S4D400

Basic ABAP Programming

EXERCISES AND SOLUTIONS

Course Version: 24
Exercise Duration: 6 Hours 35 Minutes
Material Number: 50164578

SAP Copyrights, Trademarks and Disclaimers

Typographic Conventions

American English is the standard used in this handbook.

The following typographic conventions are also used.

This information is displayed in the instructor's presentation	
Demonstration	
Procedure	
Warning or Caution	
Hint	
Related or Additional Information	
Facilitated Discussion	
User interface control	*Example text*
Window title	*Example text*

Contents

Unit 1
Exercise 1

Create an ABAP Cloud Project and Investigate ABAP Coding

In this exercise, you start your ABAP cloud project and analyze some existing ABAP code to get familiar with some basic features of the ABAP development environment.

Task 1: Create an ABAP Cloud Project
Open Eclipse, switch to the ABAP perspective and create an ABAP cloud project.

1. Open Eclipse and switch to the ABAP perspective.
2. Create an *ABAP Cloud Project*.

Task 2: Analyze ABAP Class /DMO/CL_FLIGHT_LEGACY
Use some functions of the *ABAP Development Tools* to analyze the source code of ABAP class **`/DMO/CL_FLIGHT_LEGACY`**.

Note:
You are not required to read the code in this exercise. Instead, concentrate on the navigation and display functions in the development environment.

1. Open the ABAP class **`/DMO/CL_FLIGHT_LEGACY`** in the editor view.
2. Open the *Properties* tab to display the administrative data of development object **`/DMO/CL_FLIGHT_LEGACY`**.
3. Locate the ABAP class **`/DMO/CL_FLIGHT_LEGACY`** in the *Project Explorer* view on the left of the display.
4. Find the string **`get_instance( )->get`** in the source code of ABAP class **`/DMO/CL_FLIGHT_LEGACY`**.

Note:
There has to be exactly one blank between the brackets.

5. Navigate to the definition of code element `get( .`
6. A few code lines down there is a line starting with `SELECT`. Without navigating away, display some information about code element `/dmo/travel` after keyword `FROM`.
7. Display the ABAP Language help for the keyword `SELECT`.

Unit 1
Solution 1

Create an ABAP Cloud Project and Investigate ABAP Coding

In this exercise, you start your ABAP cloud project and analyze some existing ABAP code to get familiar with some basic features of the ABAP development environment.

Task 1: Create an ABAP Cloud Project
Open Eclipse, switch to the ABAP perspective and create an ABAP cloud project.

1. Open Eclipse and switch to the ABAP perspective.
   a) Open Eclipse and close all tabs.
   b) Choose *Window* → *Perspective* → *Open Perspective* → *Other*.
   c) In the dialog box, double-click *ABAP*.
2. Create an *ABAP Cloud Project*.
   a) Choose *File* → *New* → *ABAP Cloud Project*.
   b) Select *Use a Service Key* and choose *Next*.
   c) Choose *Import* and select the file containing the service key that you have been given.
   d) Choose *Open Logon Page in Browser*.
   e) On the browser page that opens, choose *directaccess.accounts.ondemand.com* and log on with the user and password that you have been given.
   f) When you see the message, *You have been successfully logged on*, close the browser window and return to Eclipse.
   g) To finish creating the project, choose *Finish*.

Task 2: Analyze ABAP Class /DMO/CL_FLIGHT_LEGACY
Use some functions of the *ABAP Development Tools* to analyze the source code of ABAP class **`/DMO/CL_FLIGHT_LEGACY`**.

Note:
You are not required to read the code in this exercise. Instead, concentrate on the navigation and display functions in the development environment.

1. Open the ABAP class **`/DMO/CL_FLIGHT_LEGACY`** in the editor view.
   a) In the Eclipse menu, choose *Navigate* → *Open ABAP Development Object ...* or press **`Ctrl + Shift + A`**
   b) In the input field, enter **`/DMO/CL_FLIGHT`** as search string.

c) In the list of matching items, click on */DMO/CL_FLIGHT_LEGACY (Global Class)* and choose *OK*.

2. Open the *Properties* tab to display the administrative data of development object **`/DMO/CL_FLIGHT_LEGACY`**.

   a) Place the cursor anywhere in the source code of class **`/DMO/CL_FLIGHT_LEGACY`**.

   b) In the tab strip below the editor view, navigate to tab *Properties*.

   c) Analyze the administrative data that are displayed there, for example the original language, the time stamp of the last change or the user that at first created the object.

3. Locate the ABAP class **`/DMO/CL_FLIGHT_LEGACY`** in the *Project Explorer* view on the left of the display.

   a) Place the cursor anywhere in the source code of the class **`/DMO/CL_FLIGHT_LEGACY`**.

   b) On the tool bar of the project explorer, choose *Link with Editor*. This should expand a part of the tree under your ABAP project with the ABAP class **`/DMO/CL_FLIGHT_LEGACY`** as its end point.

4. Find the string **`get_instance( )->get`** in the source code of ABAP class **`/DMO/CL_FLIGHT_LEGACY`**.

   

   Note:
   There has to be exactly one blank between the brackets.

   a) Place the cursor in the first row of the source code of class **`/DMO/CL_FLIGHT_LEGACY`**.

   b) Press **`Ctrl + F`** to open the *Find/Replace* dialog.

   c) In the input field labeled with *Find:* enter **`get_instance( )->get`** and choose *Find*.

5. Navigate to the definition of code element `get( .`

   a) In the current code line, place the cursor on `get(` and choose *Navigate → Navigate to*. Alternatively, press **`F3`**.

6. A few code lines down there is a line starting with `SELECT`. Without navigating away, display some information about code element `/dmo/travel` after keyword `FROM`.

   a) In the code line starting with `SELECT`, place the cursor on `/dmo/booking` and choose *Source Code → Show Code Element Information*. Alternatively, press **`F2`**.

7. Display the ABAP Language help for the keyword `SELECT`.

   a) At the beginning of the current code line, place the cursor on `SELECT` and choose *Source Code → Show ABAP Language Help*. Alternatively, press **`F1`**.

   

   Note:
   There is no need to read or understand the documentation at this point. If you are familiar with the *Structured Query Language (SQL)* you can have a look at the first paragraph in section *Effect*.

Unit 1
Exercise 2

Create an ABAP Package

In this exercise, you first add package **`/LRN/S4D400_EXERCISE`** to the list of *Favorite Packages*. You then create a new package under the superpackage **`ZLOCAL`** (suggested name: **`ZS4D400_##`** where ## is your group number).

Note:
This package will be the home for all development objects that you will create in this course.

Your new package should have the following attributes:

Attribute	Value
Name	**`ZS4D400_##`**, where **`##`** is your group number
Description	**`My ABAP Package`**
Add to favorite packages	Checked
Superpackage	**`ZLOCAL`**
Package Type	**`Development`**
Software Component	**`ZLOCAL`**
Application Component	Leave this field blank
Transport Layer	Leave this field blank

1. Add package **`/LRN/S4D400_EXERCISE`** to the list of *Favorite Packages*.
2. In your ABAP Cloud project, create a new package with the attributes listed above. When you are prompted for a transport choose the transport request in which you are involved. If no transport request is listed, create a new request.
3. Ensure that your new package has been added to the *Favorite Packages*.

Create an ABAP Package

In this exercise, you first add package **/LRN/S4D400_EXERCISE** to the list of *Favorite Packages*. You then create a new package under the superpackage **ZLOCAL** (suggested name: **ZS4D400_##** where ## is your group number).

Note:
This package will be the home for all development objects that you will create in this course.

Your new package should have the following attributes:

Attribute	Value
Name	**ZS4D400_##**, where **##** is your group number
Description	**My ABAP Package**
Add to favorite packages	Checked
Superpackage	**ZLOCAL**
Package Type	**Development**
Software Component	**ZLOCAL**
Application Component	Leave this field blank
Transport Layer	Leave this field blank

1. Add package **/LRN/S4D400_EXERCISE** to the list of *Favorite Packages*.
   a) In the *Project Explorer* on the left, expand your ABAP cloud project and then sub node *Favorite Packages*.
   b) Right-click *Favorite Packages* and choose *Add Package ...*.
   c) In the search field, enter **/LRN/S4D400**.
   d) From the list of matching items, select */LRN/S4D400_EXERCISE*, then choose *OK*.
2. In your ABAP Cloud project, create a new package with the attributes listed above. When you are prompted for a transport choose the transport request in which you are involved. If no transport request is listed, create a new request.
   a) In the Project Explorer, right-click on your ABAP Cloud Project and choose *New* → *ABAP Package* .

b) Enter the package name **ZS4D400_##** where **##** is your group number.

c) Enter the description **My ABAP Package**.

d) Mark the checkbox *Add to favorite packages*

e) Enter the superpackage **ZLOCAL**.

f) Ensure that the *Package Type* is set to **Development**.

g) Choose *Next*.

h) Ensure the *Software component* is set to **ZLOCAL**.

i) Ensure that *Application component* is empty.

j) Ensure that *Transport layer* is empty.

k) Choose *Next*.

l) Check if there is a transport request listed under option *Choose from requests in which I am involved*. If this is the case, choose this option. If the list is empty, choose the option *Create a new request* and enter a request description, for example **ABAP Exercises**.

m) Choose *Finish*.

3. Ensure that your new package has been added to the *Favorite Packages*.

a) In the *Project Explorer* on the left, expand your ABAP cloud project and then sub node *Favorite Packages*.

b) Make sure your own package, **ZS4D400_##**, is listed here. If not, add it like you added package **/LRN/S4D400_EXERCISE** earlier.

Unit 1
Exercise 3

Create a 'Hello World' Application

Create a Hello World Application
In your package, create an new ABAP class. Let the class implement interface **`IF_OO_ADT_CLASSRUN`** so that you can use the class as the main program for an Eclipse console app.

1. In your package, create a new ABAP class with the name **`ZCL_##_HELLO_WORLD`**. Ensure that it uses the interface **`IF_OO_ADT_CLASSRUN`**. When you are prompted to assign the class to a transport request, use the transport request that you created in the previous task.
2. In the **`if_oo_adt_classrun~main( )`** method, use `out->write( )` to output the phrase *Hello World*.
3. Activate and test your class.
4. Check the output in the *Console* view of Eclipse.

Unit 1
Solution 3

Create a 'Hello World' Application

Create a Hello World Application
In your package, create an new ABAP class. Let the class implement interface **IF_OO_ADT_CLASSRUN** so that you can use the class as the main program for an Eclipse console app.

1. In your package, create a new ABAP class with the name **ZCL_##_HELLO_WORLD**. Ensure that it uses the interface **IF_OO_ADT_CLASSRUN**. When you are prompted to assign the class to a transport request, use the transport request that you created in the previous task.
   a) Choose *File* → *New* → *ABAP Class*.
   b) Enter your package **ZS4D400_##**, where **##** is your group number.
   c) Enter the name **ZCL_##_HELLO_WORLD** where **##** is your group number, and enter a description for your class.
   d) Choose *Add...* (next to the *Interfaces* group box).
   e) Enter the filter text **IF_OO_ADT_CLASSRUN**. Double-click the matching entry in the hit list.
   f) Choose *Next*.
   g) Select *Choose from requests in which I am involved* and your own transport request.
   h) Choose *Finish*.
2. In the **if_oo_adt_classrun~main( )** method, use `out->write( )` to output the phrase *Hello World*.
   a) In the editor, enter the following coding between `METHOD if_oo_adt_classrun~main` and `ENDMETHOD.`:

```
out->write( 'Hello World' ).
```

3. Activate and test your class.
   a) Activate the class with the keyboard shortcut **Ctrl + F3**.
   b) Run the class with the **F9** key.
4. Check the output in the *Console* view of Eclipse.
   a) Check the *Console* view that should have opened as a new tab below the editor view.
   b) If the *Console* view is not visible, open it by choosing *Window* → *Show view* → *Other*. Double-click *Console* in the hit list.

Unit 2
Exercise 4

Declare Variables and Process Data

In this exercise, you create a program that does a calculation based on two numbers, then formats and outputs the result.

Template:

none

Solution:

/LRN/CL_S4D400_BTS_COMPUTE (global Class)

Task 1: Variable Declaration
In your own package, create a new global class and define numeric variables for the input.

1. Create a new global class that implements the interface **`IF_OO_ADT_CLASSRUN`** (suggested name: **`ZCL_##_COMPUTE`**, where ## stands for your group number).
2. In the *IF_OO_ADT_CLASSRUN~MAIN* method, declare two variables for integer values (suggested names: **`number1`** and **`number2`**).
3. Implement a value assignment for each of the two numeric variables. Assign a negative value to the first variable and a positive value to the second variable.

Task 2: Data Processing
Calculate a ratio and write the result to the console.

1. Implement an arithmetic expression that calculates the ratio of the two numbers. Store the result in a variable that you declare using an inline declaration (suggested name: **`result`**).
2. Implement a string template that puts together a text like this:“6 / 3 = 2”, with the actual values of the variables instead of 6, 3, and 2. Store the result in a variable that you declare using an inline declaration (suggested name: **`output`**).
3. Call method **`out->write( … )`** to write the text to the console.

Task 3: Activate and Test
Activate and test the console app.

1. Activate the class.
2. Execute the console app.
3. Test the app with different input values.

What is the result of -8 / 3?

_____

_____

_____

How can you explain this result?

_____

_____

_____

Task 4: Control the Result Precision

Make sure the result is rounded to a value with 2 decimal places, and not to an integer value.

1. Instead of an inline declaration, use an explicit declaration of variable **`result`**. Use a numeric type that allows to specify the number of decimal places.
2. Activate the class and test again.

Unit 2
Solution 4

Declare Variables and Process Data

In this exercise, you create a program that does a calculation based on two numbers, then formats and outputs the result.

Template:

none

Solution:

/LRN/CL_S4D400_BTS_COMPUTE (global Class)

Task 1: Variable Declaration
In your own package, create a new global class and define numeric variables for the input.

1. Create a new global class that implements the interface **`IF_OO_ADT_CLASSRUN`** (suggested name: **`ZCL_##_COMPUTE`**, where ## stands for your group number).
   a) Choose *File* → *New* → *ABAP Class*.
   b) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_COMPUTE`**, where ## stands for your group number.
   c) Enter a description.
   d) In the *Interfaces* group box, choose *Add*.
   e) Enter **`IF_OO_ADT_CLASSRUN`**. When the interface appears in the hit list, double-click it to add it to the class definition.
   f) Choose *Next*.
   g) Select your transport request and choose *Finish*.
2. In the *IF_OO_ADT_CLASSRUN~MAIN* method, declare two variables for integer values (suggested names: **`number1`** and **`number2`**).
   a) Add the following code:

```
DATA number1 TYPE i.
DATA number2 TYPE i.
```

3. Implement a value assignment for each of the two numeric variables. Assign a negative value to the first variable and a positive value to the second variable.
   a) Add the following code:

```
number1 = -8.
number2 =  3.
```

Task 2: Data Processing
Calculate a ratio and write the result to the console.

1. Implement an arithmetic expression that calculates the ratio of the two numbers. Store the result in a variable that you declare using an inline declaration (suggested name: **`result`**).
   a) Add the following code:

```
DATA(result) = number1 / number2.
```

2. Implement a string template that puts together a text like this:“6 / 3 = 2”, with the actual values of the variables instead of 6, 3, and 2. Store the result in a variable that you declare using an inline declaration (suggested name: **`output`**).
   a) Add the following code:

```
DATA(output) = |{ number1 } / { number2 } = { result }|.
```

3. Call method **`out->write( … )`** to write the text to the console.
   a) Proceed as you have done in the previous exercise.

Task 3: Activate and Test

Activate and test the console app.

1. Activate the class.
   a) Press **`Ctrl + F3`** to activate the class.
2. Execute the console app.
   a) Press **`F9`** to execute the console app.
3. Test the app with different input values.
   a) Edit the value assignments, then activate and execute as before.

   What is the result of -8 / 3?

   The result is -3.

   How can you explain this result?

   Due to the inline declaration, variable **`result`** is of integer type and ABAP rounds the result using the “half-up” rounding rule.

Task 4: Control the Result Precision

Make sure the result is rounded to a value with 2 decimal places, and not to an integer value.

1. Instead of an inline declaration, use an explicit declaration of variable **`result`**. Use a numeric type that allows to specify the number of decimal places.
   a) At the beginning of method **`if_oo_adt_classrun~main`**, add the following code:

```
DATA result TYPE p LENGTH 8 DECIMALS 2.
```

   b) In the code line with the inline declaration, replace `DATA(result)` with `result`.
2. Activate the class and test again.
   a) Proceed as you have done before.

b) Compare your code to the following extract from the model solution:

```
 METHOD if_oo_adt_classrun~main.

* Declarations
**************************

    DATA number1 TYPE i.
    DATA number2 TYPE i.

    DATA result TYPE p LENGTH 8 DECIMALS 2.


* Input Values
**************************

    number1 = -8.
    number2 =  3.

* Calculation
**************************

*    DATA(result) = number1 / number2.   "with inline declaration

   result = number1 / number2.

    DATA(output) = |{ number1 } / { number2 } = { result }|.

* Output
**************************

    out->write( output ).

  ENDMETHOD.
```

Unit 2
Exercise 5

Implement Conditional Branching

In this exercise, you enable the program for the four basic calculation operations and handle error situations.

Template:

/LRN/CL_S4D400_BTS_COMPUTE (global Class)

Solution:

/LRN/CL_S4D400_BTS_BRANCH (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class **`ZCL_##_COMPUTE`**.

1. Open the source code of global class **`/LRN/CL_S4D400_BTS_COMPUTE`** in the ABAP editor.
2. Link the *Project Explorer* view with the editor.
3. Copy class **`/LRN/CL_S4D400_BTS_COMPUTE`** to a class in your own package (suggested name: **`ZCL_##_BRANCH`**, where ## stands for your group number).

Task 2: Variable for Operator
Declare a variable for the calculation operator and make the calculation dependent on that variable.

1. Declare a variable for the calculation operator (suggested name: **`op`**). The variable needs a type of fixed length to hold one of the following values: “+”, “-”, “*”, or “/”.
2. Directly after the value assignment for variables **`number1`** and **`number2`**, add a value assignment to set the new variable **`op`** to value “/”.
3. Use a suitable control structure to fill the variable **`result`** with different arithmetic expressions, depending on the value of the variable **`op`**.
4. Adjust the string template to use the content of **`op`**, rather than the literal character “/”.
5. Activate and test the application with different values for **`number1`**, **`number2`**, and **`op`**.

   What happens if you set **`op`** to value “%”?

   _____

   _____

   _____

Task 3: Handle Invalid Operator
Handle an invalid value of **`op`**. If **`op`** contains a value other than “+”, “-”, “*”, or “/”, write an error message to the console instead of the calculation and the result value.

1. In the CASE control structure, add a branch that is executed for any other values of **op**.
2. In this branch, fill **output** with a suitable text to describe the error situation.

Note:
We suggest that at this point, you replace the inline declaration of the variable **output** with an explicit declaration at the beginning of the method.

3. Use a suitable control structure to make sure the **output** variable is only filled with the calculation result if it is not already filled with an error text.
4. Activate and test the application with different values for **number1**, **number2**, and **op**.

   What happens if you set **number1** to **1**, **number2** to **0**, and **op** to **/**?

   _____

   _____

   _____

Task 4: Handle Division by Zero

Prevent the program termination by handling the exception.

1. Inside the branch for division (**op** has value "/"), surround the calculation with a pair of TRY and ENDTRY statements.
2. Before the ENDTRY statement, add a CATCH-block for the exception that is raised in case of a division by zero error.
3. In the CATCH-block, fill the **output** variable with a suitable error text.
4. Activate and test the application with different values for **number1**, **number2**, and **op**. In particular, test the handling of the error situations: invalid operator and division by zero.

Unit 2
Solution 5

Implement Conditional Branching

In this exercise, you enable the program for the four basic calculation operations and handle error situations.

Template:

/LRN/CL_S4D400_BTS_COMPUTE (global Class)

Solution:

/LRN/CL_S4D400_BTS_BRANCH (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class **`ZCL_##_COMPUTE`**.

1. Open the source code of global class **`/LRN/CL_S4D400_BTS_COMPUTE`** in the ABAP editor.
   a) In the *Eclipse* toolbar, choose *Open ABAP Development Object*. Alternatively, press **`Ctrl + Shift + A`**.
   b) Enter **`/LRN/CL_S4D400_BTS`** as search string.
   c) From the list of development objects choose */LRN/CL_S4D400_BTS_COMPUTE*, then choose *OK*.
2. Link the *Project Explorer* view with the editor.
   a) In the *Project Explorer* toolbar, find the *Link with Editor* button. If it is not yet selected, choose it. As a result, the development object, that is open in the editor, should be highlighted in the *Project Explorer*.
3. Copy class **`/LRN/CL_S4D400_BTS_COMPUTE`** to a class in your own package (suggested name: **`ZCL_##_BRANCH`**, where ## stands for your group number).
   a) In the *Project Explorer* view, right-click class **`/LRN/CL_S4D400_BTS_COMPUTE`** to open the content menu.
   b) From the context menu, choose *Duplicate ...*.
   c) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_BRANCH`**, where ## stands for your group number.
   d) Adjust the description and choose *Next*.
   e) Confirm the transport request and choose *Finish*.

Task 2: Variable for Operator
Declare a variable for the calculation operator and make the calculation dependent on that variable.

1. Declare a variable for the calculation operator (suggested name: **op**). The variable needs a type of fixed length to hold one of the following values: “+”, “-”, “*”, or “/”.

   a) Add the highlighted code:

```
DATA number1 TYPE i.
  DATA number2 TYPE i.

  DATA result  TYPE p LENGTH 8 DECIMALS 2.

 DATA op      TYPE c LENGTH 1.
```

2. Directly after the value assignment for variables **number1** and **number2**, add a value assignment to set the new variable **op** to value “/”.

   a) Add the highlighted code:

```
  number1 = 123.
  number2 = 0.
  op      = '/'.
```

3. Use a suitable control structure to fill the variable **result** with different arithmetic expressions, depending on the value of the variable **op**.

   a) Add the highlighted code:

```
CASE op.
   WHEN '+'.
     result = number1 + number2.
   WHEN '-'.
     result = number1 - number2.
   WHEN '*'.
     result = number1 * number2.
   WHEN '/'.
     result = number1 / number2.
ENDCASE.
```

4. Adjust the string template to use the content of **op**, rather than the literal character “/”.

   a) Adjust the string template as follows:

```
output = |{ number1 } { op } { number2 } = { result }|.
```

5. Activate and test the application with different values for **number1**, **number2**, and **op**.

   What happens if you set **op** to value “%”?

   The result is always zero (0.00) because none of the four WHEN branches is executed.

   a) Proceed as you have done in previous exercises.

Task 3: Handle Invalid Operator

Handle an invalid value of **op**. If **op** contains a value other than “+”, “-”, “*”, or “/”, write an error message to the console instead of the calculation and the result value.

1. In the CASE control structure, add a branch that is executed for any other values of **op**.

a) Add the highlighted code:

```
CASE op.
      WHEN '+'.
        result = number1 + number2.
      WHEN '-'.
        result = number1 - number2.
      WHEN '*'.
        result = number1 * number2.
      WHEN '/'.
        result = number1 / number2.
      WHEN OTHERS.

    ENDCASE.
```

2. In this branch, fill **output** with a suitable text to describe the error situation.

Note:
We suggest that at this point, you replace the inline declaration of the variable **output** with an explicit declaration at the beginning of the method.

a) At the beginning of the method, add the following code:

```
    DATA output TYPE string.
```

b) Replace the inline declaration `DATA(output)` with the explicitly declared variable **output**.

c) After the `WHEN OTHERS.` statement, add the following code:

```
    output = |'{ op }' is not a valid operator!|.
```

3. Use a suitable control structure to make sure the **output** variable is only filled with the calculation result if it is not already filled with an error text.

a) Add the highlighted code:

```
    IF output IS INITIAL.  "no error so far
      output = |{ number1 } { op } { number2 } = { result }|.
    ENDIF.
```

4. Activate and test the application with different values for **number1**, **number2**, and **op**.

What happens if you set **number1** to **1**, **number2** to **0**, and **op** to **/**?

The application terminates with exception CX_SY_ZERODIVIDE.

a) Proceed as you have done in previous exercises.

Task 4: Handle Division by Zero

Prevent the program termination by handling the exception.

1. Inside the branch for division (**op** has value "/"), surround the calculation with a pair of TRY and ENDTRY statements.

a) Add the highlighted code:

```
    WHEN '/'.

       TRY.
           result = number1 / number2.

       ENDTRY.

     WHEN OTHERS.
```

2. Before the ENDTRY statement, add a CATCH-block for the exception that is raised in case of a division by zero error.

   a) Add the highlighted code:

```
       TRY.
           result = number1 / number2.

         CATCH cx_sy_zerodivide.
        ENDTRY.
```

3. In the CATCH-block, fill the **output** variable with a suitable error text.

   a) Add the highlighted code:

```
        TRY.
           result = number1 / number2.

         CATCH cx_sy_zerodivide.
           output = |Division by zero is not defined|.
       ENDTRY.
```

4. Activate and test the application with different values for **number1**, **number2**, and **op**. In particular, test the handling of the error situations: invalid operator and division by zero.

   a) Proceed as you have done before.

   b) Compare your code to the following extract from the model solution:

```
METHOD if_oo_adt_classrun~main.

* Declarations
**************************

    DATA number1 TYPE i.
    DATA number2 TYPE i.

    DATA result  TYPE p LENGTH 8 DECIMALS 2.

    DATA op      TYPE c LENGTH 1.

    DATA output TYPE string.

* Input Values
**************************

    number1 = 123.
    number2 = 0.
    op      = '/'.

* Calculation
```

```
*************************
    CASE op.
      WHEN '+'.
        result = number1 + number2.
      WHEN '-'.
        result = number1 - number2.
      WHEN '*'.
        result = number1 * number2.
      WHEN '/'.

        TRY.
            result = number1 / number2.
          CATCH cx_sy_zerodivide.
            output = |Division by zero is not defined|.
        ENDTRY.

      WHEN OTHERS.

        output = |'{ op }' is not a valid operator!|.

    ENDCASE.

    IF output IS INITIAL.  "no error so far

      output = |{ number1 } { op } { number2 } = { result }|.

    ENDIF.

* Output
*************************

    out->write( output ).

  ENDMETHOD.
```

Unit 2
Exercise 6

Work with Simple Internal Tables

In this exercise, you use iterations and simple internal tables to calculate and display the Fibonacci numbers.

Note:
The Fibonacci numbers are a sequence of numbers in which each number is the sum of the two preceding ones. The sequence starts with 0 and 1, and each subsequent number is the sum of the two previous numbers. The sequence has various mathematical properties, patterns, and applications in fields such as mathematics, computer science, nature, and art.

Template:

none

Solution:

/LRN/CL_S4D400_BTS_ITERATE (global Class)

Task 1: Calculate the Numbers

In your own package, create a new global class. Define a constant for the number of iterations and an internal table for the Fibonacci numbers. Then calculate the Fibonacci numbers in an iteration and store them in the internal table.

1. Create a new global class that implements the interface **`IF_OO_ADT_CLASSRUN`** (suggested name: **`ZCL_##_ITERATE`**, where ## stands for your group number).
2. In the *IF_OO_ADT_CLASSRUN~MAIN* method, define a constant for the number of iterations (suggested name: **`count`**) with a small value, for example **`20`**.
3. Declare a simple internal table to store the Fibonacci numbers (suggested name: **`numbers`**).
4. Implement an iteration that is executed **`max_count`** times.
5. Inside the iteration, implement a case distinction based on the built-in iteration counter: In the first iteration, add the value **`0`** to the internal table **`numbers`**. In the second iteration, add the value **`1`** to the end of internal table **`numbers`**.

Hint:
The built-in iteration counter is system field **`sy-index`**.

6. Add a branch that is executed for all other values. In this branch, calculate the new entry as the sum of the two preceding entries in **`numbers`**.

Hint:
Access the preceding entries using index **`sy-index - 1`** and **`sy-index - 2`**.

Task 2: Prepare a Formatted Output

In a loop over the internal table, prepare a formatted output that lists each Fibonacci number with its respective sequential number. Prepare the output in another internal table of row type *string*.

1. At the beginning of the method **`if_oo_adt_classrun~main`**, declare an internal table of row type *string* (suggested name: **`output`**).
2. Implement a loop over the internal table **`numbers`** to read the Fibonacci numbers one by one into a variable **`number`**.

Hint:
Use an inline declaration for **`number`**.

3. Declare an integer variable (suggested name: **`counter`**), that you set to zero before the loop and increase by 1 in each iteration.

Hint:
You can use an inline declaration for **`counter`**.

4. In each iteration, add a new row to the internal table **`output`** which contains the content of **`counter`** (I4 characters wide, left-justified), a colon (:) and the content of **`number`** (10 characters wide, right justified).

Hint:
Use a string template with suitable format options to adjust the format.

Which format options are used to set the width and the alignment?

Task 3: Output and Test

Write the result to the console. Then activate and test your class as a console app.

1. At the end of the method, call the method **`out->write`** to write the content of **`output`** to the console. Supply the importing parameter **`name`** with a suitable caption.
2. Activate the class.
3. Execute your class as a console app.

Unit 2
Solution 6

Work with Simple Internal Tables

In this exercise, you use iterations and simple internal tables to calculate and display the Fibonacci numbers.

Note:
The Fibonacci numbers are a sequence of numbers in which each number is the sum of the two preceding ones. The sequence starts with 0 and 1, and each subsequent number is the sum of the two previous numbers. The sequence has various mathematical properties, patterns, and applications in fields such as mathematics, computer science, nature, and art.

Template:

none

Solution:

/LRN/CL_S4D400_BTS_ITERATE (global Class)

Task 1: Calculate the Numbers

In your own package, create a new global class. Define a constant for the number of iterations and an internal table for the Fibonacci numbers. Then calculate the Fibonacci numbers in an iteration and store them in the internal table.

1. Create a new global class that implements the interface **`IF_OO_ADT_CLASSRUN`** (suggested name: **`ZCL_##_ITERATE`**, where ## stands for your group number).
   a) Choose *File* → *New* → *ABAP Class*.
   b) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_ITERATE`**, where ## stands for your group number.
   c) Enter a description.
   d) In the *Interfaces* group box, choose *Add*.
   e) Enter **`IF_OO_ADT_CLASSRUN`**. When the interface appears in the hit list, double-click it to add it to the class definition.
   f) Choose *Next*.
   g) Select your transport request and choose *Finish*.
2. In the *IF_OO_ADT_CLASSRUN~MAIN* method, define a constant for the number of iterations (suggested name: **`count`**) with a small value, for example **`20`**.
   a) Add the following code:

```
CONSTANTS max_count TYPE i VALUE 20.
```

3. Declare a simple internal table to store the Fibonacci numbers (suggested name: **numbers**).

   a) Add the following code:

```
DATA numbers TYPE TABLE OF i.
```

4. Implement an iteration that is executed **max_count** times.

   a) After the declarations, add the following code:

```
DO max_count TIMES.

ENDDO.
```

5. Inside the iteration, implement a case distinction based on the built-in iteration counter: In the first iteration, add the value **0** to the internal table **numbers**. In the second iteration, add the value **1** to the end of internal table **numbers**.

Hint:
The built-in iteration counter is system field **sy-index**.

   a) Between DO and ENDDO, add the following code:

```
CASE sy-index.
   WHEN 1.
     APPEND 0 TO numbers.
   WHEN 2.
     APPEND 1 TO numbers.
ENDCASE.
```

6. Add a branch that is executed for all other values. In this branch, calculate the new entry as the sum of the two preceding entries in **numbers**.

Hint:
Access the preceding entries using index **sy-index - 1** and **sy-index - 2**.

   a) Add the highlighted code:

```
CASE sy-index.
   WHEN 1.
     APPEND 0 TO numbers.
   WHEN 2.
     APPEND 1 TO numbers.
   WHEN OTHERS.
       APPEND numbers[  sy-index - 2 ]
            + numbers[  sy-index - 1 ]
           TO numbers.
ENDCASE.
```

Task 2: Prepare a Formatted Output

In a loop over the internal table, prepare a formatted output that lists each Fibonacci number with its respective sequential number. Prepare the output in another internal table of row type *string*.

1. At the beginning of the method **`if_oo_adt_classrun~main`**, declare an internal table of row type *string* (suggested name: **`output`**).
2. Implement a loop over the internal table **`numbers`** to read the Fibonacci numbers one by one into a variable **`number`**.

Hint:
Use an inline declaration for **`number`**.

a) At the end of the method, add the following code:

```
LOOP AT numbers INTO DATA(number).

ENDLOOP.
```

3. Declare an integer variable (suggested name: **`counter`**), that you set to zero before the loop and increase by 1 in each iteration.

Hint:
You can use an inline declaration for **`counter`**.

a) Add the highlighted code:

```
DATA(counter) = 0.
LOOP AT numbers INTO DATA(number).

   counter = counter + 1.

ENDLOOP.
```

4. In each iteration, add a new row to the internal table **`output`** which contains the content of **`counter`** (I4 characters wide, left-justified), a colon (:) and the content of **`number`** (10 characters wide, right justified).

Hint:
Use a string template with suitable format options to adjust the format.

Which format options are used to set the width and the alignment?

Format options WIDTH and ALIGN.

a) Add the highlighted code:

```
DATA(counter) = 0.
LOOP AT numbers INTO DATA(number).
```

```
    counter = counter + 1.

    APPEND |{ counter WIDTH = 4 }: { number WIDTH = 10 ALIGN =
RIGHT }|
        TO output.

  ENDLOOP.
```

Task 3: Output and Test

Write the result to the console. Then activate and test your class as a console app.

1. At the end of the method, call the method **`out->write`** to write the content of **`output`** to the console. Supply the importing parameter **`name`** with a suitable caption.
   a) After `ENDLOOP.`, add the following code:

```
out->write(
        data   = output
        name   = |The first { max_count } Fibonacci Numbers|
               ) .
```

2. Activate the class.
   a) Press **`Ctrl + F3`** to activate the class.
3. Execute your class as a console app.
   a) Press **`F9`** to execute the class as console app.

Unit 2
Exercise 7

Debug ABAP Code

In this exercise, you analyze ABAP code using the ABAP debugger.

Template:

/LRN/CL_S4D400_BTT_DEBUG (global Class)

Solution:

None, the class remains unchanged.

Task 1: Preparation

Copy the class that you want to debug.

1. Open the source code of global class /LRN/CL_S4D400_BTT_DEBUG in the ABAP editor.
2. Link the *Project Explorer* view with the editor.
3. Copy class /LRN/CL_S4D400_BTT_DEBUG to a class in your own package (suggested name: ZCL_##_DEBUG, where ## stands for your group number).
4. Activate and test the class.
5. Check the output in the *Console* view.

Task 2: Analyze the Starting Values

Enter the debugger at the first executable statement in method **`if_oo_adt_classrun~main( )`** and analyze the values of some variable and constant data objects.

1. Set a breakpoint at the first statement that does not define a type or declare a data object.

Hint:
TYPES defines a data type, CONSTANTS declares a constant data object, DATA declares a variable data object.

2. Run the class as a console app and enter the debugger.
3. Display the value of data object **`loan_remaining`** and **`loan_total`** in the *Variables* view.

Task 3: Control Program Execution

Set break points and watch points. Execute single steps or resume execution until the next break point or watch point is reached. Supervise the value changes of the data objects.

1. Execute a single step to debug the value assignment in the current line.
2. Display the content of data object **`spec_repay_mode`**. Then execute another single step to see which WHEN branch of the CASE - control structure is executed.

3. Set a watch point for variable **`loan_remaining`** and resume program execution. Where does the program execution stop again?
4. Display the content of data object **`repayment_plan`**. Then execute another single step to see how it is filled with the APPEND statement.

Note:
Because **`repayment_plan`** is an internal table, it not only displays in the *Variables* view but also in the *ABAP Internal Table (Debugger)* view below the editor.

5. Inspect the string template in the APPEND statement and relate it to the resulting first row in internal table **`repayment_plan`**. Display the content of the data objects that appear in the embedded expressions.
6. Set another watch point for variable **`repayment_plan`** and resume program execution for a few times. Whenever execution reaches one of the watch points, analyze the value of **`loan_remaining`** and new rows are added to **`repayment_plan`**.
7. After a while, set a statement break point for all EXIT statements and delete the two watch points.
8. Resume program execution until you reach the first EXIT statement. Deactivate the statement breakpoint for the **`EXIT`** statement. Then execute single steps until you reach the output part of the program.
9. Open the *Console* view. Execute the remaining program and pursue the output on the *console* view.

Note:
Do not press **`F5`** to debug the output. Use **`F6`** instead.

Note:
As you will learn later in the course `out->write( ... )` is not an ABAP statement but a reusable code block that consists of many ABAP statements. By pressing **`F5`** you *Step Into* this code to analyze it in detail. With **`F6`** you *Step Over* the code block, treating it like a single statement.

10. When the application is terminated, do not forget to switch back to the *ABAP perspective*.

Unit 2
Solution 7

Debug ABAP Code

In this exercise, you analyze ABAP code using the ABAP debugger.

Template:

/LRN/CL_S4D400_BTT_DEBUG (global Class)

Solution:

None, the class remains unchanged.

Task 1: Preparation
Copy the class that you want to debug.

1. Open the source code of global class /LRN/CL_S4D400_BTT_DEBUG in the ABAP editor.
   a) In the *Eclipse* toolbar, choose *Open ABAP Development Object*. Alternatively, press **`Ctrl + Shift + A`**.
   b) Enter **`/LRN/CL_S4D400_BTT`** as search string.
   c) From the list of development objects choose */LRN/CL_S4D400_BTT_DEBUG*, then choose *Ok*.
2. Link the *Project Explorer* view with the editor.
   a) In the *Project Explorer* toolbar, find the *Link with Editor* button. If it is not yet pressed, choose it. As a result, the development object, that is open in the editor, should be highlighted in the *Project Explorer*.
3. Copy class /LRN/CL_S4D400_BTT_DEBUG to a class in your own package (suggested name: ZCL_##_DEBUG, where ## stands for your group number).
   a) In the *Project Explorer* view, right-click class /LRN/CL_S4D400_BTT_DEBUG to open the content menu.
   b) From the context menu, choose *Duplicate ...*.
   c) Enter the name of your package (**`ZS4D400_##`**, where ## stands for your group number) and the name for the new class (**`ZCL_##_DEBUG`**), then choose *Next*.
   d) Confirm the transport request and choose *Finish*.
4. Activate and test the class.
   a) Press **`Ctrl + F3`** to activate the class.
   b) Press **`F9`** to run the class.
5. Check the output in the *Console* view.
   a) Check the *Console* view that should have opened as a new tab below the editor view.
   b) If the *Console* view is not visible, open it by choosing *Window* → *Show view* → *Other*. Double-click *Console* in the hit list.

Task 2: Analyze the Starting Values

Enter the debugger at the first executable statement in method **`if_oo_adt_classrun~main( )`** and analyze the values of some variable and constant data objects.

1. Set a breakpoint at the first statement that does not define a type or declare a data object.

Hint:
TYPES defines a data type, CONSTANTS declares a constant data object, DATA declares a variable data object.

   a) Double-click the left-hand margin of the editor next to the line `loan_remaining = loan_total.` to set a break point.

2. Run the class as a console app and enter the debugger.
   a) Press **F9** to run the class.
   b) If you are asked whether you want to switch to the Debug perspective, mark *Remember my decision* and choose *OK*.
3. Display the value of data object **`loan_remaining`** and **`loan_total`** in the *Variables* view.
   a) In the current code line (the one with a green background) double-click `loan_remaining`.
   b) In the same code line, double-click **`loan_total`**.

Task 3: Control Program Execution

Set break points and watch points. Execute single steps or resume execution until the next break point or watch point is reached. Supervise the value changes of the data objects.

1. Execute a single step to debug the value assignment in the current line.
   a) In the toolbar, choose *Step Into (F5)* or press **F5**.
   b) Check that the value of **`loan_remaining`** changed from **`0..00`** to **`5000.00`**.
2. Display the content of data object **`spec_repay_mode`**. Then execute another single step to see which WHEN branch of the CASE - control structure is executed.
   a) In the next code line, double-click **`spec_repay_mode`**.
   b) Press **F5** to see that the program jumps to code line `WHEN 'Q'.`.
3. Set a watch point for variable **`loan_remaining`** and resume program execution. Where does the program execution stop again?
   a) In the *Variables* view, right-click on *LOAN_REMAINING* and choose *Set Watchpoint*.
   b) In the toolbar, choose *Resume (F8)* or press **F8**.
   c) Program execution stops immediately after code line `loan_remaining = loan_remaining - repayment_month.`, also to be precise, in the next code line with executable code.
4. Display the content of data object **`repayment_plan`**. Then execute another single step to see how it is filled with the APPEND statement.

> Note:
> Because **repayment_plan** is an internal table, it not only displays in the *Variables* view but also in the *ABAP Internal Table (Debugger)* view below the editor.

a) Double-click on **repayment_plan** at the end of the APPEND statement.

b) Press **F5** to see how **repayment_plan** is filled with a first row.

5. Inspect the string template in the APPEND statement and relate it to the resulting first row in internal table **repayment_plan**. Display the content of the data objects that appear in the embedded expressions.
   a) Double-click the data objects that appear between the curly brackets to display their contents.
6. Set another watch point for variable **repayment_plan** and resume program execution for a few times. Whenever execution reaches one of the watch points, analyze the value of **loan_remaining** and new rows are added to **repayment_plan**.
   a) In the *Variables* view, right-click on *REPAYMENT_PLAN* and choose *Set Watchpoint*.
   b) Press **F8** to resume program execution.
   c) Analyze the data objects in the *Variables* view and the *ABAP Internal Table (Debugger)* view.
7. After a while, set a statement break point for all EXIT statements and delete the two watch points.
   a) Navigate to the *Breakpoints* view.

   > Hint:
   > You find the *Breakpoints* view next to the *Variables* view.

   b) In the toolbar of the *Breakpoints* view, expand the dropdown button on the very left and choose *Add Statement Breakpoint ...*.
   c) On the dialog window that appears, enter **EXIT** as search string, click on the value *EXIT* in the hitlist, and choose *OK*.
   d) In the *Breakpoints* view, right-click the watch point *LOAN_REMAINING* and choose *Remove*.
   e) In the *Breakpoints* view, right-click the watch point *REPAYMENT_PLAN* and choose *Remove*.
8. Resume program execution until you reach the first EXIT statement. Deactivate the statement breakpoint for the **EXIT** statement. Then execute single steps until you reach the output part of the program.
   a) Press **F8** to resume program execution.
   b) Switch to the *Breakpoints* view and deselect the line that says **EXIT [Statement]**.
   c) Press **F5** until you reach the first code line that starts with `out->write(`.

9. Open the *Console* view. Execute the remaining program and pursue the output on the *console* view.

Note:
Do not press **F5** to debug the output. Use **F6** instead.

Note:
As you will learn later in the course `out->write( ... )` is not an ABAP statement but a reusable code block that consists of many ABAP statements. By pressing **F5** you *Step Into* this code to analyze it in detail. With **F6** you *Step Over* the code block, treating it like a single statement.

a) Press **F6** several times until you reach the end of the application.

b) Analyze the addition output on the *Console* view and compare it to the string template in the previous code line.

10. When the application is terminated, do not forget to switch back to the *ABAP perspective*.

a) Choose *ABAP* on the very right of the Eclipse toolbar.

Unit 3
Exercise 8

Define a Local Class

In this exercise, you define a local class inside of a global class.

Template:

none

Solution:

/LRN/CL_S4D400_CLS_LOCAL_CLASS (global Class)

Task 1: Create a Global Class

In your own package, create a new ABAP class.

1. Create a new global class that implements interface *IF_OO_ADT_CLASSRUN* (suggested name: **`ZCL_##_LOCAL_CLASS`**, where ## stands for your group number).

Task 2: Define a Local Class

Define a local class **`lcl_connection`** and declare some public attributes.

1. Inside the global class, create a new local class **`lcl_connection`**. Use code completion to generate the code, but make sure you remove the `create private` addition from the generated code.
2. In local class **`lcl_connection`**, declare the following **public** attributes:

   Table 1: Attributes

Attribute Name	Scope	Data Type
`carrier_id`	instance	/DMO/CARRIER_ID
`connection_id`	instance	/DMO/CONNECTION_ID
`conn_counter`	static	I

3. Activate the class.

Note:
Because the **`if_oo_adt_classrun~main`** method does not contain executable code, yet, there is nothing to test or debug at this point.

Unit 3
Solution 8

Define a Local Class

In this exercise, you define a local class inside of a global class.

Template:

none

Solution:

/LRN/CL_S4D400_CLS_LOCAL_CLASS (global Class)

Task 1: Create a Global Class
In your own package, create a new ABAP class.

1. Create a new global class that implements interface *IF_OO_ADT_CLASSRUN* (suggested name: **`ZCL_##_LOCAL_CLASS`**, where ## stands for your group number).
   a) Choose *File* → *New* → *ABAP Class*.
   b) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_LOCAL_CLASS`**, where **`##`** is your group number. Enter a description.
   c) In the *Interfaces* group box, choose *Add*.
   d) Enter **`IF_OO_ADT_CLASSRUN`**. When the interface appears in the hit list, double-click it to add it to the class definition.
   e) Choose *Next*.
   f) Select your transport request and choose *Finish*.

Task 2: Define a Local Class
Define a local class **`lcl_connection`** and declare some public attributes.

1. Inside the global class, create a new local class **`lcl_connection`**. Use code completion to generate the code, but make sure you remove the `create private` addition from the generated code.
   a) Switch to the *Local Types* tab.
   b) Type **`lcl`** into the editor and press **`Ctrl + Space`**.
   c) Double-click *lcl - class* in the pop-up.
   d) While there is still a frame visible around **`lcl`** in the line `class lcl definition create private.`, complete the name of the class to **`lcl_connection`**.
   e) Delete the words `create private` at the end of the code row `class lcl_connection definition.`.
2. In local class **`lcl_connection`**, declare the following **public** attributes:

Table 1: Attributes

Attribute Name	Scope	Data Type
carrier_id	instance	/DMO/CARRIER_ID
connection_id	instance	/DMO/CONNECTION_ID
conn_counter	static	I

a) After line `PUBLIC SECTION.` and before line `PROTECTED SECTION.`, add the following code:

```
DATA carrier_id    TYPE /dmo/carrier_id.
DATA connection_id TYPE /dmo/connection_id.

CLASS-DATA conn_counter TYPE i.
```

3. Activate the class.

Note:
Because the **if_oo_adt_classrun~main** method does not contain executable code, yet, there is nothing to test or debug at this point.

a) Press **Ctrl + F3** to activate the class.

b) Compare your code on tab *Local Types* to the following extract from the model solution:

```
*"* use this source file for the definition and implementation of
*"* local helper classes, interface definitions and type
*"* declarations

CLASS lcl_connection DEFINITION.

  PUBLIC SECTION.

    DATA carrier_id    TYPE /dmo/carrier_id.
    DATA connection_id TYPE /dmo/connection_id.

    CLASS-DATA conn_counter TYPE i.

  PROTECTED SECTION.
  PRIVATE SECTION.

ENDCLASS.

CLASS lcl_connection IMPLEMENTATION.

ENDCLASS.
```

Unit 3
Exercise 9

Create and Manage Instances

In this exercise, you create and manage instances of your local class.

Template:

/LRN/CL_S4D400_CLS_LOCAL_CLASS (global Class)

Solution:

/LRN/CL_S4D400_CLS_INSTANCES (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise you can skip this task and continue editing your class **`ZCL_##_LOCAL_CLASS`**.

1. Open the source code of global class **`/LRN/CL_S4D400_CLS_LOCAL_CLASS`** in the ABAP editor.
2. Link the *Project Explorer* view with the editor.
3. Copy class **`/LRN/CL_S4D400_CLS_LOCAL_CLASS`** to a class in your own package (suggested name: **`ZCL_##_INSTANCES`**, where ## stands for your group number).

Task 2: Create an Instance
In the **`main`** method of your global class, declare a reference variable and create an instance of your local class.

1. In method **`if_oo_adt_classrun~main`** of your global class, declare a reference variable (suggested name: **`connection`**) and type it with your local class **`lcl_connection`**.
2. Create an instance of the class and set the attributes **`carrier_id`** and **`connection_id`** (Suggested values: “LH” for attribute **`carrier_id`** and “0400” for attribute **`connection_id`**.
3. Activate the class and use the debugger to analyze step by step what happens.

Task 3: Manage Several Instances
Create more instances of your local class and store the references to these instances in an internal table.

1. In method **`if_oo_adt_classrun~main( )`** of your global class, declare an internal table with the line type `TYPE REF TO lcl_connection` (suggested name: **`connections`**).
2. After instantiating the class and setting the attributes, append the object reference to the internal table.
3. Create two more instances of class **`lcl_connection`**, set their instance attributes to different values than in the first instance and append the references to the new instances to internal table **`connections`**. Use the same reference variable **`connection`** for all three instances.

4. Activate the class and use the debugger to analyze step by step what happens.

Unit 3
Solution 9

Create and Manage Instances

In this exercise, you create and manage instances of your local class.

Template:

/LRN/CL_S4D400_CLS_LOCAL_CLASS (global Class)

Solution:

/LRN/CL_S4D400_CLS_INSTANCES (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise you can skip this task and continue editing your class **`ZCL_##_LOCAL_CLASS`**.

1. Open the source code of global class **`/LRN/CL_S4D400_CLS_LOCAL_CLASS`** in the ABAP editor.
   a) In the *Eclipse* toolbar, choose *Open ABAP Development Object*. Alternatively, press **`Ctrl + Shift + A`**.
   b) Enter **`/LRN/CL_S4D400_CLS`** as search string.
   c) From the list of development objects choose */LRN/CL_S4D400_CLS_LOCAL_CLASS*, then choose *OK*.
2. Link the *Project Explorer* view with the editor.
   a) In the *Project Explorer* toolbar, find the *Link with Editor* button. If it is not yet pressed, choose it. As a result, the development object, that is open in the editor, should be highlighted in the *Project Explorer*.
3. Copy class **`/LRN/CL_S4D400_CLS_LOCAL_CLASS`** to a class in your own package (suggested name: **`ZCL_##_INSTANCES`**, where ## stands for your group number).
   a) In the *Project Explorer* view, right-click class **`/LRN/CL_S4D400_CLS_LOCAL_CLASS`** to open the content menu.
   b) From the context menu, choose *Duplicate ...*.
   c) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_INSTANCES`**, where ## stands for your group number.
   d) Adjust the description and choose *Next*.
   e) Confirm the transport request and choose *Finish*.

Task 2: Create an Instance
In the **`main`** method of your global class, declare a reference variable and create an instance of your local class.

1. In method **`if_oo_adt_classrun~main`** of your global class, declare a reference variable (suggested name: **`connection`**) and type it with your local class **`lcl_connection`**.

a) Switch to the *Global Class* tab.

b) Add the highlighted code:

```
METHOD if_oo_adt_classrun~main.

  DATA connection TYPE REF TO lcl_connection.

ENDMETHOD.
```

2. Create an instance of the class and set the attributes **carrier_id** and **connection_id** (Suggested values: “LH” for attribute **carrier_id** and “0400” for attribute **connection_id**.

a) Add the highlighted code:

```
DATA connection TYPE REF TO lcl_connection.

connection = new #( ).

connection->carrier_id    = 'LH'.
connection->connection_id = '0400'.
```

Hint:
After typing the component selector (->), press **Strg + Space** to choose the attribute names from a suggestion list.

3. Activate the class and use the debugger to analyze step by step what happens.

a) Press **Ctrl + F3** to activate the class.

b) Double-click the left-hand margin of the editor next to the line `connection = new #( ).` to set a break point.

c) Press **F9** to run the class.

d) Double-click the word **connection** to display the content of reference variable **connection**.

e) Press **F5** to go one step further in the debugger. Check that the value of **connection** has changed.

f) In the *Variables* view, expand the branch **connection** to display the initial attributes of the class.

g) Press **F5** to go one step further in the debugger. Check that the value of **carrier_id** has changed.

h) Press *F5* again. Check that the value of **Connection_ID** has changed.

Task 3: Manage Several Instances

Create more instances of your local class and store the references to these instances in an internal table.

1. In method **if_oo_adt_classrun~main( )** of your global class, declare an internal table with the line type `TYPE REF TO lcl_connection` (suggested name: **connections**).

a) Add the highlighted code:

```
DATA connection TYPE REF TO lcl_connection.
DATA connections TYPE TABLE OF REF TO lcl_connection.

connection = new #( ).

connection->carrier_id    = 'LH'.
connection->connection_id = '0400'.
```

2. After instantiating the class and setting the attributes, append the object reference to the internal table.

a) Add the highlighted code:

```
connection = NEW #( ).

connection->carrier_id = 'LH'.
connection->connection_id = '0400'.

APPEND connection TO connections.
```

3. Create two more instances of class **`lcl_connection`**, set their instance attributes to different values than in the first instance and append the references to the new instances to internal table **`connections`**. Use the same reference variable **`connection`** for all three instances.

a) Add the highlighted code:

```
connection = NEW #(  ).

connection->carrier_id    = 'LH'.
connection->connection_id = '0400'.

APPEND connection TO connections.

connection = NEW #(  ).

connection->carrier_id    = 'AA'.
connection->connection_id = '0017'.

APPEND connection TO connections.

connection = NEW #(  ).

connection->carrier_id    = 'SQ'.
connection->connection_id = '0001'.

APPEND connection TO connections.
```

4. Activate the class and use the debugger to analyze step by step what happens.

a) Press **`Ctrl + F3`** to activate the class.

b) Double-click the left-hand margin of the editor next to the first line `connection = new #( )`. to remove the break point.

c) Double-click the left-hand margin of the editor next to the first line `APPEND connection TO connections`. to set a new break point.

d) Press **F9** to run the class.

e) Double-click the word **connection** to display the content of reference variable **connection**.

f) Double-click the word **connections** to display the content of internal table **connections**.

g) Press **F5** to go one step further in the debugger. Check that the content of **connections** has changed.

h) In the *Variables* view, expand the branch **connections** to display content of the internal table.

i) Press **F5** several times until you reach the end of the program. While you do so, check the content of the internal table.

j) Compare your code on tab *Global Class* to the following extract from the model solution:

```
METHOD if_oo_adt_classrun~main.

    DATA connection TYPE REF TO lcl_connection.
    DATA connections TYPE TABLE OF REF TO lcl_connection.

* First Instance
**********************************************************************

    connection = NEW #(  ).

    connection->carrier_id    = 'LH'.
    connection->connection_id = '0400'.

    APPEND connection TO connections.

* Second Instance
**********************************************************************
    connection = NEW #(  ).

    connection->carrier_id    = 'AA'.
    connection->connection_id = '0017'.

    APPEND connection TO connections.

* Third Instance
**********************************************************************
    connection = NEW #(  ).

    connection->carrier_id    = 'SQ'.
    connection->connection_id = '0001'.

    APPEND connection TO connections.

  ENDMETHOD.
```

Unit 3
Exercise 10

Define and Call Methods

In this exercise, you create and manage instances of your local class.

Template:

/LRN/CL_S4D400_CLS_INSTANCES (global Class)

Solution:

/LRN/CL_S4D400_CLS_METHODS (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class **`ZCL_##_LOCAL_CLASS`** or your class **`ZCL_##_INSTANCES`**.

1. Open the source code of global class **`/LRN/CL_S4D400_CLS_INSTANCES`** in the ABAP editor.
2. Link the *Project Explorer* view with the editor.
3. Copy class **`/LRN/CL_S4D400_CLS_INSTANCES`** to a class in your own package (suggested name: **`ZCL_##_METHODS`**, where ## stands for your group number).

Task 2: Define Methods
In your local class **`lcl_connection`**, define instance methods **`get_output`** and **`set_attributes`**.

1. Navigate to the definition of the local class **`lcl_connection`**.
2. Add a method **`get_output`** to the public section of the class. It should have one returning parameter **`r_output`**of global table type**`STRING_TABLE`**.

> Note:
> Remember to surround the name of the returning parameter with `VALUE( )`.

3. Add a method **`set_attributes`** to the public section of the class. It should have one importing parameter for each instance attribute of the class. Use the same types for the parameters that you used to type the attributes. To distinguish the parameters from the attributes, add prefix **`i_`** to the parameter names. In addition, the method should raise exception **`CX_ABAP_INVALID_VALUE`**.

Task 3: Implement Methods

1. Use a quick fix to add the method implementations to the class.
2. Implement the method **`get_output`**. Append some string templates to returning parameter **`r_output`**. In the string templates, use embedded expressions to add the values of attributes **`carrier_id`** and **`connection_id`**.

3. Implement the method **set_attributes**. If either of the two importing parameters is empty (`IS INITIAL`), raise exception **CX_ABAP_INVALID_VALUE**. Otherwise, fill the two instance attributes with the values of the importing parameters.
4. Activate your class.

Task 4: Call a Functional Method

In the **main** method of your global class, call the functional method **get_output**.

1. Switch to the implementation of method **if_oo_adt_classrun~main** in the global class.
2. At the end of the method, add a loop over the internal table **connections** with reference variable **connection** as a work area.
3. In the loop, call functional method **get_output** for each instance in turn and write the result to the console.

Hint:
You can use the call of method **get_output** directly as input for **out->write( ).**

4. Activate the class. Execute it as a console app and analyze the console output.

Task 5: Use Code Completion and Handle Exceptions

Use code completion to call the method **set_attributes** and handle the exception the method raises.

1. For the first instance of class **lcl_connection**, replace the direct access to attributes **carrier_id** and **connection_id** with a call of the instance method **set_attributes( )**. Use code completion to insert the full signature of the method.

Hint:
Press **Ctrl + Space** after you typed the component select (->) to display a list of available attributes and methods. Choose the method and then press **Shift + Enter** to insert the full signature of the method.

2. Handle the exception. Surround the method call with `TRY.` and `ENDTRY.`. Add a CATCH-Block to handle exception **CX_ABAP_INVALID_VALUE**. Make sure the new instance is only added to internal table **connections** if the method call was successful.
3. Repeat the previous step for the other two instances of class **lcl_connection**.
4. Activate the class. Execute it and debug the method calls.

Unit 3
Solution 10

Define and Call Methods

In this exercise, you create and manage instances of your local class.

Template:

/LRN/CL_S4D400_CLS_INSTANCES (global Class)

Solution:

/LRN/CL_S4D400_CLS_METHODS (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class **`ZCL_##_LOCAL_CLASS`** or your class **`ZCL_##_INSTANCES`**.

1. Open the source code of global class **`/LRN/CL_S4D400_CLS_INSTANCES`** in the ABAP editor.
   a) In the *Eclipse* toolbar, choose *Open ABAP Development Object*. Alternatively, press **`Ctrl + Shift + A`**.
   b) Enter **`/LRN/CL_S4D400_CLS`** as search string.
   c) From the list of development objects choose */LRN/CL_S4D400_CLS_INSTANCES*, then choose *OK*.
2. Link the *Project Explorer* view with the editor.
   a) In the *Project Explorer* toolbar, find the *Link with Editor* button. If it is not yet pressed, choose it. As a result, the development object, that is open in the editor, should be highlighted in the *Project Explorer*.
3. Copy class **`/LRN/CL_S4D400_CLS_INSTANCES`** to a class in your own package (suggested name: **`ZCL_##_METHODS`**, where ## stands for your group number).
   a) In the *Project Explorer* view, right-click class **`/LRN/CL_S4D400_CLS_INSTANCES`** to open the content menu.
   b) From the context menu, choose *Duplicate ...*.
   c) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_METHODS`**, where ## stands for your group number.
   d) Adjust the description and choose *Next*.
   e) Confirm the transport request and choose *Finish*.

Task 2: Define Methods
In your local class **`lcl_connection`**, define instance methods **`get_output`** and **`set_attributes`**.

1. Navigate to the definition of the local class **`lcl_connection`**.
   a) In the global class, choose *Local Types*.

2. Add a method **`get_output`** to the public section of the class. It should have one returning parameter **`r_output`**of global table type**`STRING_TABLE`**.

Note:
Remember to surround the name of the returning parameter with `VALUE( )`.

a) Add the highlighted code:

```
PUBLIC SECTION.

   DATA carrier_id    TYPE /dmo/carrier_id.
   DATA connection_id TYPE /DMO/Connection_id.

   CLASS-DATA conn_counter TYPE i.

   METHODS get_output
     returning
       value(r_output) type string_table.

 PROTECTED SECTION.
```

Note:
For the moment, ignore the syntax error *Implementation missing for method "GET_OUTPUT"*.

3. Add a method **`set_attributes`** to the public section of the class. It should have one importing parameter for each instance attribute of the class. Use the same types for the parameters that you used to type the attributes. To distinguish the parameters from the attributes, add prefix **`i_`** to the parameter names. In addition, the method should raise exception **`CX_ABAP_INVALID_VALUE`**.

a) Add the highlighted code:

```
PUBLIC SECTION.

   DATA carrier_id    TYPE /dmo/carrier_id.
   DATA connection_id TYPE /DMO/Connection_id.

   CLASS-DATA conn_counter TYPE i.

   METHODS set_attributes
     IMPORTING
       i_carrier_id    TYPE /dmo/carrier_id
       i_connection_id TYPE /dmo/connection_id
     RAISING
       cx_abap_invalid_value.

   METHODS get_output
     returning
       value(r_output) type string_table.

 PROTECTED SECTION.
```

Note:
For the moment, ignore the syntax error *Implementation missing for method "SET_ATTRIBUTES"*.

Task 3: Implement Methods

1. Use a quick fix to add the method implementations to the class.
   a) Position the cursor on the name of one of the methods in the editor and press **Ctrl + 1**.
   b) Double-click the suggestion *Add 2 unimplemented methods*.
2. Implement the method **get_output**. Append some string templates to returning parameter **r_output**. In the string templates, use embedded expressions to add the values of attributes **carrier_id** and **connection_id**.
   a) In the local class add the following code between the statements `METHOD get_output.` and `ENDMETHOD.`

```
  METHOD get_output.

     APPEND |------------------------------| TO r_output.
     APPEND |Carrier:     { carrier_id    }| TO r_output.
     APPEND |Connection:  { connection_id }| TO r_output.

  ENDMETHOD.
```

3. Implement the method **set_attributes**. If either of the two importing parameters is empty (`IS INITIAL`), raise exception **CX_ABAP_INVALID_VALUE**. Otherwise, fill the two instance attributes with the values of the importing parameters.
   a) Add the highlighted code:

```
METHOD set_attributes.

  IF i_carrier_id IS INITIAL OR i_connection_id IS INITIAL.
    RAISE EXCEPTION TYPE cx_abap_invalid_value.
  ENDIF.

  carrier_id    = i_carrier_id.
  connection_id = i_connection_id.

ENDMETHOD.
```

4. Activate your class.
   a) Choose **Ctrl + F3** to activate the class.

Task 4: Call a Functional Method

In the **main** method of your global class, call the functional method **get_output**.

1. Switch to the implementation of method **if_oo_adt_classrun~main** in the global class.
   a) In the global class, choose *Global Class* scroll down to the implementation of method **if_oo_adt_classrun~main**.
2. At the end of the method, add a loop over the internal table **connections** with reference variable **connection** as a work area.

a) Add the following code before `ENDMETHOD.`.

```
LOOP AT connections INTO connection.

ENDLOOP.
```

3. In the loop, call functional method **`get_output`** for each instance in turn and write the result to the console.

Hint:
You can use the call of method **`get_output`** directly as input for **`out->write( ).`**

a) Add the highlighted code:

```
LOOP AT connections INTO connection.

   out->write( connection->get_output( ) ).

ENDLOOP.
```

4. Activate the class. Execute it as a console app and analyze the console output.
   a) Press **`Ctrl + F3`** to activate the class.
   b) Press **`F9`** to run the class.
   c) Analyze the output on the *Console* window.

Task 5: Use Code Completion and Handle Exceptions

Use code completion to call the method **`set_attributes`** and handle the exception the method raises.

1. For the first instance of class **`lcl_connection`**, replace the direct access to attributes **`carrier_id`** and **`connection_id`** with a call of the instance method **`set_attributes( )`**. Use code completion to insert the full signature of the method.

Hint:
Press **`Ctrl + Space`** after you typed the component select (->) to display a list of available attributes and methods. Choose the method and then press **`Shift + Enter`** to insert the full signature of the method.

   a) Place the cursor in the next line after the first `connection = NEW #( ).`.
   b) Type `connection->` and press **`Ctrl + Space`**.
   c) Choose *set_attributes* and press **`Shift + Enter`**.
   d) Pass values to the importing parameters. Use the same literals that you used previously to set the attributes of this instance.
   e) Remove or comment the direct access to the instance attributes for this instance.

2. Handle the exception. Surround the method call with `TRY.` and `ENDTRY.`. Add a CATCH-Block to handle exception **`CX_ABAP_INVALID_VALUE`**. Make sure the new instance is only added to internal table **`connections`** if the method call was successful.

a) Uncomment the generated code line `CATCH cx_abap_invalid_value..`

b) Add `TRY.` before the method call .

c) Add `ENDTRY.` after the CATCH statement.

d) Between the CATCH statement and the ENDTRY statement add `out->write( `Method call failed` ).`

e) Move the APPEND statement up to between the method call and CATCH statement.

f) The complete method call should now look like this:

```
    TRY.
        connection->set_attributes(
          EXPORTING
            i_carrier_id    = 'LH'
            i_connection_id = '0400'
        ).

*       connection->carrier_id    = 'LH'.
*       connection->connection_id = '0400'.

        APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.
```

3. Repeat the previous step for the other two instances of class **`lcl_connection`**.

a) After this step, the implementation of method **`if_oo_adt_classrun~main`** should look similar to this:

```
METHOD if_oo_adt_classrun~main.

    DATA connection TYPE REF TO lcl_connection.
    DATA connections TYPE TABLE OF REF TO lcl_connection.

* First Instance
**********************************************************************
    connection = NEW #(  ).

    TRY.
        connection->set_attributes(
          EXPORTING
            i_carrier_id    = 'LH'
            i_connection_id = '0400'
        ).

*        connection->carrier_id    = 'LH'.
*        connection->connection_id = '0400'.

        APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.

* Second instance
**********************************************************************
```

```
    connection = NEW #(  ).

    TRY.
        connection->set_attributes(
          EXPORTING
            i_carrier_id    = 'AA'
            i_connection_id = '0017'
        ).

*        connection->carrier_id    = 'AA'.
*        connection->connection_id = '0017'.

        APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.

* Third instance
**********************************************************************
    connection = NEW #(  ).

    TRY.
        connection->set_attributes(
          EXPORTING
            i_carrier_id    = 'SQ'
            i_connection_id = '0001'
        ).

*        connection->carrier_id    = 'SQ'.
*        connection->connection_id = '0001'.

        APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.


* Output
**********************************************************************

    LOOP AT connections INTO connection.

      out->write( connection->get_output( ) ).

    ENDLOOP.

  ENDMETHOD.
```

4. Activate the class. Execute it and debug the method calls.
   a) Press **Ctrl + F3** to activate the class.
   b) Press **F9** to run the class.

Unit 3
Exercise 11

Use Private Attributes and a Constructor

In this exercise, you make the the static attribute of your class read-only and the instance attributes private. You use a constructor to set their values.

Template:

/LRN/CL_S4D400_CLS_METHODS (global Class)

Solution:

/LRN/CL_S4D400_CLS_CONSTRUCTOR (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class (**`ZCL_##_METHODS`**, **`ZCL_##_LOCAL_CLASS`** or **`ZCL_##_INSTANCES`**).

1. Copy class **`/LRN/CL_S4D400_CLS_METHODS`** to a class in your own package (suggested name: **`ZCL_##_CONSTRUCTOR`**, where ## stands for your group number).

Task 2: Make Attributes Private
Change the visibility of the attributes **`carrier_id`** and **`connection_id`** to enforce the use of the methods **`set_attributes( )`** and **`get_output( )`**.

1. Set the visibility of the attribute **`carrier_id`** to private using a quick fix.
2. Set the visibility of the attribute **`connection_id`** to private using a quick fix.

Task 3: Create Instance Constructor
Replace the public method **`set_attributes`** with an instance constructor to ensure that the attributes are set during the creation of a new instance and that they are not changed afterward.

1. Comment out the definition and implementation of the method **`set_attributes`**.
2. Add an instance constructor to the local class **`lcl_connection`** using a quick fix. Ensure that the constructor has importing parameters corresponding to attributes **`carrier_id`** and **`connection_id`**.
3. Extend the generated constructor definition. Add exception **`CX_ABAP_INVALID_VALUE`** to the definition of the method **`constructor`**.
4. Extend the generated constructor implementation. If either of the importing parameters is initial, raise exception **`CX_ABAP_INVALID_VALUE`**.
5. In the constructor implementation, add a statement to increase the value of the static attribute **`conn_counter`** by one. Make sure the statement is only executed if no exception was raised.

6. Make sure that it is not possible to change the value of **`conn_counter`** from outside the class.

Task 4: Use the Constructor

Adjust the instantiation of class **`lcl_connection`** to supply the parameters of the instance constructor and handle the exception.

1. In method **`if_oo_adt_classrun~main`**, go to the NEW #( ) expression for the first instance of **`lcl_connection`**.
2. In the NEW expression, supply the import parameters of **`constructor`**.

Hint:
You can copy the parameter passing from the call of method **`set_attributes`** for this instance.

3. Remove or comment out all calls of the method **`set_attributes`**.
4. Move the instance creation into the TRY block of the exception handling.
5. Repeat the previous steps for the other instances of your local class.
6. Activate the class. Execute it and debug the instantiation.

Unit 3
Solution 11

Use Private Attributes and a Constructor

In this exercise, you make the the static attribute of your class read-only and the instance attributes private. You use a constructor to set their values.

Template:

/LRN/CL_S4D400_CLS_METHODS (global Class)

Solution:

/LRN/CL_S4D400_CLS_CONSTRUCTOR (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class (**`ZCL_##_METHODS`**, **`ZCL_##_LOCAL_CLASS`** or **`ZCL_##_INSTANCES`**).

1. Copy class **`/LRN/CL_S4D400_CLS_METHODS`** to a class in your own package (suggested name: **`ZCL_##_CONSTRUCTOR`**, where ## stands for your group number).
   a) Open the source code of the global class **`/LRN/CL_S4D400_CLS_METHODS`**.
   b) Link the *Project Explorer* view with the editor.
   c) In the *Project Explorer* view, right-click the class **`/LRN/CL_S4D400_CLS_METHODS`** to open the context menu.
   d) From the context menu, choose *Duplicate ...*.
   e) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_CONSTRUCTOR`**, where ## stands for your group number.
   f) Adjust the description and choose *Next*.
   g) Confirm the transport request and choose *Finish*.

Task 2: Make Attributes Private
Change the visibility of the attributes **`carrier_id`** and **`connection_id`** to enforce the use of the methods **`set_attributes( )`** and **`get_output( )`**.

1. Set the visibility of the attribute **`carrier_id`** to private using a quick fix.
   a) Switch to the *Local Types* tab.
   b) Go to the declaration of the attribute **`carrier_id`** in local class **`lcl_connection`**.
   c) Place the cursor on **`carrier_id`** and press **`Ctrl + 1`**.
   d) Double-click the suggestion *Make carrier_id private* .
   e) Check that the declaration of the attribute **`carrier_id`** has moved to the private section.

2. Set the visibility of the attribute **connection_id** to private using a quick fix.
   a) Go to the declaration of the attribute **connection_id** in the local class **lcl_connection**.
   b) Place the cursor on **connection_id** and press **Ctrl + 1**.
   c) Double-click the suggestion *Make connection_id private* .
   d) Check that the declaration of attribute **connection_id** has moved to the private section.

Task 3: Create Instance Constructor

Replace the public method **set_attributes** with an instance constructor to ensure that the attributes are set during the creation of a new instance and that they are not changed afterward.

1. Comment out the definition and implementation of the method **set_attributes**.
   a) Select all lines that belong to the `METHODS set_attributes` statement, including the line with the period sign (.).
   b) Press **Ctrl + <** to add a star sign (*) in front of each selected line.
   c) Select all lines that belong to the implementation of the method **set_attributes** including the corresponding `ENDMETHOD.` statement and press **Ctrl + <** again.
2. Add an instance constructor to the local class **lcl_connection** using a quick fix. Ensure that the constructor has importing parameters corresponding to attributes **carrier_id** and **connection_id**.
   a) Place the cursor on the name of the class and press **Ctrl + 1**.
   b) Double-click on *Generate constructor*.
   c) In the dialog box, ensure that **carrier_id** and **connection_id** are selected and choose *Finish*.
3. Extend the generated constructor definition. Add exception **CX_ABAP_INVALID_VALUE** to the definition of the method **constructor**.
   a) Navigate to the generated definition of the method **constructor**.
   b) Add the highlighted code:

```
METHODS constructor
      IMPORTING
        i_connection_id TYPE /dmo/connection_id
        i_carrier_id    TYPE /dmo/carrier_id
      RAISING
        cx_ABAP_INVALID_VALUE.
```

4. Extend the generated constructor implementation. If either of the importing parameters is initial, raise exception **CX_ABAP_INVALID_VALUE**.
   a) Navigate from the definition to the implementation of the method **constructor**, for example, by placing the cursor on **constructor** and pressing **F3**.

b) Add the highlighted code:

```
METHOD constructor.

    IF i_carrier_id IS INITIAL OR i_connection_id IS INITIAL.
      RAISE EXCEPTION TYPE cx_abap_invalid_value.
    ENDIF.

    me->connection_id = i_connection_id.
    me->carrier_id    = i_carrier_id.

  ENDMETHOD.
```

5. In the constructor implementation, add a statement to increase the value of the static attribute **conn_counter** by one. Make sure the statement is only executed if no exception was raised.

   a) Add the highlighted code:

```
    me->connection_id = i_connection_id.
    me->carrier_id    = i_carrier_id.

    conn_counter = conn_counter + 1.

  ENDMETHOD.
```

6. Make sure that it is not possible to change the value of **conn_counter** from outside the class.

   a) Add `READ-ONLY` to the declaration of the static attribute **conn_counter**.

   b) Add the highlighted code:

```
  CLASS-DATA conn_counter TYPE i READ-ONLY.
```

Task 4: Use the Constructor

Adjust the instantiation of class **lcl_connection** to supply the parameters of the instance constructor and handle the exception.

1. In method **if_oo_adt_classrun~main**, go to the NEW #( ) expression for the first instance of **lcl_connection**.

   a) Switch to the *Global Class* tab.

   b) Find the first line with `connection = NEW #( ).`.

2. In the NEW expression, supply the import parameters of **constructor**.

Hint:
You can copy the parameter passing from the call of method **set_attributes** for this instance.

   a) Adjust the NEW expression as follows:

```
  connection = NEW #(
                     i_carrier_id    = 'LH'
                     i_connection_id = '0400'
                   ).
```

3. Remove or comment out all calls of the method **set_attributes**.

a) Select all lines that belong to the `connection->set_attributes( ... ).` statement, including the line with the closing bracket and the period sign (.).

b) Press **`Ctrl + <`** to add a star sign (*) in front of each selected line.

4. Move the instance creation into the TRY block of the exception handling.

a) Move the `TRY.` statement up, to before the instance creation.

b) Your first instance creation should now look like this:

```
TRY.
    connection = NEW #(
                         i_carrier_id    = 'LH'
                         i_connection_id = '0400'
                       ).

*         connection->set_attributes(
*           EXPORTING
*             i_carrier_id    = 'LH'
*             i_connection_id = '0400'
*         ).

          APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.
```

5. Repeat the previous steps for the other instances of your local class.

a) After this step, the implementation of the method **`if_oo_adt_classrun~main`** should look similar to this:

```
METHOD if_oo_adt_classrun~main.

    DATA connection TYPE REF TO lcl_connection.
    DATA connections TYPE TABLE OF REF TO lcl_connection.

* First Instance
**********************************************************************
    TRY.
    connection = NEW #(
                         i_carrier_id    = 'LH'
                         i_connection_id = '0400'
                       ).

*         connection->set_attributes(
*           EXPORTING
*             i_carrier_id    = 'LH'
*             i_connection_id = '0400'
*         ).

          APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.

* Second instance
**********************************************************************
    TRY.
```

```
    connection = NEW #(
                         i_carrier_id    = 'AA'
                         i_connection_id = '0017'
                       ).
        APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.

* Third instance
**********************************************************************
    TRY.
    connection = NEW #(
                         i_carrier_id    = 'SQ'
                         i_connection_id = '0001'
                       ).
        APPEND connection TO connections.

      CATCH cx_abap_invalid_value.
        out->write( `Method call failed` ).
    ENDTRY.

* Output
**********************************************************************
    LOOP AT connections INTO connection.
      out->write( connection->get_output( ) ).
    ENDLOOP.
  ENDMETHOD.
```

6. Activate the class. Execute it and debug the instantiation.
   a) Press **Ctrl + F3** to activate the class.
   b) Press **F9** to run the class.

Unit 4
Exercise 12

Read Data from a Database Table

In this exercise, you extend your local class with attributes for the departure airport and the destination airport, and read the values for these attributes from a database table.

Template:

/LRN/CL_S4D400_CLS_CONSTRUCTOR (global Class)

Solution:

/LRN/CL_S4D400_DBS_SELECT (global Class)

Task 1: Copy Template
Copy the template class. Alternatively, copy your solution of the previous exercise.

1. Copy the class **`/LRN/CL_S4D400_CLS_CONSTRUCTOR`** to a class in your own package (suggested name: **`ZCL_##_SELECT`**, where ## stands for your group number).

Task 2: Declare Additional Attributes
Extend the local class **`lcl_connection`** with the private instance attributes **`airport_from_id`**and **`airport_to_id`**. Add some output for the new attributes to the implementation of the method **`get_output`**.

Note:
The new attributes are not filled yet. In order to fill them we will add a SELECT statement to the **`constructor`** in one of the next tasks of this exercise.

1. Switch to the local class **`lcl_connection`**.
2. Add the following **private** attributes to the class definition:

   Table 2: Attributes

Attribute Name	Scope	Data Type
`airport_from_id`	instance	/DMO/AIRPORT_FROM_ID
`airport_to_id`	instance	/DMO/AIRPORT_TO_ID

3. Extend the implementation of the method **`get_output`**. Append more string templates to the returning parameter **`r_output`**. Embed the new attributes as expressions into the string templates.
4. Activate the class. Execute it and analyze the console output.

Task 3: Analyze the Database Table
Analyze the definition of the database table **`/DMO/CONNECTION`**.

1. Open the development object that contains the definition of the database table **/DMO/CONNECTION**.
2. Open the *Tooltip Description* for the database table /DMO/CONNECTION.

   Which fields have the description *Flight Reference Scenario: From Airport* and *Flight Reference Scenario: To Airport*?

Task 4: Read Data from the Database

In the method **constructor** of the local class **lcl_connection**, implement a SELECT statement that reads values for the new attributes from the database table **/DMO/CONNECTION**.

1. Return to the local class **lcl_connection** in your global class.
2. Navigate to the implementation of the method **constructor**.
3. After the `ENDIF.` statement, add a SELECT statement that reads a single record from database table **/DMO/CONNECTION**.
4. Implement the FIELDS clause. Read the table fields **airport_from_id** and **airport_to_id**.

   

   Hint:
   Use auto-completion (**Ctrl + Space**) to enter the field names.

5. Implement the WHERE condition. Restrict all key fields of the database table (except for the client field) with the values of importing parameters **i_carrier_id** and **i_connection_id**. Do not forget to escape the parameters with prefix @.

   

   Hint:
   Use auto-completion (**Ctrl + Space**) to enter the element names and parameter names.

   Why is it not necessary to specify the client field in the WERE condition?

6. Implement the INTO clause. Store the SELECT result in the attributes **airport_from_id**, and **airport_to_id**. Do not forget to escape the attributes with prefix @.

Hint:
Use auto-completion (**`Ctrl + Space`**) to enter the attribute names.

7. Implement error handling after the SELECT statement. Check the content of the system field **`sy-subrc`**. If it does not equal zero, raise exception **`CX_ABAP_INVALID_VALUE`**.
8. Activate the class. Execute it and analyze the console output. Check that the output for the new attributes displays data.

Unit 4
Solution 12

Read Data from a Database Table

In this exercise, you extend your local class with attributes for the departure airport and the destination airport, and read the values for these attributes from a database table.

Template:

/LRN/CL_S4D400_CLS_CONSTRUCTOR (global Class)

Solution:

/LRN/CL_S4D400_DBS_SELECT (global Class)

Task 1: Copy Template

Copy the template class. Alternatively, copy your solution of the previous exercise.

1. Copy the class **`/LRN/CL_S4D400_CLS_CONSTRUCTOR`** to a class in your own package (suggested name: **`ZCL_##_SELECT`**, where ## stands for your group number).
   a) Open the source code of the global class **`/LRN/CL_S4D400_CLS_CONSTRUCTOR`**.
   b) Link the *Project Explorer* view with the editor.
   c) In the *Project Explorer* view, right-click class **`/LRN/CL_S4D400_CLS_CONSTRUCTOR`** to open the context menu.
   d) From the context menu, choose *Duplicate* ....
   e) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_SELECT`**, where ## stands for your group number.
   f) Adjust the description and choose *Next*.
   g) Confirm the transport request and choose *Finish*.

Task 2: Declare Additional Attributes

Extend the local class **`lcl_connection`** with the private instance attributes **`airport_from_id`**and **`airport_to_id`**. Add some output for the new attributes to the implementation of the method **`get_output`**.

Note:
The new attributes are not filled yet. In order to fill them we will add a SELECT statement to the **`constructor`** in one of the next tasks of this exercise.

1. Switch to the local class **`lcl_connection`**.
   a) In the global class, choose *Local Types*.
2. Add the following **private** attributes to the class definition:

Table 2: Attributes

Attribute Name	Scope	Data Type
airport_from_id	instance	/DMO/AIRPORT_FROM_ID
airport_to_id	instance	/DMO/AIRPORT_TO_ID

a) Add the highlighted code:

```
  PRIVATE SECTION.
    DATA carrier_id    TYPE /dmo/carrier_id.
    DATA connection_id TYPE /dmo/connection_id.

    DATA airport_from_id TYPE /dmo/airport_from_id.
    DATA airport_to_id   TYPE /dmo/airport_to_id.

ENDCLASS.
```

3. Extend the implementation of the method **get_output**. Append more string templates to the returning parameter **r_output**. Embed the new attributes as expressions into the string templates.
   a) Navigate to the implementation of the method **get_output**.
   b) Add the highlighted code:

```
    APPEND |--------------------------------|             TO r_output.
    APPEND |Carrier:     { carrier_id      }|             TO r_output.
    APPEND |Connection:  { connection_id   }|             TO r_output.
    APPEND |Departure:   { airport_from_id }|             TO r_output.
    APPEND |Destination: { airport_to_id   }|             TO r_output.
```

4. Activate the class. Execute it and analyze the console output.
   a) Press **Ctrl + F3** to activate the class.
   b) Press **F9** to run the class.

Task 3: Analyze the Database Table

Analyze the definition of the database table **/DMO/CONNECTION**.

1. Open the development object that contains the definition of the database table **/DMO/CONNECTION**.
   a) From the eclipse toolbar, choose *Open ABAP Development Object* or press **Ctrl + Shift + A**.
   b) In the input field, enter **/dmo/con** as a search string.
   c) In the list of matching items, click on */DMO/CONNECTION (Database Table)* and choose *OK*.
2. Open the *Tooltip Description* for the database table /DMO/CONNECTION.
   a) Click on `/dmo/connection` after the keyword `define table` and press **F2** to show the tooltip description.

Which fields have the description *Flight Reference Scenario: From Airport* and *Flight Reference Scenario: To Airport*?

The field names are **airport_from_id** and **airport_to_id**.

Task 4: Read Data from the Database

In the method **constructor** of the local class **lcl_connection**, implement a SELECT statement that reads values for the new attributes from the database table **/DMO/CONNECTION**.

1. Return to the local class **lcl_connection** in your global class.
   a) In the editor view of Eclipse, open tab *ZCL_##_SELECT*.
   b) In the global class, choose *Local Types*.
2. Navigate to the implementation of the method **constructor**.
   a) Search for the code line `METHOD constructor.`.
3. After the `ENDIF.` statement, add a SELECT statement that reads a single record from database table **/DMO/CONNECTION**.
   a) Add the highlighted code:

```
IF i_carrier_id IS INITIAL OR i_connection_id IS INITIAL.
  RAISE EXCEPTION TYPE cx_abap_invalid_value.
ENDIF.

SELECT SINGLE
  FROM /dmo/connection
```

4. Implement the FIELDS clause. Read the table fields **airport_from_id** and **airport_to_id**.

Hint:
Use auto-completion (**Ctrl + Space**) to enter the field names.

   a) After `FROM /DMO/CONNECTION` enter `FIELDS`.
   b) After a blank, press **Ctrl + Space** and choose *airport_from_id*.
   c) After a comma and a blank press **Ctrl + Space** again and choose *airport_to_id*.
   d) The complete FIELDS clause should look like this:

```
    FIELDS DepartureAirport, DestinationAirport
```

5. Implement the WHERE condition. Restrict all key fields of the database table (except for the client field) with the values of importing parameters **i_carrier_id** and **i_connection_id**. Do not forget to escape the parameters with prefix @.

Hint:
Use auto-completion (**Ctrl + Space**) to enter the element names and parameter names.

a) Add the following code after the FIELDS clause:

```
  WHERE carrier_id    = @i_carrier_id
    AND connection_id = @i_connection_id
```

Why is it not necessary to specify the client field in the WERE condition?

Client handling is performed by the compiler.

6. Implement the INTO clause. Store the SELECT result in the attributes **airport_from_id**, and **airport_to_id**. Do not forget to escape the attributes with prefix @.

Hint:
Use auto-completion (**Ctrl + Space**) to enter the attribute names.

a) Add the highlighted code:

```
  WHERE carrier_id    = @i_carrier_id
    AND connection_id = @i_connection_id
  INTO ( @airport_from_id, @airport_to_id ).
```

b) The complete SELECT statement should look like this:

```
SELECT SINGLE
      FROM /dmo/connection
    FIELDS airport_from_id, airport_to_id
     WHERE carrier_id    = @i_carrier_id
       AND connection_id = @i_connection_id
      INTO ( @airport_from_id, @airport_to_id ).
```

7. Implement error handling after the SELECT statement. Check the content of the system field **sy-subrc**. If it does not equal zero, raise exception **CX_ABAP_INVALID_VALUE**.

a) Add the following code after the SELECT statement:

```
  IF sy-subrc <> 0.
     RAISE EXCEPTION TYPE cx_abap_invalid_value.
  ENDIF.
```

8. Activate the class. Execute it and analyze the console output. Check that the output for the new attributes displays data.

a) Press **Ctrl + F3** to activate the class.

b) Press **F9** to run the class.

Unit 4
Exercise 13

Analyze and Use a CDS View Entity

In this exercise, you read from a CDS view entity rather than from the database table directly. This provides you with comfortable access to the name of the airline as well.

Template:

/LRN/CL_S4D400_DBS_SELECT (global Class)

Solution:

/LRN/CL_S4D400_DBS_CDS (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class **`ZCL_##_SELECT`**.

1. Copy the class **`/LRN/CL_S4D400_DBS_SELECT`** to a class in your own package (suggested name: **`ZCL_##_CDS`**, where ## stands for your group number).

Task 2: Analyze CDS View Entity
Analyze the definition of the CDS view entity **`/DMO/I_Connection`**.

1. Open the development object that contains the definition of the CDS view entity **`/DMO/I_Connection`**.
2. Analyze the element list of the CDS view entity.

   Which are the alias names for fields **`airport_from_id`** and **`airport_to_id`**?

   _____

   _____

   _____

3. Open the *Tooltip Description* for the target of association **`_Airline`**.

   Which element is described as *Flight Reference Scenario: Carrier Name*? What is the element type?

   _____

   _____

   _____

Task 3: Declare Additional Attribute
Extend the local class **`lcl_connection`** with a private instance attribute **`carrier_name`**. Add some output for the new attribute to the implementation of method **`get_output( )`**.

Note:
The new attribute is not filled, yet. In order to fill it we will adjust the SELECT statement in the next task of this exercise.

1. Return to the local class **`lcl_connection`** in your global class.
2. Add the following **private** attribute to the class definition:

   Table 3: Attributes

Attribute Name	Scope	Data Type
`carrier_name`	instance	/DMO/CARRIER_NAME

3. Extend the implementation of method **`get_output`**. Add the carrier name to the string template with the carrier identification.
4. Activate the class. Execute it and analyze the console output.

Task 4: Use CDS View Entity

In the implementation of method **`constructor`**, replace the SELECT statement that reads from database table **`/dmo/connection`** with a SELECT statement that reads from CDS view entity **`/DMO/I_Connection`**.

1. In the implementation of method **`constructor`**, comment the SELECT statement.
2. After the commented code, add a SELECT statement that reads a single record from database table **`/DMO/I_Conncetion`**.
3. Implement the FIELDS clause. Read view elements **`DepartureAirport`** and **`DestinationAirport`**. In addition, read view element **`Name`** from the target of association **`_Airline`**.

Hint:
Use auto-completion (**`Ctrl + Space`**) to enter the names.

4. Implement the WHERE condition. Restrict the key elements of CDS view entity with the values of importing parameters **`i_carrier_id`** and **`i_connection_id`**. Do not forget to escape the parameters with prefix @.

Hint:
Use auto-completion (**`Ctrl + Space`**) to enter the element names and parameter names.

Is it possible to add the client field to the WHERE condition?

5. Implement the INTO clause. Store the SELECT result in attributes **`airport_from_id`**, **`airport_to_id`**, and **`airline_name`**. Do not forget to escape the attributes with prefix @.

Hint:
Use auto-completion (**`Ctrl + Space`**) to enter the attribute names.

6. Activate the class. Execute it and analyze the console output. Check that the output for the new attributes displays data.

Unit 4
Solution 13

Analyze and Use a CDS View Entity

In this exercise, you read from a CDS view entity rather than from the database table directly. This provides you with comfortable access to the name of the airline as well.

Template:

/LRN/CL_S4D400_DBS_SELECT (global Class)

Solution:

/LRN/CL_S4D400_DBS_CDS (global Class)

Task 1: Copy Template (Optional)
Copy the template class. If you finished the previous exercise, you can skip this task and continue editing your class **`ZCL_##_SELECT`**.

1. Copy the class **`/LRN/CL_S4D400_DBS_SELECT`** to a class in your own package (suggested name: **`ZCL_##_CDS`**, where ## stands for your group number).
   a) Open the source code of the global class **`/LRN/CL_S4D400_DBS_SELECT`**.
   b) Link the *Project Explorer* view with the editor.
   c) In the *Project Explorer* view, right-click the class **`/LRN/CL_S4D400_DBS_SELECT`** to open the context menu.
   d) From the context menu, choose *Duplicate ...*.
   e) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_CDS`**, where ## stands for your group number.
   f) Adjust the description and choose *Next*.
   g) Confirm the transport request and choose *Finish*.

Task 2: Analyze CDS View Entity
Analyze the definition of the CDS view entity **`/DMO/I_Connection`**.

1. Open the development object that contains the definition of the CDS view entity **`/DMO/I_Connection`**.
   a) From the eclipse toolbar, choose *Open ABAP Development Object* or press **`Ctrl + Shift + A`**.
   b) In the input field, enter **`/DMO/I_Con`** as a search string.
   c) In the list of matching items, click on */DMO/I_CONNECTION (Data Definition)* and choose *OK*.
2. Analyze the element list of the CDS view entity.

Which are the alias names for fields **airport_from_id** and **airport_to_id**?

The alias names are **DepartureAirport** and **DestinationAirport**.

a) Navigate to the comma-separated list between the pair of curly brackets ({ }).

b) You find the alias names after keyword `AS`.

3. Open the *Tooltip Description* for the target of association **_Airline**.
   a) Click on `/DMO/I_Carrier` before`as _Airline` and press **F2** to show the tooltip description.

   Which element is described as *Flight Reference Scenario: Carrier Name*? What is the element type?

   The element name is **Name** and the element type is **/dmo/carrier_name**.

Task 3: Declare Additional Attribute

Extend the local class **lcl_connection** with a private instance attribute **carrier_name**. Add some output for the new attribute to the implementation of method **get_output( )**.

Note:
The new attribute is not filled, yet. In order to fill it we will adjust the SELECT statement in the next task of this exercise.

1. Return to the local class **lcl_connection** in your global class.
   a) In the global class, choose *Local Types*.
2. Add the following **private** attribute to the class definition:

   Table 3: Attributes

Attribute Name	Scope	Data Type
carrier_name	instance	/DMO/CARRIER_NAME

   a) Add the highlighted code:

```
  PRIVATE SECTION.
    DATA carrier_id      TYPE /dmo/carrier_id.
    DATA connection_id   TYPE /dmo/connection_id.

    DATA airport_from_id TYPE /dmo/airport_from_id.
    DATA airport_to_id   TYPE /dmo/airport_to_id.

    DATA carrier_name    TYPE /dmo/carrier_name.
ENDCLASS.
```

3. Extend the implementation of method **get_output**. Add the carrier name to the string template with the carrier identification.
   a) Navigate to the implementation of method **get_output**.

b) Replace statement `APPEND |Carrier: { carrier_id }| TO r_output.` with the following statement:

```
APPEND |Carrier: { carrier_id } { carrier_name }| TO r_output.
```

4. Activate the class. Execute it and analyze the console output.
   a) Press **Ctrl + F3** to activate the class.
   b) Press **F9** to run the class.

Task 4: Use CDS View Entity

In the implementation of method **constructor**, replace the SELECT statement that reads from database table **/dmo/connection** with a SELECT statement that reads from CDS view entity **/DMO/I_Connection**.

1. In the implementation of method **constructor**, comment the SELECT statement.
   a) Select all lines that belong to the `SELECT SINGLE ... .`.
   b) Press **Ctrl + <** to add a star sign (*) in front of each selected line.
2. After the commented code, add a SELECT statement that reads a single record from database table **/DMO/I_Conncetion**.
   a) Add the highlighted code:

```
*    SELECT SINGLE
*      FROM /dmo/connection
*    FIELDS airport_from_id, airport_to_id
*     WHERE carrier_id    = @i_carrier_id
*       AND connection_id = @i_connection_id
*      INTO ( @airport_from_id, @airport_to_id ).

  SELECT SINGLE
    FROM /DMO/I_Connection
```

3. Implement the FIELDS clause. Read view elements **DepartureAirport** and **DestinationAirport**. In addition, read view element **Name** from the target of association **_Airline**.

> Hint:
> Use auto-completion (**Ctrl + Space**) to enter the names.

   a) After `FROM /DMO/I_Connection` enter `FIELDS`.
   b) After a blank, press **Ctrl + Space** and choose *DepartureAirport*.
   c) After a comma and a blank press **Ctrl + Space** again and choose **DestinationAirport**.
   d) After a comma and a blank, type in a backslash (\) , press **Ctrl + Space** and choose *_Airline*.
   e) Immediately after `_Airline`, type in a dash sign (-) , press **Ctrl + Space** and choose *Name*.

f) The complete FIELDS clause should look like this:

```
    FIELDS DepartureAirport, DestinationAirport, \_Airline-Name
```

4. Implement the WHERE condition. Restrict the key elements of CDS view entity with the values of importing parameters **`i_carrier_id`** and **`i_connection_id`**. Do not forget to escape the parameters with prefix @.

Hint:
Use auto-completion (**`Ctrl + Space`**) to enter the element names and parameter names.

a) Add the following code after the FIELDS clause:

```
  WHERE AirlineID    = @i_carrier_id
    AND ConnectionID = @i_connection_id
```

Is it possible to add the client field to the WHERE condition?

No, the element list of the CDS view entity does not contain the client field.

5. Implement the INTO clause. Store the SELECT result in attributes **`airport_from_id`**, **`airport_to_id`**, and **`airline_name`**. Do not forget to escape the attributes with prefix @.

Hint:
Use auto-completion (**`Ctrl + Space`**) to enter the attribute names.

a) Add the highlighted code:

```
  WHERE AirlineID    = @i_carrier_id
    AND ConnectionID = @i_connection_id
   INTO ( @airport_from_id, @airport_to_id, @carrier_name ).
```

b) The complete SELECT statement should look like this:

```
SELECT SINGLE
    FROM /DMO/I_Connection
  FIELDS DepartureAirport, DestinationAirport, \_Airline-Name
   WHERE AirlineID    = @i_carrier_id
     AND ConnectionID = @i_connection_id
    INTO ( @airport_from_id, @airport_to_id, @carrier_name  ).
```

6. Activate the class. Execute it and analyze the console output. Check that the output for the new attributes displays data.

a) Press **`Ctrl + F3`** to activate the class.

b) Press **`F9`** to run the class.

Unit 5
Exercise 14

Use a Structured Data Object

In this exercise, you declare a structured attribute, fill it using a SELECT statement and access the structure components.

Template:

/LRN/CL_S4D400_DBS_CDS (global Class)

Solution:

/LRN/CL_S4D400_STS_STRUCTURE (global Class)

Task 1: Copy Template

Copy the template class. Alternatively, copy your solution of the previous exercise.

1. Copy the class **`/LRN/CL_S4D400_DBS_CDS`** to a class in your own package (suggested name: **`ZCL_##_STRUCTURE`**, where ## stands for your group number).

Task 2: Declare a Structured Data Object

In the local class, declare a structured attribute **`details`** to replace the scalar attributes **`airport_from_id`**, **`airport_to_id`**, and **`carrier_name`**. Begin by defining a private structure type **`st_details`** inside the local class.

1. Switch to the local class **`lcl_connection`**.
2. Define a **private** structure type **`st_details`** with the following components:

   Table 4: Components of structure type st_details:

Component Name	Data Type
`DepartureAirport`	**`/dmo/airport_from_id`**
`DestinationAirport`	**`/dmo/airport_to_id`**
`AirlineName`	**`/dmo/carrier_name`**

3. Comment or remove the declaration of attributes **`airport_from_id`**, **`airport_to_id`**, and **`carrier_name`**.
4. Declare a new private instance attribute **`details`** and type it with structure type **`st_details.`**.

Task 3: Access Structure Components

Use the components of the structured attribute **`details`** in the method **`get_output`**.

1. Adjust the implementation of the method **`get_output`**. Replace any access to the attributes **`airport_from_id`**, **`airport_to_id`**, and **`carrier_name`** with the corresponding component of attribute **`details`**.

Hint:
Do not type in the component names manually! After typing the structure component selector (-), press **`Ctrl + Space`** to get a list of all components.

Task 4: Fill the Structured Attribute in the SELECT Statement

Use the structured attribute as the target of the SELECT statement in the **`constructor`** method.

1. Adjust the SELECT statement in the implementation of the **`constructor`** method. Replace the list of data objects in the INTO clause with the structured attribute **`details`**.
2. Optional: Use syntax variant `INTO CORRESPONDING FIELDS OF @details.`.
3. Activate the class. Execute it and analyze the console output. Check that the output displays data for all attributes.

Unit 5
Solution 14

Use a Structured Data Object

In this exercise, you declare a structured attribute, fill it using a SELECT statement and access the structure components.

Template:

/LRN/CL_S4D400_DBS_CDS (global Class)

Solution:

/LRN/CL_S4D400_STS_STRUCTURE (global Class)

Task 1: Copy Template
Copy the template class. Alternatively, copy your solution of the previous exercise.

1. Copy the class **`/LRN/CL_S4D400_DBS_CDS`** to a class in your own package (suggested name: **`ZCL_##_STRUCTURE`**, where ## stands for your group number).
   a) Open the source code of the global class **`/LRN/CL_S4D400_DBS_CDS`**.
   b) Link the *Project Explorer* view with the editor.
   c) In the *Project Explorer* view, right-click the class **`/LRN/CL_S4D400_DBS_CDS`** to open the context menu.
   d) From the context menu, choose *Duplicate* ....
   e) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_STRUCTURE`**, where ## stands for your group number.
   f) Adjust the description and choose *Next*.
   g) Confirm the transport request and choose *Finish*.

Task 2: Declare a Structured Data Object
In the local class, declare a structured attribute **`details`** to replace the scalar attributes **`airport_from_id`**, **`airport_to_id`**, and **`carrier_name`**. Begin by defining a private structure type **`st_details`** inside the local class.

1. Switch to the local class **`lcl_connection`**.
   a) In the global class, choose *Local Types*.
2. Define a **private** structure type **`st_details`** with the following components:

   Table 4: Components of structure type st_details:

Component Name	Data Type
`DepartureAirport`	**`/dmo/airport_from_id`**
`DestinationAirport`	**`/dmo/airport_to_id`**

Component Name	Data Type
AirlineName	**/dmo/carrier_name**

a) After line `PRIVATE SECTION.`, add the following code:

```
  TYPES:
    BEGIN OF st_details,
      DepartureAirport   TYPE /dmo/airport_from_id,
      DestinationAirport TYPE   /dmo/airport_to_id,
      AirlineName        TYPE   /dmo/carrier_name,
    END OF st_details.
```

3. Comment or remove the declaration of attributes **airport_from_id**, **airport_to_id**, and **carrier_name**.
   a) Select the lines with the three DATA statements.
   b) Press **Ctrl + <** to add a star sign (*) in front of each selected line.
4. Declare a new private instance attribute **details** and type it with structure type **st_details.**.
   a) Add the highlighted code:

```
*    DATA airport_from_id TYPE /dmo/airport_from_id.
*    DATA airport_to_id   TYPE /dmo/airport_to_id.
*
*    DATA carrier_name    TYPE /dmo/carrier_name.

    DATA details TYPE st_details.
```

Task 3: Access Structure Components

Use the components of the structured attribute **details** in the method **get_output**.

1. Adjust the implementation of the method **get_output**. Replace any access to the attributes **airport_from_id**, **airport_to_id**, and **carrier_name** with the corresponding component of attribute **details**.

Hint:
Do not type in the component names manually! After typing the structure component selector (-), press **Ctrl + Space** to get a list of all components.

   a) Navigate to the implementation of the method **get_output**.
   b) Adjust the APPEND statements as follows:

```
*    APPEND |--------------------------------|             TO r_output.
*    APPEND |Carrier:     { carrier_id } { carrier_name }| TO r_output.
*    APPEND |Connection:  { connection_id   }|             TO r_output.
*    APPEND |Departure:   { airport_from_id }|             TO r_output.
*    APPEND |Destination: { airport_to_id   }|             TO r_output.

    APPEND |--------------------------------|                  TO
r_output.
```

```
    APPEND |Carrier:     { carrier_id } { details-airlinename }| TO
r_output.
    APPEND |Connection:  { connection_id   }|                    TO
r_output.
    APPEND |Departure:   { details-departureairport     }|       TO
r_output.
    APPEND |Destination: { details-destinationairport   }|       TO
r_output.
```

Task 4: Fill the Structured Attribute in the SELECT Statement

Use the structured attribute as the target of the SELECT statement in the **`constructor`** method.

1. Adjust the SELECT statement in the implementation of the **`constructor`** method. Replace the list of data objects in the INTO clause with the structured attribute **`details`**.
   a) Navigate to the implementation of method **`constructor`**.
   b) Adjust the SELECT statement as follows:

```
SELECT SINGLE
      FROM /DMO/I_Connection
    FIELDS DepartureAirport, DestinationAirport, \_Airline-Name
     WHERE AirlineID    = @i_carrier_id
       AND ConnectionID = @i_connection_id
*       INTO ( @airport_from_id, @airport_to_id, @carrier_name  ).
      INTO @details.
```

2. Optional: Use syntax variant `INTO CORRESPONDING FIELDS OF @details.`.
   a) In the SELECT statement, replace `INTO @details.` with the following code:

```
  INTO CORRESPONDING FIELDS OF @details.
```

   b) Add alias name **`AirlineName`** for the path expression.
   c) The SELECT statement should now look like this:

```
SELECT SINGLE
      FROM /DMO/I_Connection
    FIELDS DepartureAirport, DestinationAirport, \_Airline-Name as
AirlineName
     WHERE AirlineID    = @i_carrier_id
       AND ConnectionID = @i_connection_id
      INTO CORRESPONDING FIELDS OF @details.
```

3. Activate the class. Execute it and analyze the console output. Check that the output displays data for all attributes.
   a) Press **`Ctrl + F3`** to activate the class.
   b) Press **`F9`** to run the class.

Unit 6
Exercise 15

Use a Complex Internal Table

In this exercise, you declare a table-like attribute, fill it using a SELECT statement and access the content.

Template:

/LRN/CL_S4D400_STS_STRUCTURE (global Class)

Solution:

/LRN/CL_S4D400_ITS_ITAB (global Class)

Task 1: Copy Template
Copy the template class. Alternatively, copy your solution of the previous exercise.

1. Copy the class **`/LRN/CL_S4D400_STS_STRUCTURE`** to a class in your own package (suggested name: **`ZCL_##_ITAB`**, where ## stands for your group number).

Task 2: Declare an Internal Table
In the local class, declare a static, table-like attribute **`airports`** to buffer detail information on all available airports. Begin by defining a private structure type **`st_airport`** and a table type **`tt_airports`** inside the local class.

1. Switch to the local class **`lcl_connection`**.
2. Define a **private** structure type **`st_airport`** with the following components:

   Table 5: Components of structure type st_airport:

Component Name	Data Type
`AirportID`	**`/dmo/airport_id`**
`Name`	**`/dmo/airport_name`**

3. Define a **private** table type **`tt_airports`** with the following properties:

   Table 6: Properties of table type tt_airports:

Property	Value
Line type	**`st_airport`**
Table kind	**`STANDARD TABLE`**
`Key Definition`	**`NON-UNIQUE DEFAULT KEY`**

4. Declare a new private static attribute **`airports`** and type it with table type **`tt_airports.`**.

Task 3: Fill the Static Attribute in a Class Constructor

Define a class constructor and implement a SELECT statement that reads all available airports from the CDS view entity **`/DMO/I_Airports`** into the static attribute **`Airports`**.

1. Add a class constructor to the local class **`lcl_connection`** using a quick fix.
2. In the class constructor, implement a SELECT statement that reads all data sets from the CDS view entity **`/DMO/I_Airports`** into the static attribute **`Airports`** .

Task 4: Access the Content of the Internal Table

Use the content of internal table **`Airports`** in the method **`get_output`** to add the airport names to the output.

1. Navigate to the implementation of the method **`get_output`**.
2. At the beginning of the method, read the details of the departure airport into a structured data object **`departure`**.

Hint:
Use a table expression `Airports[ ... ]` and an inline declaration for the data object **`departure`**.

3. Similarly, read the details of the destination airport into a structured data object **`destination`**.
4. Use the component **`name`** of the two structures to add the airport names to the output.
5. Optional: Omit the structured data object and use the table expressions directly in the string templates.
6. Activate the class. Execute it and analyze the console output. Check that the output displays data for all attributes.

Unit 6
Solution 15

Use a Complex Internal Table

In this exercise, you declare a table-like attribute, fill it using a SELECT statement and access the content.

Template:

/LRN/CL_S4D400_STS_STRUCTURE (global Class)

Solution:

/LRN/CL_S4D400_ITS_ITAB (global Class)

Task 1: Copy Template
Copy the template class. Alternatively, copy your solution of the previous exercise.

1. Copy the class **`/LRN/CL_S4D400_STS_STRUCTURE`** to a class in your own package (suggested name: **`ZCL_##_ITAB`**, where ## stands for your group number).
   a) Open the source code of the global class **`/LRN/CL_S4D400_STS_STRUCTURE`**.
   b) Link the *Project Explorer* view with the editor.
   c) In the *Project Explorer* view, right-click the class **`/LRN/CL_S4D400_STS_STRUCTURE`** to open the context menu.
   d) From the context menu, choose *Duplicate* ....
   e) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_ITAB`**, where ## stands for your group number.
   f) Adjust the description and choose *Next*.
   g) Confirm the transport request and choose *Finish*.

Task 2: Declare an Internal Table
In the local class, declare a static, table-like attribute **`airports`** to buffer detail information on all available airports. Begin by defining a private structure type **`st_airport`** and a table type **`tt_airports`** inside the local class.

1. Switch to the local class **`lcl_connection`**.
   a) In the global class, choose *Local Types*.
2. Define a **private** structure type **`st_airport`** with the following components:

   Table 5: Components of structure type st_airport:

Component Name	Data Type
`AirportID`	**`/dmo/airport_id`**
`Name`	**`/dmo/airport_name`**

a) Add the highlighted code:

```
  TYPES:
     BEGIN OF st_details,
       DepartureAirport   TYPE /dmo/airport_from_id,
       DestinationAirport TYPE /dmo/airport_to_id,
       AirlineName        TYPE /dmo/carrier_name,
     END OF st_details.

   TYPES:
     BEGIN OF st_airport,
       AirportId TYPE /dmo/airport_id,
       Name      TYPE /dmo/airport_name,
     END OF st_airport.
```

3. Define a **private** table type **`tt_airports`** with the following properties:

Table 6: Properties of table type tt_airports:

Property	Value
Line type	**`st_airport`**
Table kind	**`STANDARD TABLE`**
`Key Definition`	**`NON-UNIQUE DEFAULT KEY`**

a) Add the highlighted code:

```
TYPES:
     BEGIN OF st_airport,
       AirportId TYPE /dmo/airport_id,
       Name      TYPE /dmo/airport_name,
     END OF st_airport.

   TYPES tt_airports TYPE STANDARD TABLE OF st_airport
                          WITH NON-UNIQUE DEFAULT KEY.
```

4. Declare a new private static attribute **`airports`** and type it with table type **`tt_airports`**..

a) Add the highlighted code at the end of the class definition:

```
  DATA details TYPE st_details.

  CLASS-DATA airports TYPE tt_airports.

ENDCLASS.
```

Task 3: Fill the Static Attribute in a Class Constructor

Define a class constructor and implement a SELECT statement that reads all available airports from the CDS view entity **`/DMO/I_Airports`** into the static attribute **`Airports`**.

1. Add a class constructor to the local class **`lcl_connection`** using a quick fix.

a) Place the cursor on the name of the class and press **`Ctrl + 1`**.

b) Double-click on *Generate class constructor*.

2. In the class constructor, implement a SELECT statement that reads all data sets from the CDS view entity **`/DMO/I_Airports`** into the static attribute **`Airports`** .

a) Navigate to the implementation of the method **class_constructor**.

b) Inside the method implementation, add the following code:

```
    SELECT FROM /DMO/I_Airport
         FIELDS AirportID, Name
           INTO TABLE @airports.
```

Task 4: Access the Content of the Internal Table

Use the content of internal table **Airports** in the method **get_output** to add the airport names to the output.

1. Navigate to the implementation of the method **get_output**.
   a) Proceed as you have done in previous exercises.
2. At the beginning of the method, read the details of the departure airport into a structured data object **departure**.

Hint:
Use a table expression `Airports[ ... ]` and an inline declaration for the data object **departure**.

   a) At the beginning of the method, add the following code:

```
DATA(departure)   = airports[ airportID = details-departureairport ].
```

3. Similarly, read the details of the destination airport into a structured data object **destination**.
   a) Add the highlighted code:

```
  DATA(departure)   = airports[ airportID = details-
departureairport   ].
  DATA(destination) = airports[ airportID = details-
destinationairport ].
```

4. Use the component **name** of the two structures to add the airport names to the output.
   a) Add the highlighted code:

```
    APPEND |Departure:   { details-departureairport   } { departure-
name   }| TO r_output.
    APPEND |Destination: { details-destinationairport } { destination-
name }| TO r_output.
```

5. Optional: Omit the structured data object and use the table expressions directly in the string templates.
   a) Comment the code lines where you fill the structures **departure** and **destination**.
   b) In the string template replace `departure-name` with `airports[ airportid = details-departureairport ]-name`.
   c) Similarly replace `destination-name` with `airports[ airportid = details-destination ]-name`.
6. Activate the class. Execute it and analyze the console output. Check that the output displays data for all attributes.

a) Press **`Ctrl + F3`** to activate the class.

b) Press **`F9`** to run the class.

Unit 7
Exercise 16

Analyze a Business Object

In this exercise, you analyze the business object interface /DMO/I_AgencyTP to find out about its structure and the data manipulation operations it offers.

Task 1: Analyze the Interface Behavior
Analyze the behavior definition of the business object interface **`/DMO/I_AGENCYTP`**.

1. Open the behavior definition **`/DMO/I_AGENCYTP`** in the editor.
2. Analyze the source code. Open the ABAP language help to find more information.

   What is the purpose of the statements `interface;`, `use draft;`, and `extensible`?

   How many entities does the business object interface consist of? What are the alias names of these entities?

   Which standard operations are accessible through this interface?

   Where do you find the non-standard operations that the interface offers?

Task 2: Analyze the Data Structure
Analyze the CDS view entity that defines the data structure of the business object interface. Navigate to its data source until you reach a CDS view entity that is not a projection.

1. Navigate to the CDS view entity **`/DMO/I_AgencyTP`** and analyze the source code.

Which are the key elements of the CDS view entity?

Is the data structure of the business object interface also extensible?

What is the data source of the CDS view entity **`/DMO/I_AgencyTP`**?

2. Navigate to the data source of the CDS view entity **`/DMO/I_AgencyTP`** and analyze the source code.

   Is the CDS view entity **`/DMO/R_AgencyTP`** also extensible?

   What is the data source of the CDS view entity **`/DMO/R_AgencyTP`**?

   Does CDS view entity **`/DMO/R_AgencyTP`** belong to a business object interface or a business object projection?

Task 3: Analyze the Business Object Behavior

Analyze the behavior definition of the business object **`/DMO/R_AGENCYTP`** which lies under business object interface **`/DMO/I_AGENCYTP`**.

1. Open the behavior definition **`/DMO/R_AGENCYTP`** in the editor.
2. Analyze the source code. Open the ABAP language help to find more information.

What is the purpose of the statement `managed ...;`?

What is the purpose of the addition `... implementation in class /dmo/bp_r_agencytp unique;`

Are there any elements that must not be changed?

What is the purpose of the statement `validation /DMO/validateName on save ... ;`

What is the purpose of the addition `{ create; field Name; }`

Task 4: Analyze the Behavior Implementation

Analyze the behavior implementation of the business object **`/DMO/R_AGENCYTP`**.

1. Navigate to the ABAP class **`/DMO/BP_R_AGENCYTP`** that contains the behavior implementation of the business object **`/DMO/R_AGENCYTP`**.

   Why are the definition and implementation of the global class **`/dmo/bp_r_agencytp`** both empty?

2. Go to the local class **`lhc_agency`** and analyze the definition.

   How many methods are defined in the local class **`lhc_agency`**? What is their purpose?

Unit 7
Solution 16

Analyze a Business Object

In this exercise, you analyze the business object interface /DMO/I_AgencyTP to find out about its structure and the data manipulation operations it offers.

Task 1: Analyze the Interface Behavior
Analyze the behavior definition of the business object interface **`/DMO/I_AGENCYTP`**.

1. Open the behavior definition **`/DMO/I_AGENCYTP`** in the editor.
2. Analyze the source code. Open the ABAP language help to find more information.

   What is the purpose of the statements `interface;`, `use draft;`, and `extensible`?

   `interface;` states that this behavior definition does not belong to the business object itself but rather to a released interface on top of the business object. `use draft;` stipulates that this interface supports the handling of interim versions of data (draft instances). `extensible` indicates that extensions of this interface by partners and customers are supported.

   How many entities does the business object interface consist of? What are the alias names of these entities?

   The business object interface consists of only one entity because the behavior definition contains only one `DEFINE BEHAVIOR FOR ...` statement. The alias name is **`/DMO/Agency`**.

   Which standard operations are accessible through this interface?

   The interface offers operations create, update, delete because the behavior contains the statements `use create;`, `use update;`, and `use delete;`

   Where do you find the non-standard operations that the interface offers?

   Non-standard operations are listed after the keyword `use action`. Note, that all actions in this interface are draft actions. They are related to the handling of interim versions of data.

Task 2: Analyze the Data Structure
Analyze the CDS view entity that defines the data structure of the business object interface. Navigate to its data source until you reach a CDS view entity that is not a projection.

1. Navigate to the CDS view entity **/DMO/I_AgencyTP** and analyze the source code.
   a) Go to the statement `define behavior for /DMO/I_AgencyTP`.
   b) Hold down the **Ctrl** key and left-click on `/DMO/I_AgencyTP`. Alternatively, place the cursor on `/DMO/I_AgencyTP` and press **F3**..

   Which are the key elements of the CDS view entity?

   There is only one element with the prefix `key`. The name of this view element is **AgencyID**.

   Is the data structure of the business object interface also extensible?

   Yes, there is a group of annotations starting with `@AbapCatalog.extensibility` with annotation `extensible: true`.

   What is the data source of the CDS view entity **/DMO/I_AgencyTP**?

   CDS view entity **/DMO/I_AgencyTP** is a projection on CDS view entity **/DMO/R_AgencyTP**.

2. Navigate to the data source of the CDS view entity **/DMO/I_AgencyTP** and analyze the source code.
   a) Go to the code fragment `as projection on /DMO/R_AgencyTP`.
   b) Hold down the **Ctrl** key and left-click on `/DMO/R_AgencyTP`. Alternatively, place the cursor on `/DMO/R_AgencyTP` and press **F3**..

   Is the CDS view entity **/DMO/R_AgencyTP** also extensible?

   Yes, it also contains annotation `extensible : true`.

   What is the data source of the CDS view entity **/DMO/R_AgencyTP**?

   The CDS view entity **/DMO/R_AgencyTP** reads from the CDS view entity **/DMO/I_Agency**.

   Does CDS view entity **/DMO/R_AgencyTP** belong to a business object interface or a business object projection?

   No. CDS view entity **/DMO/R_AgencyTP** reads its data using the keyword `as select from` and not with `as projection on`. The CDS view entity **/DMO/R_AgencyTP** belongs to the definition of the business object itself.

Task 3: Analyze the Business Object Behavior

Analyze the behavior definition of the business object **/DMO/R_AGENCYTP** which lies under business object interface **/DMO/I_AGENCYTP**.

1. Open the behavior definition **/DMO/R_AGENCYTP** in the editor.
2. Analyze the source code. Open the ABAP language help to find more information.

   What is the purpose of the statement `managed ...;`?

   `managed` determines that this is a business object and not an interface or a projection. It also indicates that for this business object the standard operations create, update, delete are handled by the framework.

   What is the purpose of the addition `... implementation in class /dmo/bp_r_agencytp unique;`

   `... implementation in class /dmo/bp_r_agencytp` specifies the ABAP class for the behavior implementation of this business object. Note that the addition `unique` is mandatory and defines that each operation can be implemented exactly once.

   Are there any elements that must not be changed?

   Yes, the statement `field ( readonly ) AgencyID;` declares that element **AgencyID** must nor be changed. Note, that this is the key element of the CDS view entity **/DMO/R_AgencyTP**.

   What is the purpose of the statement `validation /DMO/validateName on save ... ;`

   This statement defines a consistency check (validation) for the business object. The name indicates that it checks the content of the element **Name** of the CDS view entity **/DMO/R_AgencyTP**.

   What is the purpose of the addition `{ create; field Name; }`

   The addition defines the trigger conditions: When a new data set is created (standard operation create) the check is always performed. When a data set is changed (standard operation update) the check is only performed in case the value of the element **Name** has changed.

Task 4: Analyze the Behavior Implementation

Analyze the behavior implementation of the business object **/DMO/R_AGENCYTP**.

1. Navigate to the ABAP class **/DMO/BP_R_AGENCYTP** that contains the behavior implementation of the business object **/DMO/R_AGENCYTP**.
   a) Go to the statement `managed implementation in class /dmo/bp_r_agencytp unique;`.
   b) Hold down the **Ctrl** key and left-click on `/dmo/bp_r_agencytp`. Alternatively, place the cursor on `/dmo/bp_r_agencytp` and press **F3.**.

   Why are the definition and implementation of the global class **/dmo/bp_r_agencytp** both empty?

   The global class **/dmo/bp_r_agencytp** is a behavior pool. It serves as a wrapper for one or more local classes. The behavior implementation itself is located in those local classes.

2. Go to the local class **lhc_agency** and analyze the definition.
   a) Go to the tab *Local Types*. Alternatively, expand root node */DMO/BP_R_AGENCYTP* in the *Outline* view and choose *LHC_AGENCY*.

   How many methods are defined in the local class **lhc_agency**? What is their purpose?

   There are five methods. Four methods implement validations (addition `VALIDATE ON SAVE`) . One method implements authorization checks (addition `FOR GLOBAL AUTHORIZATION`.

Unit 7
Exercise 17

Modify Data Using EML

In this exercise, you use EML to perform the standard operation *update* for the root entity of the business object **/DMO/R_AGENCYTP**. To avoid later inconsistencies, you do not access the business object directly but via its released stable interface **/DMO/_I_AGENCYTP.**

Template:

none

Solution:

/LRN/CL_S4D400_BOS_EML (global Class)

Task 1: Analyze Current Data
Use the *Data Preview* tool to check the current name of the travel agency with ID “0700##”, where ## stands for your group number.)

1. Open the definition of the CDS view entity **/DMO/I_AgencyTP**.
2. Open the *Data Preview* tool.
3. Check the current name of the travel agency with ID “0700##”, where ## stands for your group number.)

Task 2: Create a Global Class
In your own package, create a new ABAP class.

1. Create a new ABAP class called **ZCL_##_EML**, where **##** stands for your group number. Ensure that the class implements the interface **IF_OO_ADT_CLASSRUN**.

Task 3: Update Data
Implement an EML statement to change the name of the travel agency with ID “0700##”. (## stands for your group number.)

1. In the method **IF_OO_ADT_CLASSRUN~MAIN**, declare an internal table with the correct derived type for an EML update of entity **/DMO/I_AgencyTP** (suggested name: **agencies_upd)**.
2. Fill the internal table with a single line containing “0700##” as the value for the column **AGENCYID** (where ## stands for your group number) and any new value for the column **NAME**.
3. Implement an EML statement MODIFY ENTITIES for the business object interface **/DMO/I_AgencyTP**. Update the data of the root entity, ensuring that only the field **NAME** is changed.

Hint:
Remember that the interface behavior defines alias name **`/DMO/Agency`** for the root entity **`/DMO/I_AgencyTP`**.

4. In order to get some visible result from the application, write a text literal to the console using the method **`out->write`**.
5. Activate and test the class.
6. Check whether the new name has been written to the database.
7. Add the COMMIT ENTITIES statement to your code to commit the changes.
8. Activate and test the class again. Confirm that the changes are now in the database.

Unit 7
Solution 17

Modify Data Using EML

In this exercise, you use EML to perform the standard operation *update* for the root entity of the business object **/DMO/R_AGENCYTP**. To avoid later inconsistencies, you do not access the business object directly but via its released stable interface **/DMO/_I_AGENCYTP.**

Template:

none

Solution:

/LRN/CL_S4D400_BOS_EML (global Class)

Task 1: Analyze Current Data
Use the *Data Preview* tool to check the current name of the travel agency with ID “0700##”, where ## stands for your group number.)

1. Open the definition of the CDS view entity **/DMO/I_AgencyTP**.
   a) If you still have the data definition **/DMO/I_AGENCYTP** open from the previous exercise, switch to the corresponding tab. Otherwise, press **Ctrl + Shift + A** to open the ABAP development object.
2. Open the *Data Preview* tool.
   a) Right-click anywhere in the source code to open the context menu.
   b) From the context menu, choose *Open With* → *Data Preview*.
3. Check the current name of the travel agency with ID “0700##”, where ## stands for your group number.)
   a) Scroll down to the entry with the value “0700##” in the column **AgencyID** and take a note of the value in the column **Name**.

Task 2: Create a Global Class
In your own package, create a new ABAP class.

1. Create a new ABAP class called **ZCL_##_EML**, where **##** stands for your group number. Ensure that the class implements the interface **IF_OO_ADT_CLASSRUN**.
   a) Choose *File* → *New* → *ABAP Class*.
   b) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **ZCL_##_EML**, where **##** is your group number. Enter a description.
   c) In the *Interfaces* group box, choose *Add*.
   d) Enter **IF_OO_ADT_CLASSRUN**. When the interface appears in the hit list, double-click it to add it to the class definition.
   e) Choose *Next*.
   f) Select your transport request and choose *Finish*.

Task 3: Update Data

Implement an EML statement to change the name of the travel agency with ID “0700##”. (## stands for your group number.)

1. In the method **`IF_OO_ADT_CLASSRUN~MAIN`**, declare an internal table with the correct derived type for an EML update of entity **`/DMO/I_AgencyTP`** (suggested name: **`agencies_upd)`**.
   a) Add the following code:

```
DATA agencies_upd TYPE TABLE FOR UPDATE /DMO/I_AgencyTP.
```

2. Fill the internal table with a single line containing “0700##” as the value for the column **`AGENCYID`** (where ## stands for your group number) and any new value for the column **`NAME`**.
   a) Add some code similar to the highlighted code:

```
DATA agencies_upd TYPE TABLE FOR UPDATE /dmo/i_agencytp.

agencies_upd = VALUE #( ( agencyid = '0700##' name = 'Some fancy
new name' ) ).
```

3. Implement an EML statement MODIFY ENTITIES for the business object interface **`/DMO/I_AgencyTP`**. Update the data of the root entity, ensuring that only the field **`NAME`** is changed.

Hint:
Remember that the interface behavior defines alias name **`/DMO/Agency`** for the root entity **`/DMO/I_AgencyTP`**.

   a) At the end of the method, add the following code:

```
MODIFY ENTITIES OF /dmo/i_agencytp
   ENTITY /dmo/agency
   UPDATE FIELDS ( name )
     WITH agencies_upd.
```

4. In order to get some visible result from the application, write a text literal to the console using the method **`out->write`**.
   a) After the `MODIFY ENTITIES` statement, add the following code:

```
out->write( `Method execution finished!`  ).
```

5. Activate and test the class.
   a) Press **`Ctrl + F3`** to activate the class.
   b) Press **`F9`** to run the class.
6. Check whether the new name has been written to the database.
   a) Return to the data preview for CDS view entity **`/DMO/I_AgencyTP`**.
   b) From the toolbar of the *Data Preview* tab, choose *Refresh*.

Why has your data change not yet arrived in the database?

Your program stores the changes in the application buffer, but they are not written to the database until a COMMIT ENTITIES statement.

7. Add the COMMIT ENTITIES statement to your code to commit the changes.
   a) Return to the method **`IF_OO_ADT_CLASSRUN~MAIN`** in your global class **`ZCL_##_EML`**.
   b) Add the highlighted code :

```
METHOD if_oo_adt_classrun~main.

    DATA agencies_upd TYPE TABLE FOR UPDATE /dmo/i_agencytp.

    agencies_upd = VALUE #( ( agencyid = '0700##' name = 'Some fancy
new name' ) ).

    MODIFY ENTITIES OF /dmo/i_agencytp
      ENTITY /dmo/agency
      UPDATE FIELDS ( name )
        WITH agencies_upd.

    COMMIT ENTITIES.

    out->write( `Method execution finished!`  ).

  ENDMETHOD.
```

8. Activate and test the class again. Confirm that the changes are now in the database.
   a) Press **`Ctrl + F3`** to activate the class.
   b) Press **`F9`** to run the class.
   c) Return to the data preview for the CDS view entity **`/DMO/I_AgencyTP`**.
   d) From the toolbar of the *Data Preview* tab, choose *Refresh*.

Unit 8
Exercise 18

Copy a Database Table

In the exercises of this unit, you will create an OData UI service. This UI service will form the basis for an application that allows you to change the price of flights. Firstly, you create a dedicated package for the required development objects. Then you copy a database table and fill it with data.

Template:

/LRN/S4D400_APT (Database Table)
/LRN/CL_S4D400_APT_COPY (global Class)

Solution:

/LRN/S4D400AFLGT (Database Table)
/LRN/CL_S4D400_APS_COPY (global Class)

Task 1: Create Subpackage
Create a new subpackage under your own package ZS4D400_## (where ## stands for your group number).

The subpackage should have the following attributes:

Attribute	Value
Name	**ZS4D400_##_RAP**, where **##** stands for your group number
Description	Objects for OData UI Service
Add to Favorite Packages	Checked
Superpackage	**ZS4D400_##**, , where **##** stands for your group number
Package Type	Development
Software Component	ZLOCAL
Application Component	**CA**
Transport Layer	

1. Create a sub package under your own package ZS4D400_## (where ## stands for your group number).

Task 2: Copy Database Table
Under the new package, create a copy of the database table **/LRN/S4D400_APT**. Name the new database table **Z##FLIGHT**, where ## stands for your group number.

1. Copy database table **`/LRN/S4D400_APT`** to your own package **`ZS4D400_##_RAP`**, where ## stands for your group number.
2. Activate the database table definition.

Task 3: Fill Database Table

In your own package, create a copy global class **`/LRN/CL_S4D400_APT_COPY`** (suggested name: **`ZCL_##_COPY`**. In this copy, change the value of constant **`table name`** to the name of your own database table. Then execute the class as a console app. This will fill your database table with data.

1. Copy the class **`/LRN/CL_S4D400_APT_COPY`** to a class in your own package **`ZS4D400_##`**.
2. In the copy, adjust the source code of the method **`IF_OO_ADT_CLASSRUN~MAIN`**. Change the value of the constant **`table_name`** to the name of your own database table.
3. Activate and execute the class as a console app.

Unit 8
Solution 18

Copy a Database Table

In the exercises of this unit, you will create an OData UI service. This UI service will form the basis for an application that allows you to change the price of flights. Firstly, you create a dedicated package for the required development objects. Then you copy a database table and fill it with data.

Template:

/LRN/S4D400_APT (Database Table)
/LRN/CL_S4D400_APT_COPY (global Class)

Solution:

/LRN/S4D400AFLGT (Database Table)
/LRN/CL_S4D400_APS_COPY (global Class)

Task 1: Create Subpackage
Create a new subpackage under your own package ZS4D400_## (where ## stands for your group number).

The subpackage should have the following attributes:

Attribute	Value
Name	**ZS4D400_##_RAP**, where **##** stands for your group number
Description	Objects for OData UI Service
Add to Favorite Packages	Checked
Superpackage	**ZS4D400_##**, , where **##** stands for your group number
Package Type	Development
Software Component	ZLOCAL
Application Component	**CA**
Transport Layer	

1. Create a sub package under your own package ZS4D400_## (where ## stands for your group number).
   a) In the Project Explorer, right-click on your ABAP package ZS4D400_## and choose *New* → *ABAP Package* .
   b) Enter the package name **ZS4D400_##_RAP** where **##** stands for your group number.

c) Enter the description **`Objects for OData UI Service`**.

d) Mark the checkbox *Add to favorite packages*

e) Ensure that the field *Superpackage* contains the name of your already existing package.

f) Ensure that the *Package Type* is set to *Development*.

g) Choose *Next*.

h) Ensure that the *Software Component* is set to *ZLOCAL*.

i) Enter the Application component **`CA`**.

j) Ensure that the *Transport Layer* is empty.

k) Choose *Next*.

l) Confirm the transport request and choose *Finish*.

Task 2: Copy Database Table

Under the new package, create a copy of the database table **`/LRN/S4D400_APT`**. Name the new database table **`Z##FLIGHT`**, where ## stands for your group number.

1. Copy database table **`/LRN/S4D400_APT`** to your own package **`ZS4D400_##_RAP`**, where ## stands for your group number.
   a) Open the definition of the database table **`/LRN/S4D400_APT`** .
   b) Link the *Project Explorer* view with the editor.
   c) In the *Project Explorer* view, right-click database table **`/LRN/S4D400_APT`** to open the context menu.
   d) From the context menu, choose *Duplicate ...*.
   e) Enter the name of your package **`ZS4D400_##_RAP`** in the *Package* field. In the *Name* field, enter the name **`Z##FLIGHT`**, where ## stands for your group number.
   f) Adjust the description and choose *Next*.
   g) Confirm the transport request and choose *Finish*.
2. Activate the database table definition.
   a) Press **`Ctrl + F3`** to activate the development object.

Task 3: Fill Database Table

In your own package, create a copy global class **`/LRN/CL_S4D400_APT_COPY`** (suggested name: **`ZCL_##_COPY`**. In this copy, change the value of constant **`table name`** to the name of your own database table. Then execute the class as a console app. This will fill your database table with data.

1. Copy the class **`/LRN/CL_S4D400_APT_COPY`** to a class in your own package **`ZS4D400_##`**.
   a) Open the source code of the global class **`/LRN/CL_S4D400_APT_COPY`**.
   b) If necessary, link the *Project Explorer* view with the editor.

c) In the *Project Explorer* view, right-click the class **`/LRN/CL_S4D400_APT_COPY`** to open the context menu.

d) From the context menu, choose *Duplicate* ....

e) Enter the name of your package in the *Package* field. In the *Name* field, enter the name **`ZCL_##_COPY`**, where ## stands for your group number.

f) Adjust the description and choose *Next*.

g) Confirm the transport request and choose *Finish*.

2. In the copy, adjust the source code of the method **`IF_OO_ADT_CLASSRUN~MAIN`**. Change the value of the constant **`table_name`** to the name of your own database table.

   a) Make sure the value of the constant euqls the name of your database table.

3. Activate and execute the class as a console app.

   a) Press **`Ctrl + F3`** to activate the class.

   b) Press **`F9`** to run the class.

   c) Check the *Console* view. The last output should read: <...> *was filled with data*. If the output reads <...> *is not a table of the right type.* make sure a database table of that name exists, is active, and is an exact copy of the template table.

Unit 8
Exercise 19

Generate and Preview an OData UI Service

In this exercise, you generate the repository objects for an OData UI Service and preview the service as an SAP Fiori app.

Template:

none

Solution:

The generated development objects in package **`/LRN/S4D400_EXERCISE_RAP`**.

You completed the previous exercises. You created the database table **`Z##FLIGHT`** (where ## is your group number) and the database table contains data.

Task 1: Use the Generator
Use the object generator *ABAP RESTful Application Programming Model → OData UI Service* to generate the following development objects in your **sub** package **`ZS4D400_##_RAP`**:

Object Type	Object Name
CDS Entity (model)	**`ZR_##Flight`** (Alias name: **`Flight`**)
Implementation Class (model)	**`ZBP_R_##FLIGHT`**
Draft Table	**`Z##FLIGHT_D`**
CDS Entity (Projection)	**`ZC_##Flight`**
Implementation Class (projection)	**`ZBP_C_##FLIGHT`**
Service Definition	**`ZUI_##FLIGHT_O4`**
Service Binding	**`ZUI_##FLIGHT_O4`** (Binding Type: *OData V4 - UI*)

1. Start the object generator for your database table **`Z##FLIGHT`**(where ## is your group number).
2. In the dialog box, select the generator.
3. Ensure that the development objects are created in your sub package **`ZS4D400_##_RAP`**.
4. Configure the generator with the data from the table.

Caution:
Do not choose *Next* before you have entered all data.

5. Preview the generator output and generate the objects.
6. Analyze the generator output in the *Project Explorer*.

Task 2: Preview the App

Publish the OData UI Service and preview the result as an SAP Fiori app.

1. Publish the local service endpoint if needed.
2. Preview the OData UI service as an SAP Fiori app.
3. Display the list of flights.

   Which buttons are available on the list toolbar?

   _____
   _____
   _____

4. For one of the flights, navigate to the details view.

   Which buttons are available on the object page toolbar?

   _____
   _____
   _____

5. For this flight, switch to change mode.

   Which fields can you edit in change mode?

   _____
   _____
   _____

   Why is it not possible to edit all fields?

   _____
   _____
   _____

6. Enter some text in the field *Flight Price* , for example **`a`** and a long text in the separate field *Currency Code* (at least 6 characters), for example **`abcdef`**.

   Is it possible to save the changes?

   _____
   _____
   _____

7. Enter **`abc`** in the **second** currency code field and a negative value in the *Flight Price*.

Can you save the changes now?

Unit 8
Solution 19

Generate and Preview an OData UI Service

In this exercise, you generate the repository objects for an OData UI Service and preview the service as an SAP Fiori app.

Template:

none

Solution:

The generated development objects in package **`/LRN/S4D400_EXERCISE_RAP`**.

You completed the previous exercises. You created the database table **`Z##FLIGHT`** (where ## is your group number) and the database table contains data.

Task 1: Use the Generator

Use the object generator *ABAP RESTful Application Programming Model → OData UI Service* to generate the following development objects in your sub package **`ZS4D400_##_RAP`**:

Object Type	Object Name
CDS Entity (model)	**`ZR_##Flight`** (Alias name: **`Flight`**)
Implementation Class (model)	**`ZBP_R_##FLIGHT`**
Draft Table	**`Z##FLIGHT_D`**
CDS Entity (Projection)	**`ZC_##Flight`**
Implementation Class (projection)	**`ZBP_C_##FLIGHT`**
Service Definition	**`ZUI_##FLIGHT_O4`**
Service Binding	**`ZUI_##FLIGHT_O4`** (Binding Type: *OData V4 - UI*)

1. Start the object generator for your database table **`Z##FLIGHT`**(where ## is your group number).
   a) In the *Project Explorer*, expand the following path: *Favorite Packages → ZS4D400_## → ZS4D400_##_RAP → Dictionary → Database Tables* (where ## is your group number). Alternatively, open the definition of database table **`Z##FLIGHT`** and, if necessary, link the *Project Explorer* view with the editor.
   b) In the *Project Explorer* view, right-click database table **`Z##FLIGHT`** to open the context menu.
   c) From the context menu, choose *Generate ABAP Repository Object ...*.
2. In the dialog box, select the generator.

a) On the left-hand side of the dialog box, choose *ABAP RESTful Application Programming Model → OData UI Service*.

b) Choose *Next* to navigate to the next step.

3. Ensure that the development objects are created in your sub package **`ZS4D400_##_RAP`**.

a) When the window reads *Enter Package* at the top, ensure that the field *Package* contains **`ZS4D400_##_RAP`**.

b) Choose *Next* to navigate to the next step.

4. Configure the generator with the data from the table.

Caution:
Do not choose *Next* before you have entered all data.

a) On the left-hand side of the dialog box, choose *Data Model*. In the right-hand pane, enter the *Data Definition Name* **`ZR_##_Flight`** and *Alias Name* **`Flight`**.

b) On the left-hand side of the dialog box, choose *Behavior*. In the right-hand pane, enter the *Behavior Implementation Class* **`ZBP_R_##FLIGHT`** and the *Draft Table Name* **`Z##FLIGHT_D`**.

c) On the left-hand side of the dialog box, choose *Service Projection*. In the right-hand pane, enter the *Name* **`ZC_##Flight`**.

d) On the left-hand side of the dialog box, choose *Service Projection Behavior*. In the right-hand pane, enter the *Behavior Implementation Class* **`ZBP_C_##FLIGHT`**.

e) On the left-hand side of the dialog box, choose *Service Definition*. In the right-hand pane, enter the *Name* **`ZUI_##FLIGHT_O4`**.

f) On the left-hand side of the dialog box, click *Service Binding*. In the right-hand pane, enter the *Name* **`ZUI_##FLIGHT_O4`** and ensure the *Binding Type* is set to **`OData V4 - UI`**.

g) After you entered all data choose *Next*.

5. Preview the generator output and generate the objects.

a) Check the list of repository objects that are displayed.

b) When you are sure you have entered all names correctly choose *Next*.

c) Confirm the transport request and choose *Finish*.

6. Analyze the generator output in the *Project Explorer*.

a) In the *Project Explorer* view on the left-hand side, place the cursor on your sub package **`ZS4D400_##_RAP`** and press **`F5`** to reload the content of the package.

b) Expand all sub nodes of the package to see the names of the generated development objects.

Task 2: Preview the App

Publish the OData UI Service and preview the result as an SAP Fiori app.

1. Publish the local service endpoint if needed.

a) Open the Service Binding that was generated earlier.

b) If the Service Binding is inactive press **`Ctrl + F3`** to activate it.

c) In the *Services* section on the left-hand side, check the status of the service. If field *Local Service Endpoint* contains the value *Unpublished*, the service is not yet published.

d) To publish the service, choose *Publish* in the center of the view.

2. Preview the OData UI service as an SAP Fiori app.
   a) In the *Service Version Details* section on the right-hand side, choose *Flight* and then *Preview ...*.
   b) A browser window opens with a SAP Fiori app.
3. Display the list of flights.
   a) Choose *Go* to display the list of flights that are stored in your database table **`Z##FLIGHT`**.

   Which buttons are available on the list toolbar?

   Create,*Delete*, *Settings*, *Export Table*.

4. For one of the flights, navigate to the details view.
   a) Choose one of the list entries to navigate to the *Object Page* with details of that flight.

   Which buttons are available on the object page toolbar?

   Edit,*Delete*, *Share*.

5. For this flight, switch to change mode.
   a) Choose *Edit*.

   Which fields can you edit in change mode?

   Fields *Flight Price*, *Currency Code*, and *Plane Type*.

   Why is it not possible to edit all fields?

   Fields *Airline ID*, *Flight Number*, *Flight Date* belong to the primary key of the database table. Changing the primary key is never a good idea. By default, the generator sets all key fields to write protected during update.

6. Enter some text in the field *Flight Price* , for example **`a`** and a long text in the separate field *Currency Code* (at least 6 characters), for example **`abcdef`**.

Is it possible to save the changes?

No. The generated application includes technical input checks. Field *Flight Price* is based on a numeric type and the technical type of field *Currency Code* is 5 characters long.

7. Enter **abc** in the **second** currency code field and a negative value in the *Flight Price*.

   Can you save the changes now?

   Yes, the checks for positive prices and valid currency codes are business logic. They have to be implemented in the behavior of the business object.

Unit 8
Exercise 20

Validate Price and Currency

In this exercise you define and implement validations for the price and currency code.

Template:

none

Solution:

/LRN/S4D400_R_FLIGHT (Behavior Definition)
/LRN/BP_S4D400_R_FLIGHT (Global Class)

You completed the previous exercises. You created and filled the database table **`Z##FLIGHT`** (where ## is your group number) and generated the development objects for an OData UI Service.

Task 1: Validate the Price

Define and implement a validation to check that the field **`Price`** has a positive value (suggested name: **`validatePrice`**). If the value is negative or equals zero, reject the change and report a suitable error message from message class **`/LRN/S4D400`**.

1. In the behavior definition **`ZR_##FLIGHT`**, define a new validation **`validatePrice`**. Ensure that the validation is always executed during the standard operation **`create`** but only if the value of **`Price`** changed for all other operations.

   

   Hint:
   Use code-completion wherever possible to enter the code.

2. Activate the behavior definition.

3. Use a quick fix to create the validation implementation method in the behavior handler class.

   Where is the method created?

   _____

   _____

   _____

4. At the beginning of the method **`validatePrice`**, declare a structured data object that you type with the line type of **`failed-flight`** (suggested name: **`failed_record`**). Similarly, declare a structured data object that you type with the line type of **`reported-flight`** (suggested name: **`reported_record`**).

Note:
These structures will be used to add rows to **failed-flight** and **reported-flight**, in case the validation finds an error.

5. Use a READ ENTITIES statement to read the user input from the transactional buffer. Use the statement IN LOCAL MODE and ensure that only the key fields and the field **Price** are read. Use an inline declaration for the result set (suggested name: **flights**).

Note:
It is not necessary to list the key fields after the FIELDS addition. READ ENTITIES always reads the key fields.

6. Implement a loop over the data that you just read. Use an inline declaration for the work area (suggested name: **flight**).
7. Inside the loop, check if the component **Price** is greater than zero. If not, fill the structure **failed_record** with the key of the current flight and add the new code row to **failed-flight**. Similarly, fill structure **reported_record** with the key of the current flight and add the new code row to **reported-flight**.

Hint:
In draft-enabled business objects, it is recommended that you use the component **%tky** to assign the key.

8. Before the APPEND statement fill the component **%msg** of the structure **reported_record** with a reference to a message object. To create the message object, call the method **me->new_message** with the following input:

Parameter name	Value
id	'/LRN/S4D400'
number	'101'
severity	if_abap_behv_message=>severity-error

9. Activate the class.

Task 2: Validate Currency

Define a validation **validateCurrencyCode** to ensure that *CurrencyCode* contains a valid value. Use CDS view entity **I_Currency** to check whether a currency code is valid or not.

1. In the behavior definition **ZR_##FLIGHT**, define a new validation **validateCurrency**. Set the trigger conditions to `create` and `field CurrencyCode`.

Hint:
Use code-completion wherever possible to enter the code.

2. Activate the behavior definition and use a quick fix to create the validation implementation.
3. In the method **`validateCurrencyCode`**, declare structured data objects **`failed_record`** and **`reported_record`** as before.
4. For the flights that need to be validated, read the field **`CurrencyCode`** and the key fields from the transactional buffer and implement a loop over the data.
5. Inside the loop, check if the value of the component **`CurrencyCode`** is valid.

Hint:
Implement a SELECT statement with CDS view entity **`I_Currency`** as data source and `Currency = flight-CurrencyCode` as WHERE clause.

6. If the currency code is not valid, add a row to **`failed-flight`** and **`reported-flight`**. Do not forget to fill **`reported_record-%msg`** with a reference to a suitable message object. To create the message object, call the method **`me->new_message`** with the following input:

Parameter name	Value
`id`	'/LRN/S4D400'
`number`	'102'
`severity`	if_abap_behv_message=>severity-error
`v1`	flight-currencycode

7. Activate the class.

Task 3: Test and Debug

Set breakpoints in the validation implementations. Restart the preview of the OData UI service and change the price and currency code of an existing flight. Make valid and invalid entries, and debug the validations.

1. Set breakpoints at the READ ENTITIES statement of both methods.
2. Restart the preview of the OData Service.
3. In the app, display a list of flights, open the details for one of the flights and switch to change mode.
4. Make some changes to the price and currency code, and then choose *Save*. Analyze the validations in the debugger.

Note:
Enter the currency code in the **second** currency code field.

Unit 8
Solution 20

Validate Price and Currency

In this exercise you define and implement validations for the price and currency code.

Template:

none

Solution:

/LRN/S4D400_R_FLIGHT (Behavior Definition)
/LRN/BP_S4D400_R_FLIGHT (Global Class)

You completed the previous exercises. You created and filled the database table **Z##FLIGHT** (where ## is your group number) and generated the development objects for an OData UI Service.

Task 1: Validate the Price
Define and implement a validation to check that the field **Price** has a positive value (suggested name: **validatePrice**). If the value is negative or equals zero, reject the change and report a suitable error message from message class **/LRN/S4D400**.

1. In the behavior definition **ZR_##FLIGHT**, define a new validation **validatePrice**. Ensure that the validation is always executed during the standard operation **create** but only if the value of **Price** changed for all other operations.

Hint:
Use code-completion wherever possible to enter the code.

   a) In the *Project Explorer*, navigate to the behavior definition **ZR_##FLIGHT**.

   b) Add the highlighted code :

```
create;
update;
delete;

validation validatePrice on save { create; field Price; }
```

2. Activate the behavior definition.
   a) Press **Ctrl + F3** to activate the behavior definition.

3. Use a quick fix to create the validation implementation method in the behavior handler class.
   a) Position the cursor on the name of the validation (`validatePrice`) and choose *Ctrl + 1*.

b) Double-click the entry *Add validation...*.

Where is the method created?

In the local class **lhc_flight** of global class **ZBP_R_##FLIGHT**.

4. At the beginning of the method **validatePrice**, declare a structured data object that you type with the line type of **failed-flight** (suggested name: **failed_record**). Similarly, declare a structured data object that you type with the line type of **reported-flight** (suggested name: **reported_record**).

Note:
These structures will be used to add rows to **failed-flight** and **reported-flight**, in case the validation finds an error.

a) Add the following code:

```
DATA failed_record   LIKE LINE OF failed-flight.
DATA reported_record LIKE LINE OF reported-flight.
```

5. Use a READ ENTITIES statement to read the user input from the transactional buffer. Use the statement IN LOCAL MODE and ensure that only the key fields and the field **Price** are read. Use an inline declaration for the result set (suggested name: **flights**).

Note:
It is not necessary to list the key fields after the FIELDS addition. READ ENTITIES always reads the key fields.

a) After the declarations, add the following code, replacing **##** with your group number:

```
READ ENTITIES OF zr_##flight IN LOCAL MODE
      ENTITY flight
      FIELDS ( price )
        WITH CORRESPONDING #(  keys )
      RESULT DATA(flights).
```

6. Implement a loop over the data that you just read. Use an inline declaration for the work area (suggested name: **flight**).

a) After the EML statement, add following code:

```
LOOP AT flights INTO DATA(flight).

ENDLOOP.
```

7. Inside the loop, check if the component **Price** is greater than zero. If not, fill the structure **failed_record** with the key of the current flight and add the new code row to **failed-flight**. Similarly, fill structure **reported_record** with the key of the current flight and add the new code row to **reported-flight**.

Hint:
In draft-enabled business objects, it is recommended that you use the component **%tky** to assign the key.

a) Inside the loop, add the following code:

```
      IF flight-price <= 0.

        failed_record-%tky = flight-%tky.
        APPEND failed_record TO failed-flight.

        reported_record-%tky = flight-%tky.
        APPEND reported_record TO reported-flight.
      ENDIF.
```

8. Before the APPEND statement fill the component **%msg** of the structure **reported_record** with a reference to a message object. To create the message object, call the method **me->new_message** with the following input:

Parameter name	Value
id	'/LRN/S4D400'
number	'101'
severity	if_abap_behv_message=>severity-error

a) Add the highlighted code:

```
LOOP AT flights INTO DATA(flight).
  IF flight-price <= 0.

    failed_record-%tky = flight-%tky.
    APPEND failed_record TO failed-flight.

    reported_record-%tky = flight-%tky.

    reported_record-%msg =
      new_message(
            id       = '/LRN/S4D400'
            number   = '101'
            severity = if_abap_behv_message=>severity-error
      ).

    APPEND reported_record TO reported-flight.

  ENDIF.
ENDLOOP.
```

9. Activate the class.

a) Press **Ctrl + F3** to activate the class.

Task 2: Validate Currency

Define a validation **validateCurrencyCode** to ensure that *CurrencyCode* contains a valid value. Use CDS view entity **I_Currency** to check whether a currency code is valid or not.

1. In the behavior definition **ZR_##FLIGHT**, define a new validation **validateCurrency**. Set the trigger conditions to `create` and `field CurrencyCode`.

Hint:
Use code-completion wherever possible to enter the code.

   a) In the *Project Explorer*, navigate to the behavior definition **ZR_##FLIGHT**.

   b) Add the highlighted code :

```
  create;
  update;
  delete;

  validation validatePrice        on save { create; field Price; }
  validation validateCurrencyCode on save { create; field
CurrencyCode; }
```

2. Activate the behavior definition and use a quick fix to create the validation implementation.
   a) Press **Ctrl + F3** to activate the behavior definition.
   b) Position the cursor on the name of the validation (`validateCurrencyCode`) and choose *Ctrl + 1*.
   c) Double-click the entry *Add validation...*.
3. In the method **validateCurrencyCode**, declare structured data objects **failed_record** and **reported_record** as before.
   a) Add the following code:

```
    DATA failed_record   LIKE LINE OF failed-flight.
    DATA reported_record LIKE LINE OF reported-flight.
```

4. For the flights that need to be validated, read the field **CurrencyCode** and the key fields from the transactional buffer and implement a loop over the data.
   a) After the declarations, add the following code, replacing **##** with your group number:

```
   READ ENTITIES OF zr_##flight IN LOCAL MODE
         ENTITY flight
         FIELDS ( currencycode )
           WITH CORRESPONDING #(  keys )
         RESULT DATA(flights).

   LOOP AT flights INTO DATA(flight).

   ENDLOOP.
```

5. Inside the loop, check if the value of the component **CurrencyCode** is valid.

Hint:
Implement a SELECT statement with CDS view entity **`I_Currency`** as data source and `Currency = flight-CurrencyCode` as WHERE clause.

a) At the beginning of the method, add the highlighted code:

```
DATA failed_record   LIKE LINE OF failed-flight.
DATA reported_record LIKE LINE OF reported-flight.

DATA exists TYPE abap_bool.
```

b) Inside the loop, add the following code:

```
    exists = abap_false.

    SELECT SINGLE FROM i_currency
          FIELDS @abap_true
          WHERE currency = @flight-currencycode
          INTO @exists.

    IF  exists = abap_false. " the currency code is not valid

    ENDIF.
```

Note:
There are different ways to implement such an existence check. This example has the advantage that no data is actually retrieved from the database. The FIELDS list contains only a constant. However, it is technically not possible to entirely omit the FIELDS list and the INTO clause .

6. If the currency code is not valid, add a row to **`failed-flight`** and **`reported-flight`**. Do not forget to fill **`reported_record-%msg`** with a reference to a suitable message object. To create the message object, call the method **`me->new_message`** with the following input:

Parameter name	Value
`id`	'/LRN/S4D400'
`number`	'102'
`severity`	if_abap_behv_message=>severity-error
`v1`	flight-currencycode

a) Inside the IF-Block, add the following code:

```
 failed_record-%tky = flight-%tky.
 APPEND failed_record TO failed-flight.

 reported_record-%tky = flight-%tky.
```

```
    reported_record-%msg =
       new_message(
            id       = '/LRN/S4D400'
            number   = '102'
            severity = if_abap_behv_message=>severity-error
            v1       = flight-currencycode
        ).

    APPEND reported_record TO reported-flight.
```

7. Activate the class.
   a) Press **Ctrl + F3** to activate the class.

Task 3: Test and Debug

Set breakpoints in the validation implementations. Restart the preview of the OData UI service and change the price and currency code of an existing flight. Make valid and invalid entries, and debug the validations.

1. Set breakpoints at the READ ENTITIES statement of both methods.
   a) In the implementation of the method **validateprice**, find the READ ENTITIES statement and double-click the area to the left of the row number to set a breakpoint.
   b) Do the same in the implementation of the method **validateCurrencyCode**.
2. Restart the preview of the OData Service.
   a) Close the browser window or browser tab that contains the preview.
   b) Open your service binding. In the *Entity Set and Associations* list on the right-hand side, first choose entity *Flight* and then choose *Preview...*.
3. In the app, display a list of flights, open the details for one of the flights and switch to change mode.
   a) Proceed as you have done in previous exercises.
4. Make some changes to the price and currency code, and then choose *Save*. Analyze the validations in the debugger.

Note:
Enter the currency code in the **second** currency code field.

   a) Proceed as you have done in previous exercises.

Unit 8
Exercise 21

Adjust the User Interface

In this exercise, you adjust the user interface of your app; either directly by editing the UI-metadata, or indirectly by adjusting the business object behavior and the data model projection.

Template:

none

Solution:

/LRN/S4D400_C_FLIGHT (Behavior Projection)
/LRN/S4D400_C_FLIGHT (Metadata Extension)
/LRN/S4D400_C_FLIGHT (Projection View)

You completed the previous exercises. You created and filled the database table **`Z##FLIGHT`** (where ## is your group number) and generated the development objects for an OData UI Service.

Task 1: Adjust the Behavior
Exclude the standard operations *update* and *delete*, and set the field **`PlaneTypeID`** to read-only .

Note:
These changes should only apply to your OData UI Service. The business object itself should still support the creation and deletion of flights, and the changing of the airplane type.

1. In the behavior projection for your service, disable the standard operations *create* and *delete*.
2. In the behavior projection of your business object, set the field **`PlaneTypeID`** to read-only.
3. Activate the behavior projection.

Task 2: Adjust the UI Metadata
Hide the administrative fields from the UI. Make sure only **`CarrierID`** and **`ConnectionID`** are displayed as selection fields on the report list page. Change the field sequence on the object page so that **`PlaneTypeID`** is displayed before **`Price`** and **`CurrencyCode`**.

1. Edit the Metadata Extension of the projection view (**`ZC_##Flight`**). Remove all UI annotations for the fields **`LocalCreatedBy`**, **`LocalCreatedAt`**, **`LocalLastChangedBy`**, **`LocalLastChangedAt`**, and **`LastChangedAt`** and replace them with **`@UI.hidden: true`**.
2. Remove the **`@UI.selectionField`** annotation for all elements except for the elements **`CarrierID`** and **`ConnectionID`**.

3. Change the position of the **`PlaneTypeID`** field on the object page. Place it between **`FlightDate`** and **`Price`**.

Hint:
The position on the object page is specified with annotation **`@UI.identification`**.

4. Activate the metadata extension.

Task 3: Provide a Value Help

In the data model projection for your service, use existing CDS view **`I_CurrencyStdVH`** to define a value help for the field **`CurrencyCode`**.

1. Edit the definition of the projection view (**`ZC_##Flight`**). Annotate the view element **`CurrencyCode`** with annotation **`@Consumption.valueHelpDefinition: [{ }]`**. Use **`I_CurrencyStdVH`** as the value for the subannotation **`entity.name`** and **`Currency`** as the value for the subannotation **`entity.element`**.

Note:
`Currency` is the name of the key element of CDS view **`I_CurrencyStdVH`**.

2. Activate the projection view definition.

Task 4: Verify the Result

Restart the preview of the OData UI Service and verify the changes to the user interface.

1. Restart the preview of the OData Service.
2. On the report list page, verify that the buttons *Create* and *Delete* have disappeared.
3. Verify that there are two selection fields at the top of the report list page. Test their functionality.

Hint:
Be aware that the selection fields support the entry of multiple values and ranges.

4. For any data set, navigate to the object page. Verify that the *Delete* button has disappeared.
5. Verify that field *Plane Type* is displayed after field *Flight Date* and before field *Flight Price*.
6. Switch to change mode and check the order of the fields.

Unit 8
Solution 21

Adjust the User Interface

In this exercise, you adjust the user interface of your app; either directly by editing the UI-metadata, or indirectly by adjusting the business object behavior and the data model projection.

Template:

none

Solution:

/LRN/S4D400_C_FLIGHT (Behavior Projection)
/LRN/S4D400_C_FLIGHT (Metadata Extension)
/LRN/S4D400_C_FLIGHT (Projection View)

You completed the previous exercises. You created and filled the database table **`Z##FLIGHT`** (where ## is your group number) and generated the development objects for an OData UI Service.

Task 1: Adjust the Behavior
Exclude the standard operations *update* and *delete*, and set the field **`PlaneTypeID`** to read-only .

Note:
These changes should only apply to your OData UI Service. The business object itself should still support the creation and deletion of flights, and the changing of the airplane type.

1. In the behavior **projection** for your service, disable the standard operations *create* and *delete*.
   a) Open the behavior projection **`ZC_##Flight`** (where **`##`** is your group number).
   b) Add comment markers in front of the statement `use create;` and `use delete;`.
      To add the comment markers, proceed as usual: select the code rows you want to comment and press **`Ctrl + <`**.
      Your code should then look like this:

```
//  use create;
  use update;
//  use delete;
```

2. In the behavior **projection** of your business object, set the field **`PlaneTypeID`** to read-only.

a) In the behavior projection, add the following code:

```
field (readonly)   PlaneTypeID;
```

Note:
The precise position of the statement is not important. We suggest that you place it before the **`use ...`** statements.

3. Activate the behavior projection.
   a) Press **`Ctrl + F3`** to activate the behavior projection.

Task 2: Adjust the UI Metadata

Hide the administrative fields from the UI. Make sure only **`CarrierID`** and **`ConnectionID`** are displayed as selection fields on the report list page. Change the field sequence on the object page so that **`PlaneTypeID`** is displayed before **`Price`** and **`CurrencyCode`**.

1. Edit the Metadata Extension of the projection view (**`ZC_##Flight`**). Remove all UI annotations for the fields **`LocalCreatedBy`**, **`LocalCreatedAt`**, **`LocalLastChangedBy`**, **`LocalLastChangedAt`**, and **`LastChangedAt`** and replace them with **`@UI.hidden: true`**.
   a) Open the metadata extension **`ZC_##FLIGHT`** where **`##`** is your group number.
   b) Adjust the code as follows:

```
@UI.hidden: true
LocalCreatedBy;

@UI.hidden: true
LocalCreatedAt;

@UI.hidden: true
LocalLastChangedBy;

@UI.hidden: true
LocalLastChangedAt;

@UI.hidden: true
LastChangedAt;
```

2. Remove the **`@UI.selectionField`** annotation for all elements except for the elements **`CarrierID`** and **`ConnectionID`**.
   a) Adjust the code as follows:

```
@UI.identification: [ {
  position: 30 ,
  label: 'Flight Date'
} ]
@UI.lineItem: [ {
  position: 30 ,
  label: 'Flight Date'
} ]
FlightDate;

@UI.identification: [ {
```

```
  position: 40 ,
  label: 'Flight Price'
} ]
@UI.lineItem: [ {
  position: 40 ,
  label: 'Flight Price'
} ]
Price;

@UI.identification: [ {
  position: 50 ,
  label: 'Currency Code'
} ]
@UI.lineItem: [ {
  position: 50 ,
  label: 'Currency Code'
} ]
CurrencyCode;

@UI.identification: [ {
  position: 60 ,
  label: 'Plane Type'
} ]
@UI.lineItem: [ {
  position: 60 ,
  label: 'Plane Type'
} ]
PlaneTypeId;
```

3. Change the position of the **`PlaneTypeID`** field on the object page. Place it between **`FlightDate`** and **`Price`**.

Hint:
The position on the object page is specified with annotation **`@UI.identification`**.

   a) Change the value of annotation **`@UI.identification: [ { position: } ]`** for element **`planetypeid`**. Choose a value that lies between the values for elements **`flightdate`** and **`price`**.

   b) Adjust the code as follows:

```
@UI.identification: [ {
  position: 35,
  label: 'Plane Type'
} ]
@UI.lineItem: [ {
  position: 60 ,
  label: 'Plane Type'
} ]
PlaneTypeId;
```

4. Activate the metadata extension.
   a) Press **`Ctrl + F3`** to activate the metadata extension.

Task 3: Provide a Value Help

In the data model projection for your service, use existing CDS view **`I_CurrencyStdVH`** to define a value help for the field **`CurrencyCode`**.

1. Edit the definition of the projection view (**`ZC_##Flight`**). Annotate the view element **`CurrencyCode`** with annotation **`@Consumption.valueHelpDefinition: [{ }]`**. Use **`I_CurrencyStdVH`** as the value for the subannotation **`entity.name`** and **`Currency`** as the value for the subannotation **`entity.element`**.

Note:
`Currency` is the name of the key element of CDS view **`I_CurrencyStdVH`**.

   a) Open the definition of CDS view entity **`ZC_##FLIGHT`** where **`##`** is your group number.

   b) Before view element **`CurrencyCode`**, add the following code:

```
@Consumption.valueHelpDefinition: [{ entity.name:
'I_CurrencyStdVH',
                                    entity.element: 'Currency' }]
```

2. Activate the projection view definition.
   a) Press **`Ctrl + F3`** to activate the projection view definition.

Task 4: Verify the Result

Restart the preview of the OData UI Service and verify the changes to the user interface.

1. Restart the preview of the OData Service.
   a) Close the browser window or browser tab that contains the preview.
   b) Open your service binding. In the *Entity Set and Associations* list on the right-hand side, first choose entity *Flight* and then choose *Preview...*.
2. On the report list page, verify that the buttons *Create* and *Delete* have disappeared.
   a) Proceed as you have done in previous exercises.
3. Verify that there are two selection fields at the top of the report list page. Test their functionality.

Hint:
Be aware that the selection fields support the entry of multiple values and ranges.

   a) Enter a value in the selection field *Airline ID* (for example: “LH”) and choose *Go*.
   b) Repeat with other values.
4. For any data set, navigate to the object page. Verify that the *Delete* button has disappeared.
   a) Proceed as you have done in previous exercises.
5. Verify that field *Plane Type* is displayed after field *Flight Date* and before field *Flight Price*.
   a) Proceed as you have done in previous exercises.
6. Switch to change mode and check the order of the fields.