ).
plane->get_attributes( ).

DATA(cargo_plane) = NEW lcl_cargo_plane( ).
plane = cargo_plane.
```

Note: Since a subclass contains all components of all super classes along the inheritance tree and the interfaces of methods cannot be changed, a reference variable typed with reference to a superclass or to an interface implemented by a superclass may contain references to objects of all

subclasses of this superclass. This means that the content of a reference variable typed with reference to a subclass can always be assigned to reference variables typed with reference to one of its super classes or the interfaces of these super classes (Up Cast). In particular, the target variable can always be typed with reference to the class object.

3.3. Implementing polymorphism using inheritance

Goal 1: Understand polymorphism.

Definition: ***Polymorphism***
When an instance method is redefined in one or more subclasses, different implementations of the method can be executed after a method call using the same reference variable, depending on where the class of the referenced object is in the inheritance tree. The feature that different classes have the same interface and can therefore be addressed using reference variables of one type is called polymorphism.

3.4. Implementing downcast using inheritance

Goal 1: Understand class hierarchies.
Goal 2: Implement downcast using inheritance.

Code: ***Downcast***

```
DATA(passenger_plane) = NEW lcl_passenger_plane( ).
DATA(plane) = CAST lcl_plane( passenger_plane ).

DATA(cargo_plane) = NEW lcl_cargo_plane( ).
plane = cargo_plane.

passenger_plane = plane.

passenger = CAST #( plane ).
```

Task 12: Implement the code snippet. If there is an error, correct it.

Solution: The solution to the task can be found in a separate document.

Note: ***Cast Operator***

The syntax check does not let you assign a superclass reference directly to a subclass reference. However, you can still do it using the CAST operator.

Code ***Securing a down-cast.***

```
DATA(passenger_plane) = NEW lcl_passenger_plane( ).
DATA(plane) = CAST lcl_plane( passenger_plane ).

IF plane IS INSTANCE OF lcl_passenger_plane.
  passenger = CAST #( plane ).
ENDIF.
```

4. Interfaces and casting

4.1. Defining and implementing local interfaces

Goal 1: Understand the use of interfaces.
Goal 2: Create generalization and specialization relationships using interfaces.
Goal 3: Integrate different submodels using interfaces.
Goal 4: Create and use interface hierarchies.

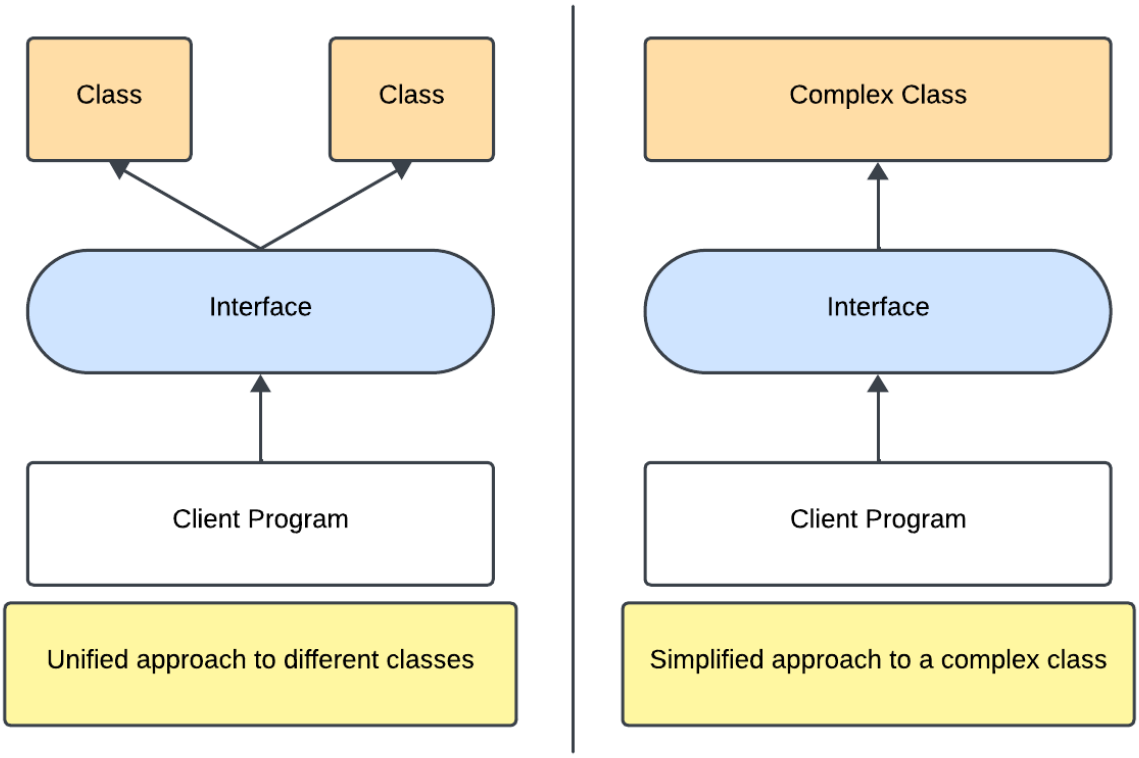
Definition: ***Interfaces***
Interfaces are independent structures that enable the class-specific public points of contact to be enhanced by implementing them in classes.

Note: **Reasons to Use Interfaces**
Interfaces allow you to define how a program will address one or more classes. In an interface, you can define methods, attributes, types, and constants, just as you would in a class. These components form a description of how a client program could interact with the corresponding objects. However, an interface cannot be instantiated, and does not contain any implementations of its methods. To make the concept work, we also need one or more classes that implement the interface. This means that they include the interface in their own class declaration and the components that are declared in the interface become part of the class itself. If the interface contains method declarations, the class that implements the interface must also provide an implementation of that method.

Note: ***Two important uses – unified approach***
Providing a unified approach to different classes: If you have

classes that are not related by inheritance, but that should still provide common services to their users, you can define the common services in an interface. This ensures that each class provides a method with the same name and the same signature to perform a particular task. Programs that use the classes then have a single way to call functions regardless of which class they are addressing.

Note: ***Two important uses – simplified approach***
Providing a simplified approach to a complex class: interfaces provide a restricted view of an object. This enables you to expose a particular set of methods to a program that will use a particular class based on the functions that the program requires.



Code: ***Interface definition***

```
INTERFACE lif_partner. "No definition required.

 TYPES tt_attributes TYPE … "No visible sections.

 METHODS get_partner_attributes
         RETURNING VALUE(rt_attributes)
         TYPE tt_attributes.

ENDINTERFACE.
```

Code: ***Declaring the interface***

```
CLASS lcl_car_rental DEFINITION.
 PUBLIC SECTION.
   INTERFACES lif_partner.
ENDCLASS.

CLASS lcl_car_rental IMPLEMENTATION.
 METHOD lif_partner~get_partner_attributes.
   …
 ENDMETHOD.
ENDCLASS.
```

Task 13: Implement Interfaces for the plane example.

Solution: The solution to the task can be found in a separate document.

Task 14: Copy the following code snippet and try to implement it. Make changes if necessary. Familiarize yourself with the code.

```
INTERFACE lif_partner.
 METHODS get_partner_attributes.
ENDINTERFACE.

CLASS lcl_travel_Agency DEFINITION.
 PUBLIC SECTION.
ENDCLASS.

CLASS lcl_airline DEFINITION.
 PUBLIC SECTION.
 INTERFACES lif_Partner.

 TYPES: BEGIN OF ts_detail,
          name TYPE string,
          value TYPE string,
        END OF ts_detail,
        tt_Details TYPE SORTED TABLE OF ts_detail WITH UNIQUE
     KEY name.


  METHODS get_details RETURNING VALUE(rt_details) TYPE
tt_details.
ENDCLASS.

CLASS lcl_car_Rental DEFINITION.
 PUBLIC SECTION.
 INTERFACES lif_Partner.

 TYPES: BEGIN OF ts_info,
```

```
          name TYPE c LENGTH 20,
          value TYPE c LENGTH 20,
         END OF ts_info,
         tt_Info TYPE SORTED TABLE OF ts_info WITH UNIQUE KEY
      name.

 METHODS get_information RETURNING VALUE(rt_details) TYPE
tt_info.
ENDCLASS.

CLASS lcl_airline IMPLEMENTATION.
 METHOD get_details.
 ENDMETHOD.

 METHOD lif_partner~get_partner_attributes.
 ENDMETHOD.

ENDCLASS.

CLASS lcl_car_rental IMPLEMENTATION.
 METHOD get_information.
 ENDMETHOD.

 METHOD lif_partner~get_partner_attributes.
 ENDMETHOD.
ENDCLASS.
```

Solution: The solution to the task can be found in a separate document.

Task 15: Implement an interface that contains a Main method. Incorporate this method into your coding.

4.2. Implementing polymorphism using interfaces

Goal: Implement polymorphism using interfaces.

Task 16: Implement inheritance between the travel agency and the other objects.

Solution: The solution to the task can be found in a separate document.

4.3. Integrating class models using interfaces

Goal 1: Implement downcast with interfaces.

Code: Declaring an interface reference

```
DATA: partner TYPE REF TO lif_partner.

rental = NEW lcl_rental( ).
partner = rental.

rental = partner.

rental->lif_partner~get_partner_attributes( ).
partner->get_partner_attributes( ).
```

Task 17: Implement the code snippet “declaring an interface reference”. Find the error. Specify two ways to eliminate the error.

Solution: The solution to the task can be found in a separate document.

5. Class-based exceptions

5.1. Explaining class-based exceptions

Goal 1: Understand class-based exceptions.
Goal 2: Understand handling class-based exceptions.
Goal 3: Understand the hierarchy of predefined exception classes.
Goal 4: Understand the different ways of handling an exception.

Definition: ***Class-based exceptions***
Class-based exceptions are realized as instances of exception classes. Exception classes are either predefined globally in the system or can be defined by the user (globally or locally). Class-based exceptions are raised either by the ABAP runtime environment or by a program.

Code: **Exception Classes**

```
DATA: partner TYPE REF TO lif_partner.

DATA number1 TYPE i VALUE 2000000000.
DATA number2 TYPE p LENGTH 2 DECIMALS 1 VALUE '0.5'.
DATA result TYPE i.

TRY.
    result = number1 / number2.
  CATCH cx_sy_arithmetic_overflow.
    out->write ( 'Arithmetic Overflow' ).
ENDTRY.
```

Note: **Sequence of CATCH Statements**

When you write CATCH statements, you must place the most specific class (or classes) first and the most generic ones last. If you place a superclass above one of its subclasses, you will cause a syntax error. This is because the system could not process the specific CATCH block because it is overshadowed by the more generic one.

Code: **Sequence of CATCH Statements.**

```
DATA: partner TYPE REF TO lif_partner.

DATA number1 TYPE i VALUE 2000000000.
DATA number2 TYPE p LENGTH 2 DECIMALS 1 VALUE '0.5'.
DATA result TYPE i.

TRY.
    result = number1 / number2.
  CATCH cx_sy_arithmetic_overflow.
    …
  CATCH cx_sy_arithmetic_error.
    …
  CATCH cx_root.
    …
ENDTRY.
```

Note: **INTO Addition**

Each exception is represented by an object. You can place this object in a reference variable and work with it.

Code: **INTO Addition**

```
DATA: partner TYPE REF TO lif_partner.

DATA number1 TYPE i VALUE 2000000000.
DATA number2 TYPE p LENGTH 2 DECIMALS 1 VALUE '0.5'.
DATA result TYPE i.

TRY.
    result = number1 / number2.
  CATCH cx_root INTO DATA(exception).
    DATA(message) = exception->get_text( ).
ENDTRY.
```

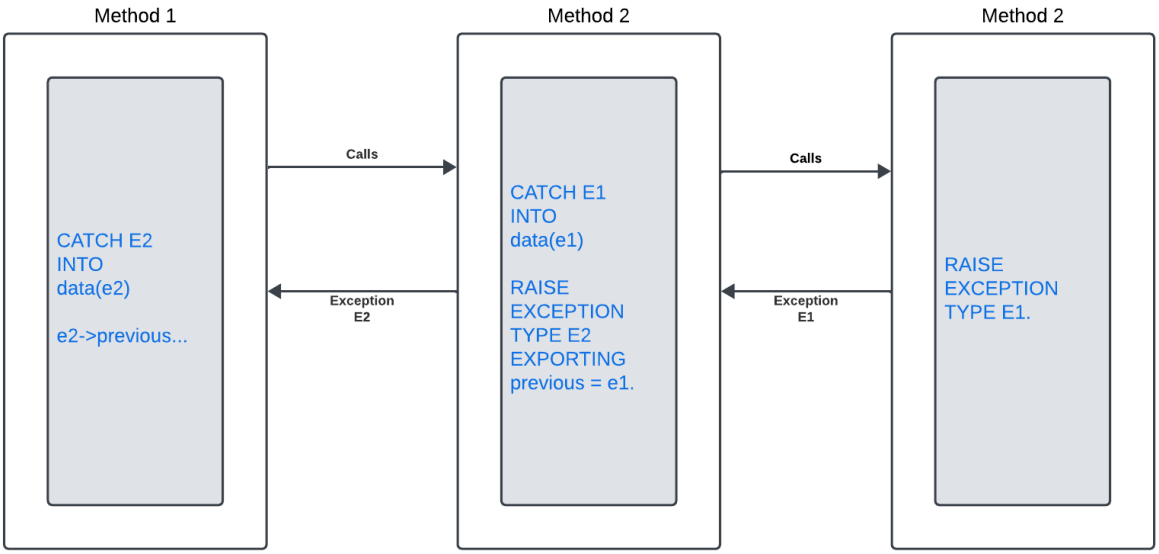
Task 18: Implement the following code snippet. Play around with the coding and familiarize yourself with it.

```
DATA number1 TYPE i VALUE 2000000000.
DATA number2 TYPE p LENGTH 2 DECIMALS 1 VALUE '0.5'.
DATA result TYPE i.

TRY.

result = number1 / number2.

CATCH cx_sy_arithmetic_overflow.
“Implement an output
CATCH cx_sy_zerodivide.
“Implement an output
ENDTRY.


number2 = 0.
TRY.

result = number1 / number2.

CATCH cx_sy_arithmetic_overflow.
“Implement an output
CATCH cx_sy_zerodivide.
“Implement an output
ENDTRY.

TRY.
result = number1 / number2.
CATCH cx_sy_arithmetic_error.
“Implement an output
ENDTRY.

TRY.
result = number1 / number2.
CATCH cx_root.
“Implement an output
ENDTRY.

TRY.
result = number1 / number2.
CATCH cx_root.
“Implement an output
ENDTRY.
```

Solution: The solution to the task can be found in a separate document.

Note: **PREVIOUS**
Sometimes within a call chain a method will catch an exception. In response, it will raise a different exception for its own caller to catch. This might happen, for example, if a framework raises an exception that is too technically

oriented for the application to process by itself. However, in certain circumstances the method that does this (method 2 in the example) may want to trigger its own exception, but also pass on the first exception. It can do this using the previous attribute of the exception class.



5.2. Defining and raising class-based exceptions

Goal 1: Define global exceptions.
Goal 2: Raise class-based exceptions.
Goal 3: Propagate exceptions.

Code: ***Definition of an exception class.***

```
CLASS lcx_no_connection DEFINITION
INHERITING FROM cx_static_check.
 …
ENDCLASS.
```

Task 19: **Step 1:**
Copy code below and use it as a local exception. Implement a report for the call. Play around with the source code to familiarize yourself with it.

```
CLASS lcx_no_connection DEFINITION INHERITING FROM
cx_static_Check.
PUBLIC SECTION.
interfaces if_t100_message.

METHODS constructor
IMPORTING
textid LIKE if_t100_message=>t100key OPTIONAL
previous LIKE previous OPTIONAL
airlineid TYPE /DMO/CARRIER_ID OPTIONAL
connectionnumber TYPE /DMO/CONNECTION_ID OPTIONAL.
CONSTANTS:
BEGIN OF lcx_no_connection,
msgid TYPE symsgid VALUE 'ZS4D401_EXCEPTIONS',
msgno TYPE symsgno VALUE '001',
attr1 TYPE scx_attrname VALUE 'AIRLINEID',
attr2 TYPE scx_attrname VALUE 'CONNECTIONNUMBER',
attr3 TYPE scx_attrname VALUE 'attr3',
attr4 TYPE scx_attrname VALUE 'attr4',
END OF lcx_no_connection.

DATA airlineid TYPE /dmo/carrier_id READ-ONLY.
DATA connectionnumber TYPE /dmo/connection_id READ-ONLY.

ENDCLASS.

CLASS lcx_no_Connection IMPLEMENTATION.
METHOD constructor.

super->constructor( previous = previous ).

me->airlineid = airlineid.
me->connectionnumber = connectionnumber.

CLEAR me->textid.
IF textid IS INITIAL.
if_t100_message~t100key = lcx_no_connection.
ELSE.
if_t100_message~t100key = textid.
ENDIF.


ENDMETHOD.

ENDCLASS.

CLASS lcl_connection DEFINITION.
PUBLIC SECTION.
METHODS constructor
IMPORTING
i_airlineid TYPE /dmo/carrier_id
i_connectionnumber TYPE /dmo/connection_id
RAISING lcx_no_connection.
```

```
PRIVATE SECTION.
DATA AirlineId TYPE /dmo/carrier_id.
DATA ConnectionNumber TYPE /dmo/connection_id.
DATA fromAirport TYPE /dmo/airport_from_id.
DATA toAirport TYPE /dmo/airport_to_id.
ENDCLASS.


CLASS lcl_Connection IMPLEMENTATION.


METHOD constructor.
DATA fromairport TYPE /dmo/airport_from_Id.
DATA toairport TYPE /dmo/airport_to_id.


SELECT SINGLE FROM /dmo/connection
FIELDS airport_from_id, airport_to_id
WHERE carrier_id = @i_airlineid
AND connection_id = @i_connectionnumber
INTO ( @fromairport, @toairport ).


IF sy-subrc <> 0.
RAISE EXCEPTION TYPE lcx_no_connection
EXPORTING
airlineid = i_airlineid
connectionnumber = i_connectionnumber.
ELSE.
me->connectionnumber = i_connectionnumber.
me->fromairport = fromairport.
me->toairport = toairport.
ENDIF.
ENDMETHOD.
ENDCLASS.
```

Solution: The solution to the task can be found in a separate document.

5.3. Implementing advanced based exception handling

Goal 1: Retry after exceptions.
Goal 2: Implement resumable exceptions.
Goal 3: Understand Unwind.
Goal 4: Understand Cleanup
Goal 5: Understand Resume.

Code: ***Syntax RETRY, UNWIND, CLEANUP***

```
TRY.
    …
  CATCH [BEFORE UNWIND] lcx_class1 [INTO DATA(oref)].
```

```
    …
    IF oref->is_resumeable = abap_true.
      [RESUME].
    endif.

    [RETRY].
   CLEANUP [INTO DATA(oref)].
    …
ENDTRY.
```

Code: ***Raise resumable exception.***

```
CLASS lcl_implementation IMPLEMENTATION.
  METHOD meth.
    …
    RAISE RESUMEABLE EXCEPTION TYPE lcx_resume.
    …
  ENDMETHOD.
ENDCLASS.
```

Definition: ***RETRY***
This statement exits the CATCH handling of a class-based exception and continues processing with the TRY statement of the current TRY control structure.

Definition: ***CLEANUP***
A CLEANUP block is executed when a class-based exception in the TRY block of the same TRY control structure is raised but is handled in a CATCH block of an external TRY control structure. A CLEANUP block is executed immediately before the context of the exception is deleted.

Definition: ***RESUME***
This statement exits the CATCH handling of a resumable exception and resumes processing after the statement that raised the exception. This statement can only be executed in a CATCH block of a TRY control structure for which the addition BEFORE UNWIND is declared. When exception handling is exited using RESUME, the context of the exception is not deleted, and any CLEANUP blocks are not executed.

Definition: ***BEFORE UNWIND***
If the addition BEFORE UNWIND is specified, the context in which the exception was raised is not deleted until the CATCH block is deleted. Instead, the context is preserved (including all called procedures and their local data) during

the execution of the CATCH block.

If no RESUME statement is executed in the CATCH block, the context is deleted when the CATCH block is exited.

Of a RESUME statement is executed in the CATCH block, processing resumes after the statement that raised the exception.

Task 20: Implement and throw exceptions with RETRY, CLEANUP, RESUME and BEFORE_UNWIND.

Solution: The solution to the task can be found in a separate document.

6. Object-oriented repository objects

6.1. Creating global classes

Goal 1: Create global classes.
Goal 2: Test global classes.
Goal 3: Use global classes.

Task 21: Implement a global class and global exceptions. You can choose a suitable previous task as an example. Call the class from a report.

Question: Find three suitable ways to test your class.

Solution: The solution to the task can be found in a separate document.

6.2. Defining and implementing global interfaces

Goal: Implement global interfaces.

Task 22: Define and implement an interface for the class from the previous task.

Solution: The solution to the task can be found in a separate document.

6.3. Use local classes in global classes.

Goal 1: Understand why to use local classes in a global class.
Goal 2: Implement a local class in a global class.

Discussion 1: When should global classes and when should local classes be used?

Task 23: Implement a local class in a global class. Try to call the local class within the global class. Try to use the local class within a report that uses the global class as an object. Test whether you can use the local class in inheritance.

Discussion 2: When should global classes and when should local classes be used?

Solution: The solution to the task can be found in a separate document.

7. Examples for object-oriented ABAP programming

7.1. Using ABAP class-based list viewer.

Goal 1: Implement a simple ALV Grid.

Task 24: Copy zjaj_salv_01.

7.2. Implement events (double click event in ALV Grid)

Goal 1: Implement event-controlled method calls.
Goal 2: Trigger and handle events.
Goal 3: Register for events.
Goal 4: Understand visibility sections in event handling.

Task 25: Copy class zcl_alv_service. Use the class in your ALV report.

Task 26: Implement double-click.

Additional Task 1: Reorganize your report so that only one main method is shown in START-OF-SELECTION.

Additional Task 2: Throw an exception if no data is found in the database.

```
* Possible events are:
* SET HANDLER gr_events->on_user_command FOR o_alv-
>get_event( ).
* SET HANDLER gr_events->on_before_user_command FOR o_alv-
>get_event( ).
* SET HANDLER gr_events->on_after_user_command FOR o_alv-
>get_event( ).
* SET HANDLER gr_events->on_double_click FOR o_alv-
>get_event( ).
```

```
* SET HANDLER gr_events->on_top_of_page FOR o_alv->get_event(
).
* SET HANDLER gr_events->on_end_of_page FOR o_alv->get_event(
).

CLASS lcl_events DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS: on_double_click FOR EVENT double_click OF
cl_salv_events_table
      IMPORTING
          row
          column
          sender.
    CLASS-METHODS: on_link_click FOR EVENT link_click OF
cl_salv_events_table
      IMPORTING
          row
          column
          sender.
ENDCLASS.

CLASS lcl_events IMPLEMENTATION.
  METHOD on_double_click.
    MESSAGE | { row } , { column } | TYPE 'I'.
  ENDMETHOD.
  METHOD on_link_click.
    MESSAGE | { row } , { column } | TYPE 'I'.
  ENDMETHOD.
ENDCLASS.

DATA: o_salv TYPE REF TO cl_salv_table.

...

SET HANDLER lcl_events=>on_double_click FOR o_salv-
>get_event( ).
SET HANDLER lcl_events=>on_link_click FOR o_salv->get_event(
).
```

Solution: The solution to the task can be found in a separate document.

7.3. Business Add-Ins (BAdIs)

Goal 1: Understand object oriented BAdIs.
Goal 2: Implement a BAdI.

Definition: Business Add-Ins (BAdIs) are enhancements to the standard version. They can be inserted into the SAP System to accommodate user requirements too specific to be included in the standard delivery. Since specific industries often

require special functions, SAP allows you to predefine these points in your software.

Hands On: Eclipse

Task 27: Implement BAdIs using filters.

8. Object-Oriented Design Patterns

8.1. Implementing Advanced Object-Oriented Techniques

Goal 1: Implement abstract classes.
Goal 2: Implement final classes.
Goal 3: Work with conditions on dynamic type of an object reference.
Goal 4: Restrict the visibility of the instance constructor.

Definition: ***Abstract classes***
Abstract Class is a special kind of class which can't be instantiated. We can only instantiate the subclasses of the Abstract class if they are not abstract. Abstract class should at least contain one abstract method. Abstract methods are methods without any implementation – only a declaration. We can certainly define the variables referencing to Abstract class and instantiate with specific subclass at runtime.

Differences: ***Differences between abstract classes and interfaces***

Multiple Inheritance
We can achieve multiple inheritance using Interfaces. Since ABAP doesn't support more than one Super class, we can have only one abstract class as Super class.

New Functionality
If we add a new method in the Interface, all the implementing classes must implement this method. If we don't implement the method, it would result into Run-time error. For Abstract class, if we add a non-abstract method, it's not required to redefine that in each inherited class.

Default Behavior
We can have a default behavior of a non-abstract method in abstract class. We can't have any implementation in Interface as it only contains the empty stub.

Visibility
All interface components are PUBLIC by default. For Abstract class, we can set the visibility of each component.

Code: ***Simple abstract class***

```
CLASS zcl_implementation DEFINITION ABSTRACT.
  PUBLIC SECTION.
    METHODS: <class_name> ABSTRACT IMPORTING … TYPE … .
  PRIVATE SECTION.
    METHODS: <class_name> …
ENDCLASS.
```

Definition: ***Final class***
Final classes cannot have any further subclasses and finalize an inheritance tree.

Note: In classes that are both abstract and final, only the static components can be used. Instance components can be declared, but these cannot be used. The joint specification of ABSTRACT and FINAL therefore only makes sense for static classes.

Note: Private methods cannot be redefined and therefore cannot be abstract.

Code: ***Simple final class***

```
CLASS zcl_implementation DEFINITION FINAL.
  …
ENDCLASS.
```

Task 29: Inheritance from an abstract class.

Task 30: Create a simple final class (maximum one method) and inherit from it.

8.2. Implementing Factory Methods and Singleton Patterns

Goal 1: Implement factory methods.
Goal 2: Implement the singleton pattern.

Note: Sometimes a class needs to keep control of its instances. This could be to prevent multiple instances being created. To do so, it must prevent it"s users from creating instances themselves. You do this in ABAP by using the CREATE PRIVATE addition in the CLASS DEFINITION statement.

Definition: **Singleton pattern**
A singleton pattern ensures that you always get back the same instance of whatever type you are retrieving, whereas the factory pattern generally gives you a different instance of each type.

Definition: **Factory pattern**
The purpose of the factory is to create and return new instances.

Code: ***Factory / Singleton***

```
CLASS lcl_connection DEFINITION CREATE PRIVATE.
 PUBLIC SECTION.
   CLASS-METHODS get_connection
                 IMPORTING airlineID
                           TYPE /dmo/carrier_id
                           ConnectionNumber
                           TYPE /dmo/connection_id
                  RETURNING VALUE(ro_connection)
                            TYPE REF TO lcl_connection.
ENDCLASS.

"Find the correct implementation.
"Option A
DATA(connection1) = new lcl_connection( … ).
"Option B
DATA(connection2) = lcl_connection=>get_connection( … ).
```

Note: Any attempt to use the NEW operator outside the class will now lead to a syntax error. Note that other programming languages achieve this effect by altering the visibility of the constructor method. ABAP does not support this.

Task 30: Implement a simple singleton and factory method.

Additional Task: ***Step 1:***
Implement a global class-based factory method for the plane sample.

Step 2:
Implement a global class-based singleton pattern for the plane sample.

Solution: The solution to the tasks can be found in a separate document.

8.3. Implementing Friendship

Goal 1: Implement friendship relationships.

Definition: ***Frinds***
A class can grant friendship to other classes and interfaces (and thus to all classes that implement this interface). This relationship is created using the FRIENDS additions of the statement CLASS ... DEFINITION by specifying all classes and interfaces to which friendship is to be granted. These friends are granted access to all components of the class offering the friendship, regardless of their visibility section or the addition READ-ONLY and can always create instances of this class regardless of the addition CREATE of the statement CLASS.

Note: Friendship granted is not inherited, in contrast to the friend attribute. A friend of a superclass is, therefore, not automatically a friend of its subclasses.

Task 31: Test Report ...

Note: ***The three friends' additions***

... FRIENDS cif1 ... cifn
Friendship relationships between local classes of a program and other classes as well as interfaces in the same program and the class library

... GLOBAL FRIENDS cif1 ... cifn
Exclusively for global classes through the Class Builder. Friendly relationships to other global classes and interfaces

... LOCAL FRIENDS cif1 ... cifn
Defined separately from the class declaration in a class pool. Friendship with the local classes and interfaces of the class pool of the global class.

Disadvantages: Friends classes can make your code less secure as they allow unauthorized access to private and protected elements.

Friends classes can make your code less comprehensible, as it can be difficult to understand the flow of information between friends classes.

Friends classes introduce a dependency between the two classes. This means that if you make changes to one class, you must also make changes to the other class.