

# BC412

## ABAP Dialog Programming Using EnjoySAP Controls

SAP NetWeaver

Date \_\_\_\_\_  
Training Center \_\_\_\_\_  
Instructors \_\_\_\_\_  
\_\_\_\_\_  
Education Website \_\_\_\_\_

### Participant Handbook

Course Version: 63

Course Duration: 5 Days

Material Number: 50085285



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

## Copyright

Copyright © 2007 SAP AG. All rights reserved.

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

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

## Trademarks

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

## Disclaimer

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

# About This Handbook

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

## Typographic Conventions

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

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

## Icons in Body Text

The following icons are used in this handbook.

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

# Contents

<b>Course Overview .....</b>	<b>vii</b>
Course Goals .....	vii
Course Objectives .....	vii
<b>Unit 1: Introduction to EnjoySAP Controls .....</b>	<b>1</b>
Introduction to EnjoySAP Controls .....	2
<b>Unit 2: Control Framework Basics .....</b>	<b>7</b>
Control Framework Basics Overview.....	8
Controls and Screens .....	15
Changing the Attributes of a Control .....	37
Automation Queue.....	57
Control Events.....	78
<b>Unit 3: SAP Container .....</b>	<b>119</b>
The Container Concept .....	121
SAP Custom Container Control .....	126
SAP Dialog Box Container Control .....	130
SAP Docking Container Control .....	134
SAP Splitter Container Control.....	159
SAP Easy Splitter Container Control .....	190
<b>Unit 4: Context Menus .....</b>	<b>195</b>
Context Menus .....	196
<b>Unit 5: Text Edit Control.....</b>	<b>233</b>
Text Edit Control.....	234
<b>Unit 6: SAP Grid Control .....</b>	<b>265</b>
SAP Grid Control: Introduction.....	267
Transporting Data and Additional Information.....	286
Adapting the Grid Layout .....	293
Events .....	317
<b>Unit 7: Tree Control.....</b>	<b>347</b>
Tree Control: Introduction.....	349
Tree Control: Creating a Tree Control Instance .....	361
Tree Control: Creating a Hierarchy .....	380
Tree Control: Additional Functions of the Tree Model .....	389

Tree Control: Events .....	392
Tree Control: Additional Data in the Column and List Trees ...	419
<b>Unit 8: Drag&amp;Drop Functions .....</b>	<b>451</b>
Drag & Drop Functions .....	452
<b>Unit 9: Including Different Controls in Complex User Dialogs</b>	<b>495</b>
Including Different Controls in Complex User Dialogs .....	496

# Course Overview

Show functionality and usability of the Control Framework and of various basic controls and the technical specifications regarding the link between EnjoySAP Controls and screen based dialog elements.

## Target Audience

This course is intended for the following audiences:

- Developers and consultants who would like to create or change ABAP applications using the EnjoySAP Controls

## Course Prerequisites

### Required Knowledge

- SAPTEC – SAP NetWeaver
- Sound knowledge of ABAP programming techniques and ABAP Workbench – BC400 ABAP Workbench Fundamentals
- Knowledge of object-oriented programming – BC401 ABAP Objects
- Familiar with ‘classical’ dialog programming – BC410 Programming User Dialogs



## Course Goals

This course will prepare you to:

- Show functionality and usability of the Control Framework and of various basic controls.
- Present the technical specifications regarding the link between EnjoySAP Controls and screen based dialog elements.
- Develop user dialogs in ABAP using EnjoySAP Controls



## Course Objectives

After completing this course, you will be able to:

- Use the SAP Control Framework
- Use SAP container controls
- Use selected EnjoySAP Controls (Picture, HTML Viewer, Text Edit, SAP List Viewer, and Tree)
- Use special – control-based – mouse operations (context menus and Drag&Drop)

## SAP Software Component Information

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

- SAP Web Application Server 4.6C

# Unit 1

## Introduction to EnjoySAP Controls

### Unit Overview

This unit summarizes the goals of EnjoySAP Controls initiative.



### Unit Objectives

After completing this unit, you will be able to:

- Name the objectives of the EnjoySAP initiative
- Use an ALV Grid Control in a simple report

### Unit Contents

Lesson: Introduction to EnjoySAP Controls .....	2
---	---

# Lesson: Introduction to EnjoySAP Controls

## Lesson Overview

The Lesson will introduce the goals of the EnjoySAP Controls initiative.



## Lesson Objectives

After completing this lesson, you will be able to:

- Name the objectives of the EnjoySAP initiative
- Use an ALV Grid Control in a simple report

## Business Example

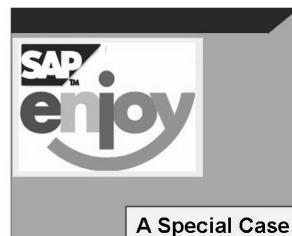
You are just getting started learning about EnjoySAP Controls. You want to become familiar with the objectives of the EnjoySAP Controls initiative.

## EnjoySAP Initiative



### For the user:

Applications are easy to learn, easy to adapt, and easy to use



### For the developer:

Applications can be programmed to be cross-platform using ABAP Objects classes

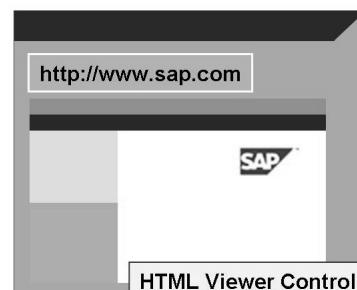
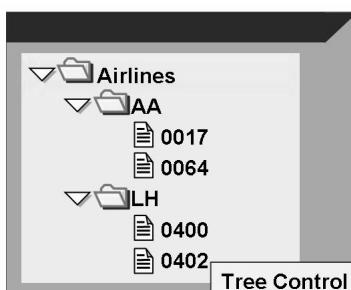


Figure 1: EnjoySAP Initiative

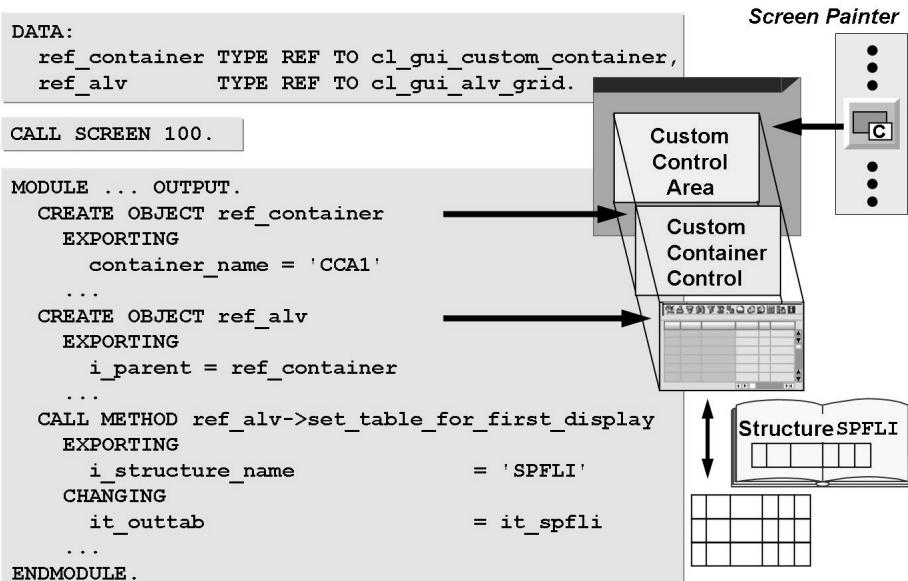
As of SAP Basis Release 4.6A, the ABAP Workbench has offered a range of **EnjoySAP Controls** that transfer dialog functions from the application server to the presentation server. You direct these controls using an object-oriented approach known as ABAP Objects classes, specifically the **SAP Control Framework**.



That is, you manipulate these controls from your ABAP program using method calls. The Control Framework sends the user's requests to the presentation server, where they are implemented in a cross-platform way.

You can offer the user the option of triggering events on the presentation server. These are then implemented by the Control Framework so that your ABAP program can react to them.

## Programming Steps When Implementing the SAP Grid Control



**Figure 2: Programming Steps When Implementing the SAP Grid Control**

The above source code fragment has already been discussed in the training course **BC400 (ABAP Workbench: Foundations and Concept)** and may be familiar to you already.

It is dealt with in more detail in the units **Introduction to the Control Framework**, **SAP Container**, and **SAP Grid Control**.



## Lesson Summary

You should now be able to:

- Name the objectives of the EnjoySAP initiative
- Use an ALV Grid Control in a simple report



## Unit Summary

You should now be able to:

- Name the objectives of the EnjoySAP initiative
- Use an ALV Grid Control in a simple report



# Unit 2

## Control Framework Basics

### Unit Overview

This unit covers all technical background information behind the EnjoySAP Control, except the Container technique which is covered in the next unit.



### Unit Objectives

After completing this unit, you will be able to:

- Describe the Controll Framework architecture
- Work with controls and screens
- Explain the relationship between screen, container and EnjoySAP control
- Display data in a SAP Picture control and SAP HTML Viewer on a screen
- Understand and use the attributes of a control
- Understand how the actions are transferred to the presentation server
- Describe the Automation Queue and its Runtime Performance
- Describe the events triggered by the EnjoySAP Controls and learn the steps necessary to react to them

### Unit Contents

Lesson: Control Framework Basics Overview .....	8
Lesson: Controls and Screens .....	15
Exercise 1: Controls and Screens.....	25
Lesson: Changing the Attributes of a Control .....	37
Exercise 2: Changing the Attributes of a Control.....	49
Lesson: Automation Queue .....	57
Exercise 3: Changing the Attributes of a Control Dynamically .....	65
Lesson: Control Events .....	78
Exercise 4: Control Events.....	101

## Lesson: Control Framework Basics Overview

### Lesson Overview

This lesson presents the Control Framework architecture.



### Lesson Objectives

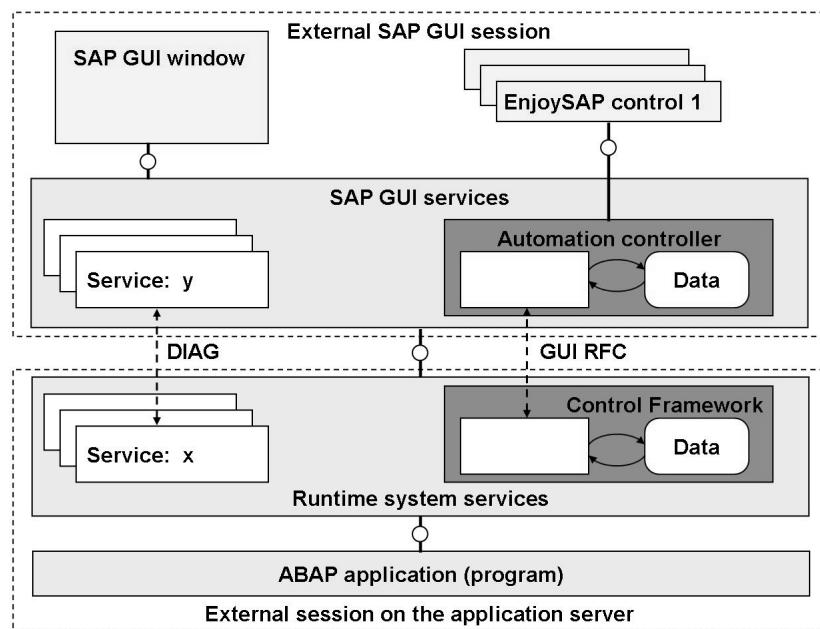
After completing this lesson, you will be able to:

- Describe the Control Framework architecture

### Business Example

You are just getting started learning about EnjoySAP Controls. You want to become familiar with the Control Framework architecture.

### Control Framework Architecture: Overview



**Figure 3: Control Framework Architecture: Overview**

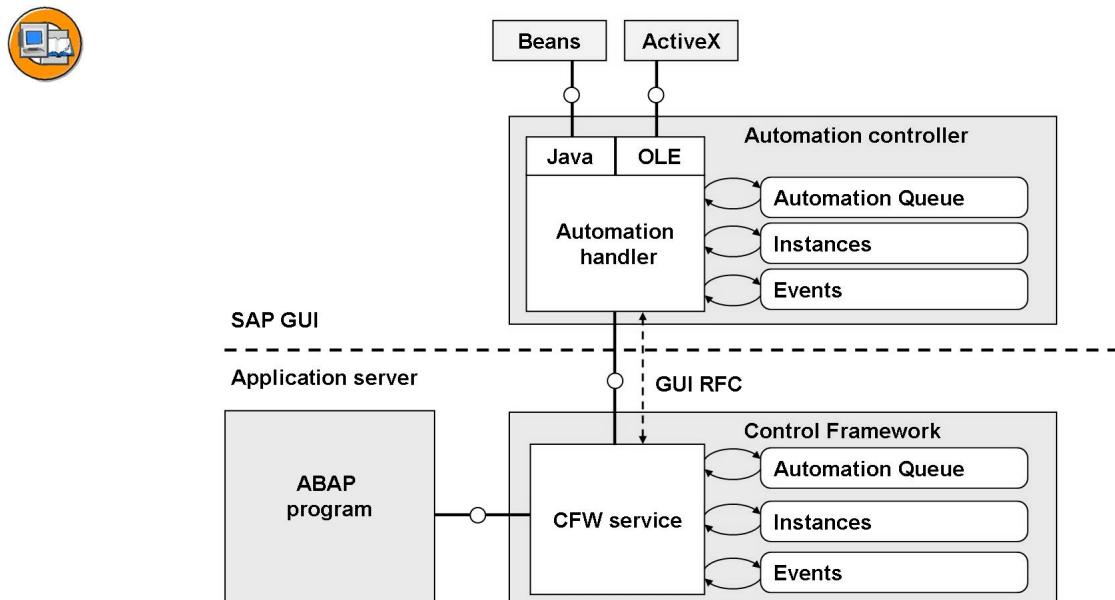
The graphical user interface in the SAP R/3 System is based on SAP GUI windows (screens from the technical view of the programmer). Each user dialog in the system is implemented using screens in an application program.

At runtime, application programs and screens communicate using runtime system services (the ABAP processor and screen processor) and other services from the SAP GUI (window management, dialog box level, data transfer, and so on).

From R/3 Release 4.5 onwards, developers have been able to use graphical elements in SAP GUI windows, known as controls, as well as those provided by the Screen Painter. These additional screen elements are known as **controls**. Controls are reusable standalone software components. As a developer, you can use controls in your user dialogs for a range of functions.

Communication between the control and your application takes place through different channels to “classic” screen elements. From the SAP R/3 Release 4.6A onwards, the communication channel for all controls is the SAP Control Framework (**CFW**). The Control Framework is implemented using special services in the SAP GUI and the runtime system on the application server (the Automation Controller on the presentation server and the CFW service on the application server).

## CFW Architecture in Detail



**Figure 4: CFW Architecture in Detail**

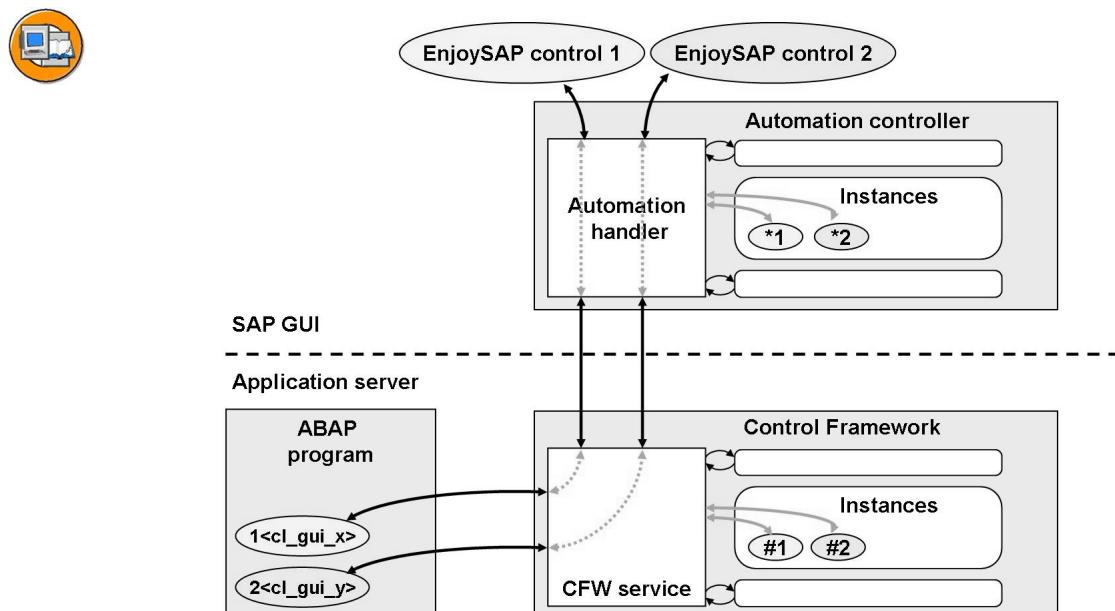
The two services of the Control framework – the CFW service in the runtime system and the Automation Handler in the SAP GUI – each control their own data pertaining to a control.

- A list of all instances (EnjoySAP controls) with which they have to communicate
- The automation queue
- A list of the events of the EnjoySAP controls that the application program is to handle.

The aim of the Control Framework is to optimize communication between controls and the application program.

EnjoySAP controls are installed on the presentation server, either as Java Beans or ActiveX controls. Communication between the EnjoySAP control instances and the automation handler takes place using the OLE or Java interface in the Automation Controller.

## Controls and ABAP Objects



**Figure 5: Controls and ABAP Objects**

Each EnjoySAP control with which an application program communicates is an object (instance) in the sense of object orientation on the presentation server. The communication between the program and the instances is implemented using method calls and the events of the objects.

From Release 4.6, there are **proxy classes** in the global class library for all controls. These access the Control Framework in a uniform manner and encapsulate the control-specific communication. We call this kind of class a control wrapper.

When you use a control in an ABAP program, there are actually **two** objects involved. There is an instance in the ABAP program, and an instance of the control on the presentation server. The communication between the ABAP program and the control is implemented as follows:

- The ABAP program always communicates with the proxy object (method calls, setting and retrieving attribute values). The proxy object uses the CFW and executes the corresponding operation on the presentation server instance.
- When an action comes from the presentation server, the CFW directs it to the proxy object, which then communicates with the ABAP program.

The CFW contains a list of the active instances both on the presentation server and on the application server.

## Automation Queue

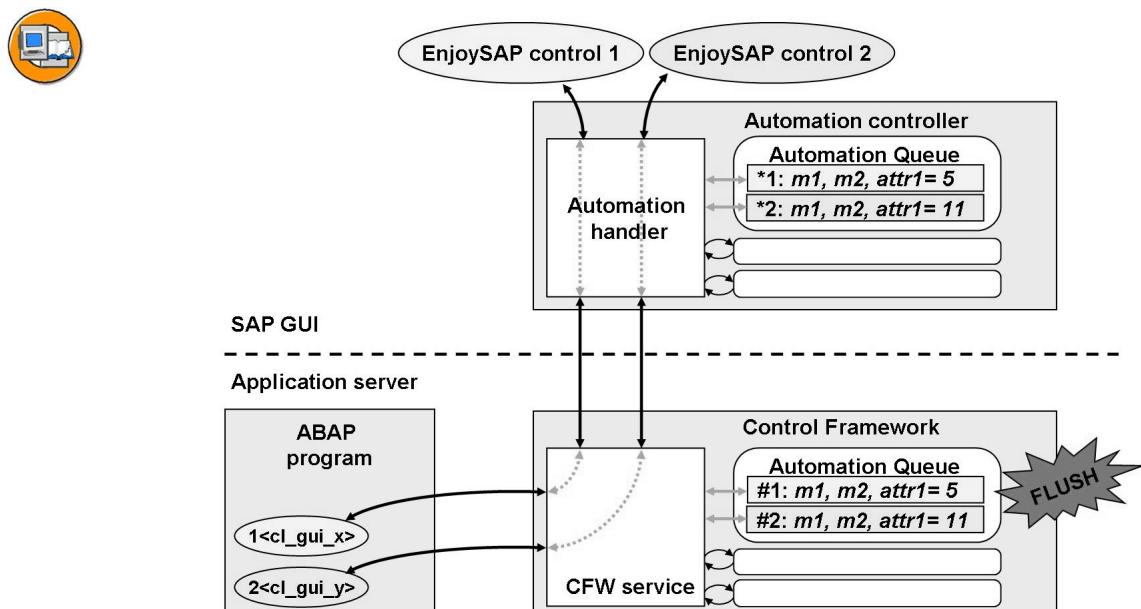


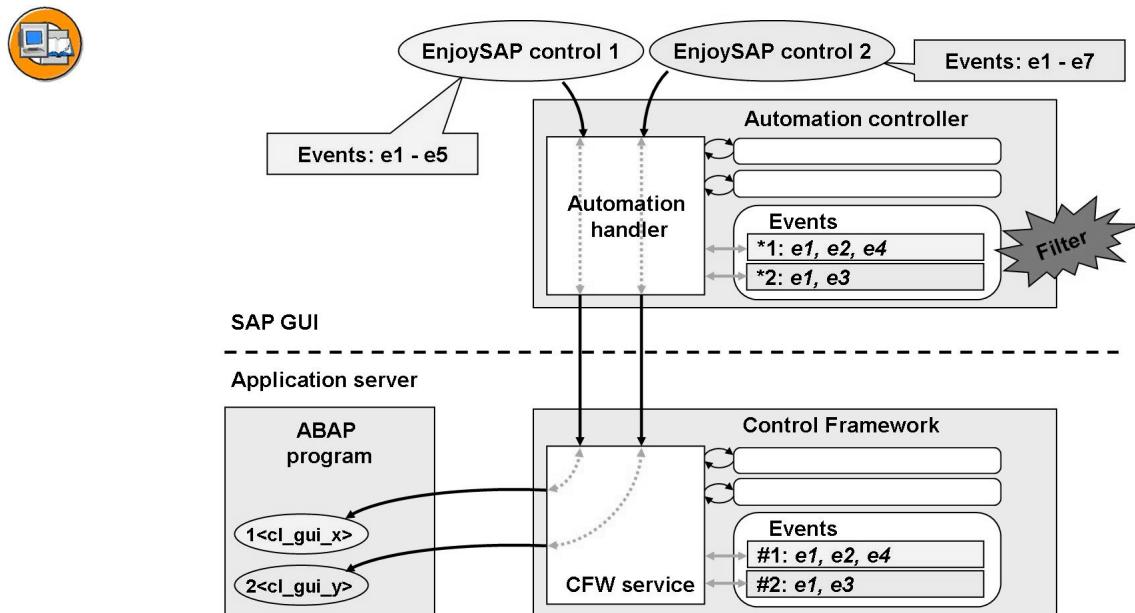
Figure 6: Automation Queue

Every communication step with a presentation server control necessitates a transfer of data over the network between the application and presentation servers. This communication is separate from the communication involving other screen elements, and can therefore cause increased network load. (Remember – screens are always transferred as a whole unit.)

To improve performance, method calls and attribute operations in the Control Framework are collected in the **Automation Queue**. The contents of the queue are automatically sent to the presentation server (that is the queue is **FLUSHed**) at

particular synchronization points. The corresponding operations are then carried out there. In the same step, all values that had been collected at the presentation server are passed back to the application server.

## Events



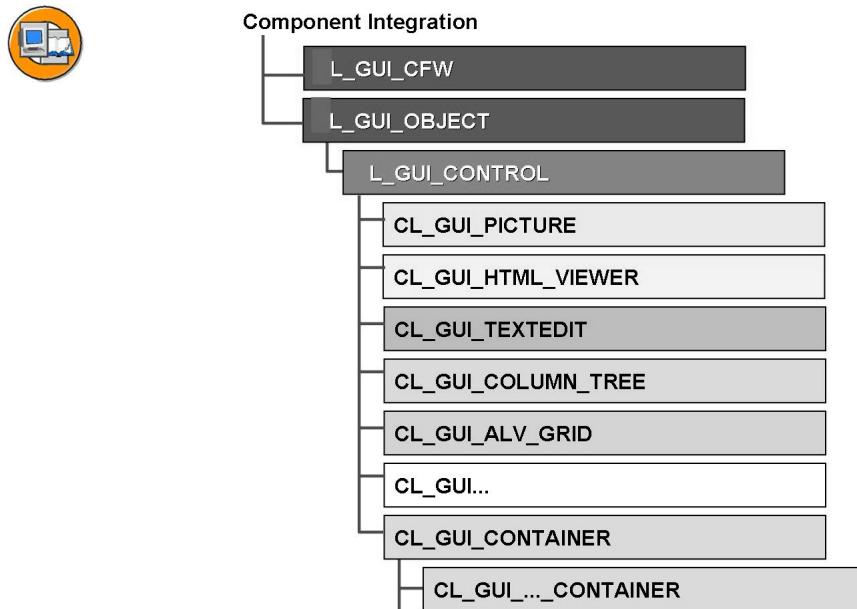
**Figure 7: Events**

The control on the presentation server can have events, which could be triggered by user actions on the screen. These events are transferred back to the appropriate proxy objects in the ABAP program with the help of the Control Framework. The proxy object then handles the event using the normal event concept of ABAP Objects.

There are many events that each presentation server control could trigger. However, as a rule, only a few of these will be relevant for an application program.

Since every event handled by the CFW involves a communication step and a data transfer, the event filter is set in the CFW as follows: The initial state of the filter is not to transfer any events. If you want to handle events in your ABAP program, you must construct your own filter.

## Control Framework: Class Hierarchy



**Figure 8: Control Framework: Class Hierarchy**

The proxy objects for EnjoySAP controls are instances of global classes. Global classes are actively integrated in the ABAP Workbench.

All proxy classes for EnjoySAP controls inherit attributes and methods from the classes CL\_GUI\_OBJECT and CL\_GUI\_CONTROL.

CL\_GUI\_OBJECT implements the interface IF\_CACHED\_PROP.

All classes used in control programming have the prefix CL\_GUI\_.... . Some of them implement global interfaces, such as IF\_GUI\_DYNAMIC\_DATA, IF\_DRAGDROP, etc.

There is an additional class, CL\_GUI\_CFW, which provides service methods. These are used extensively in the individual control wrappers, but you can also use them directly in your programs.

You can display individual classes either in the Class Builder or in the Object Navigator. To display the inheritance hierarchy, use the Class Browser.

The classes derived from CL\_GUI\_CONTAINER are control wrappers with a special function within the Control Framework.



## Lesson Summary

You should now be able to:

- Describe the Controll Framework architecture

# Lesson: Controls and Screens

## Lesson Overview

This lesson presents the relationship between container, control, and screen.



## Lesson Objectives

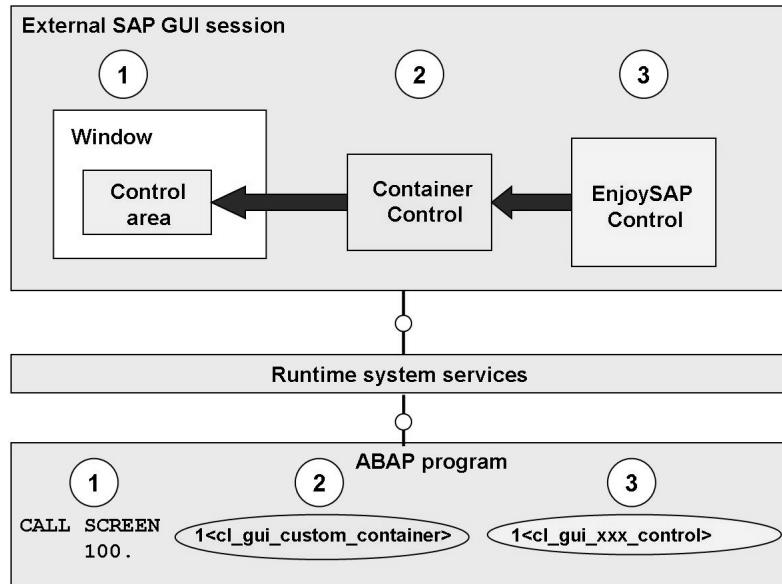
After completing this lesson, you will be able to:

- Work with controls and screens
- Explain the relationship between screen, container and EnjoySAP control
- Display data in a SAP Picture control and SAP HTML Viewer on a screen

## Business Example

The project team of which you are a member has decided to implement an application using EnjoySAP Controls. You want to understand the controls and screens.

## Controls and Screens



**Figure 9: Controls and Screens**

You cannot create a standalone instance of a control on the presentation server. You must always have two other components – a screen and a special control called a **container control**.

The container control attaches a control to a screen.

The screen contains a reserved area for the container (like a subscreen area, reserved for a subscreen)

In your program, you assign an instance of the container control class `cl_gui_custom_container` to the reserved area on the screen.

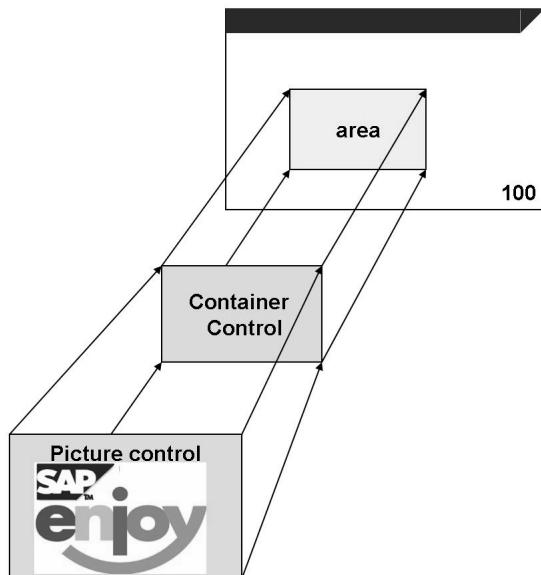
Finally, you assign an instance of the control you want to display to the container control instance.

There are other ways of attaching a container instance to a screen. We will look at these in a later unit. In this unit, we will only use the case explained on this page.

The first two steps help to integrate the presentation server control into normal screen processing (PBO, PAI, resizing).

Since each presentation server control has a proxy instance in the ABAP program, you will be communicating with two ABAP Objects instances and at least one screen. (There is a total of four instances – two in the ABAP program, two on the presentation server).

## Using Objects Graphically



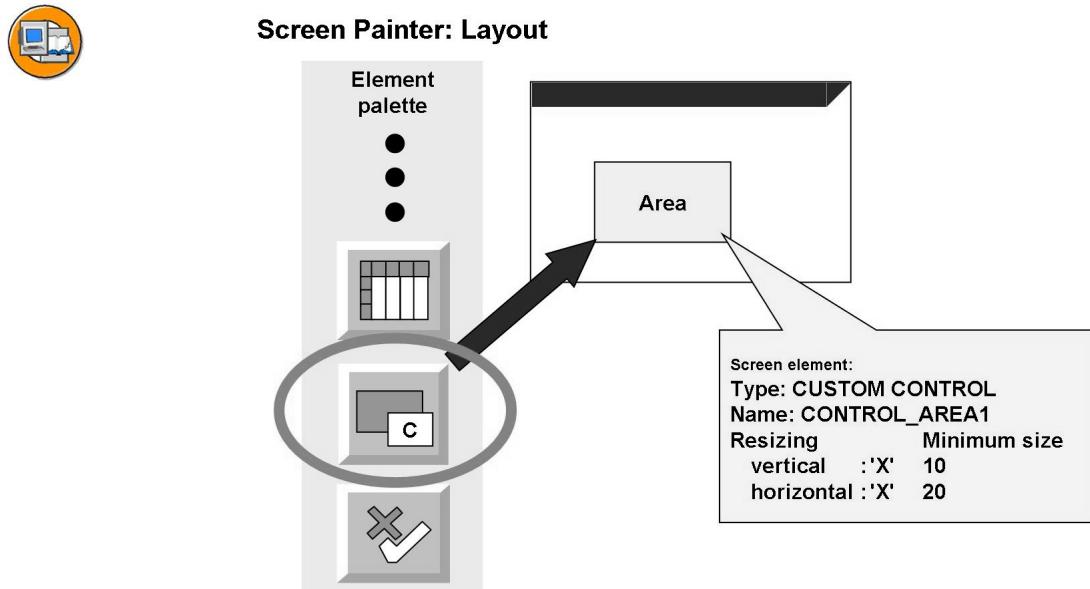
**Figure 10: Using Objects Graphically**

If we look at it graphically, attaching an EnjoySAP control to a screen is just like nesting two areas, one inside the other:

- First, we define an area in which to display the control.
- Next, we attach a container control to the area.
- The container control holds the actual control that we want to use (for example, to display a picture).

We will now look at what you need to do to carry out these three steps in an ABAP program.

## Creating a Custom Control Area



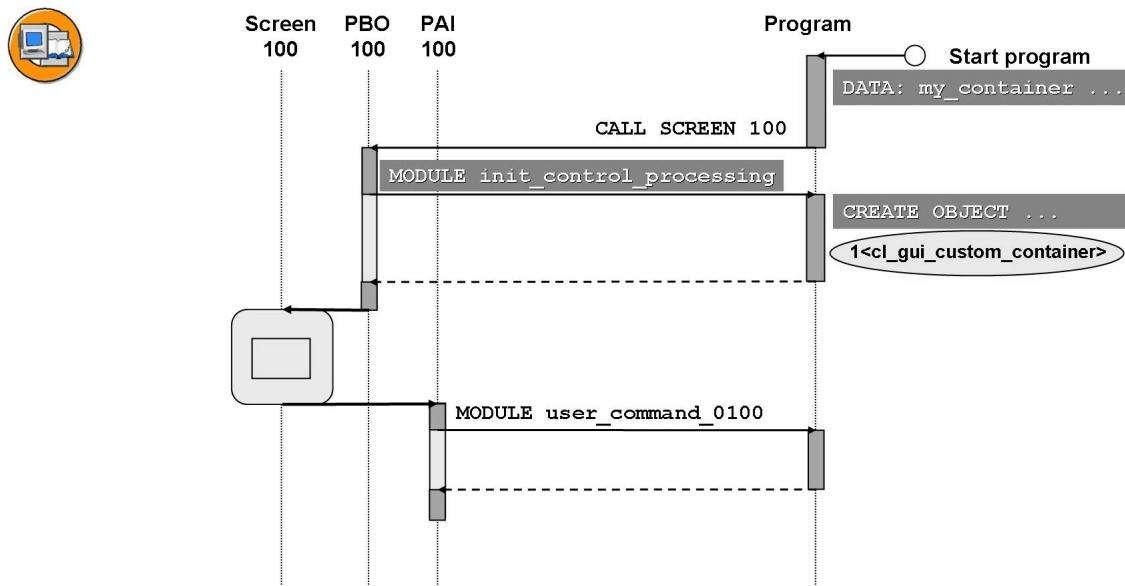
**Figure 11: Creating a Custom Control Area**

To reserve a screen area for an EnjoySAP control, start the Graphical Layout Editor in the Screen Painter. (That is, choose the Layout pushbutton).

The element toolbar at the left-hand edge of the screen contains an icon with the label Custom Control. This works very similarly to the Subscreen function:

- Choose Custom Control. Define the custom control area on the screen as follows: Define the top left-hand corner of the area by single-clicking. Hold the mouse button down and drag the container area out to the required size. Then release the mouse button.
- You can change the size and position of the area at any time by clicking one of the “handles” at the edge of the object and dragging it to its new size. This handling is similar to that of subscreen areas.
- Give the screen element a name (in the example: CONTROL\_AREA1).
- The Resizing vertical and Resizing horizontal attributes allow you to specify whether the control area should be resized when the user changes the window size. If you set the attributes, you can specify the minimum size to which the area can be reduced in the Lines min. and Columns min. fields. The maximum size of the area is the size in which you created it.

## Creating a Container Control Instance: Process Flow



**Figure 12: Creating a Container Control Instance: Process Flow**

The next step is to create a container control instance. You must do this in your program before the screen containing the control is displayed. An appropriate point in the program is therefore the PBO event of the screen containing the custom control area.

To make your program easier to follow, it is a good idea to place all control programming in a separate module. In our example, we have a module called `init_control_processing`.

Before you can create a container control instance, you need a reference variable for it.

The following slides discuss each step in this process in more detail.

## Syntax: Defining a Reference Variable



<code>DATA: &lt;variable_name&gt; "</code>	Name of reference variable
<code>TYPE REF TO</code>	Specifies that the variable is an object reference
<code>&lt;class_name&gt;.</code>	Name of a class (global or local)

**Figure 13: Syntax: Defining a Reference Variable**

If you write a program that uses instances of classes (“object-oriented programming”), you need a pointer to the objects (instances) that the Basis system creates and administers in main memory. You administer these pointers in your ABAP programs using reference variables.

To create a reference variable in a program, use the **DATA** statement and assign a name to it. To specify the type of the variable (field), use the addition **TYPE REF TO <class\_name>**. This addition declares a data object that can contain a pointer to an instance. After **TYPE REF TO** you must specify the name of a class. The reference variable can then contain memory addresses to instances of that class. For further details about reference variables (static types, dynamic types, and so on), refer to the keyword documentation for the **DATA** statement.

The class to which you refer when you define a reference variable can be either local or global. The classes used for controls are global classes. They are defined in the Class Builder and can be accessed from any program in the system.

## Syntax: Creating Instances



```

DATA: <object_name> TYPE REF TO <class_name>.
CREATE OBJECT <object_name>
  EXPORTING
    <parameter_name_1> = <program_value_1>
    ...
    <parameter_name_n> = <program_value_n>
  EXCEPTIONS
    <exception_name_1> = <return_code_1>
    ...
    <exception_name_m> = <return_code_m>.
  
```

Constructor interface

**Figure 14: Syntax: Creating Instances**

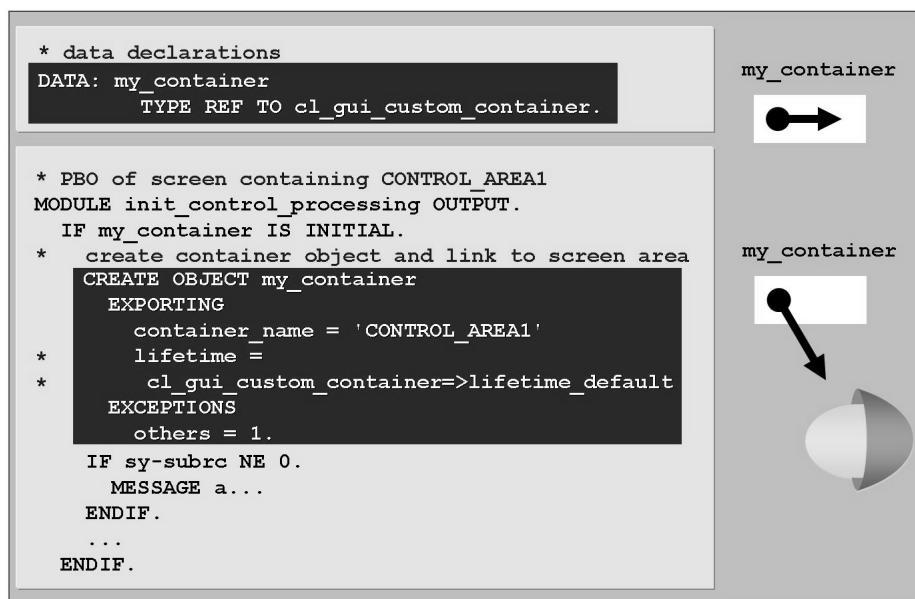
The **CREATE OBJECT <object\_name>** statement creates an instance of the class specified after **TYPE REF TO** in the definition of the reference variable **<object\_name>**.

If the class that you want to instantiate has a method called **constructor**, this method is executed automatically when the instance is created. The constructor method can only have **IMPORTING** parameters and **EXCEPTIONS** in its interface. You fill the interface parameters in the **CREATE OBJECT** statement. In the **EXPORTING** parameters of the **CREATE OBJECT** statement, you must specify values for all of the obligatory **IMPORTING** parameters in the interface of the constructor method.

The syntax of the interface in the call is similar to that for function modules:

- The names of the interface parameters of the method are on the left-hand side of the expressions in the **EXPORTING** section. On the right of the equals sign, you specify literals or program variables that you want to pass to the method.
- You can assign return codes (in the form of integers) to the exceptions listed in the interface. Directly after the **CREATE OBJECT** statement, you can find out the value of the return code from the system field **SY-SUBRC**.

## Example: Creating a Container Control Instance



**Figure 15: Example: Creating a Container Control Instance**

To create a **reference variable** for your container control instance, use the **DATA** statement to create a variable that has the type **TYPE REF TO cl\_gui\_custom\_container**. In our example, we have called it **my\_container**.

You can now create the proxy instance for the container control **on the presentation server** (an instance of class **cl\_gui\_custom\_container**) using the ABAP statement **CREATE OBJECT**.

In the **IMPORTING** parameter **CUSTOMER\_NAME** of the **CREATE OBJECT** statement, you specify the name of the custom control area you defined on the screen. ("CONTROL\_AREA1").

In the optional **IMPORTING** parameter **LIFETIME**, you specify the “**lifetime**” of your container, using one of three class constants:

- <classname>=>**lifetime\_imode** specifies that the control remains “alive” as long as the internal session remains open (as long as no LEAVE PROGRAM has been executed).
- <classname>=>**lifetime\_dynpro** specifies that the control remains alive as long as the instance of the screen exists – that is, remains on the screen stack. Specifying this constant also controls the **visibility** of the control. It is only shown when the screen on which it was generated is active.
- If you use <classname>=>**lifetime\_default** the control assumes the lifetime of the container. If the control is not bound to a container (for example, because it is itself a container), the lifetime is set to <classname>=>**lifetime\_imode**.

## Ensuring that Controls Are Generated Only Once

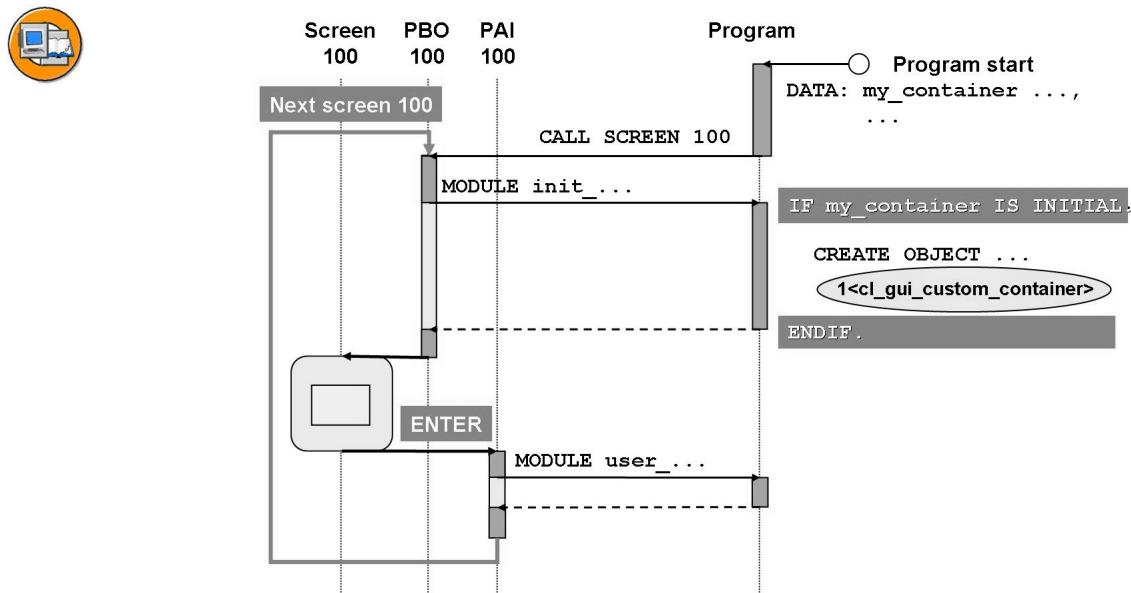
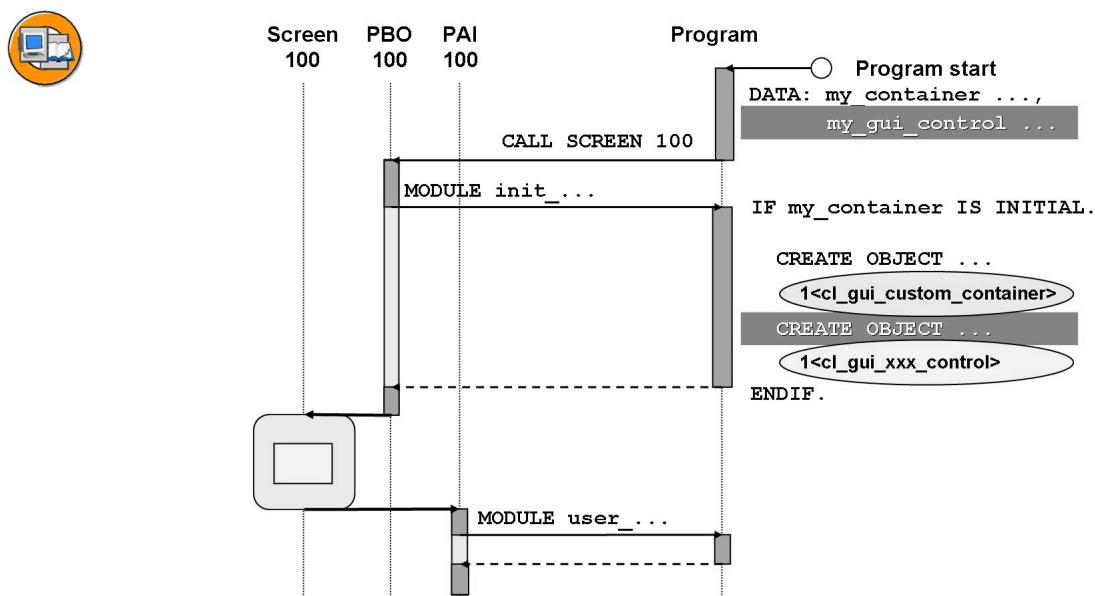


Figure 16: Ensuring that Controls Are Generated Only Once

If you place your control-specific processing within the flow logic of the container screen (the screen on which you are going to display the control), you need a mechanism to ensure that the steps that create the control are only carried out the first time the screen is called. Otherwise there is a danger that a new object is created every time the screen is processed:

- The statically-defined next screen for the container screen is usually itself.
- Every action on the screen that triggers a PAI (such as Enter) causes the system to process the screen again after the PAI event.
- The results of any control operations (method calls) processed during PAI would not be displayed on the screen, since another instance would be linked to the container screen in each PBO.
- You can stop this happening by using a flag variable - in the above example, the data object my\_container.
- Alternatively you can use a data object of your own as a switch. For example: DATA: first\_time. IF first\_time IS INITIAL. first\_time = "X" CREATE OBJECT my\_container ... ENDIF.

## Creating an EnjoySAP Control Instance: Process Flow



**Figure 17: Creating an EnjoySAP Control Instance: Process Flow**

You create the EnjoySAP control instance in the same way that you created the instance for the container control.

The instance of the EnjoySAP control must be created after the container control instance, but before the screen is sent to the presentation server.



## Example: Creating an Instance of the SAP HTML Viewer

```
* data declarations
DATA: my_container      TYPE REF TO cl_gui_custom_container,
      my_html_viewer   TYPE REF TO cl_gui_html_viewer.

MODULE start_control_handling OUTPUT.

IF my_container IS INITIAL.
  ...
  *  create html viewer object and link to container
  CREATE OBJECT my_html_viewer
    EXPORTING
      parent = my_container
    EXCEPTIONS
      others = 1.

  IF sy-subrc NE 0.
    MESSAGE a...
  ENDIF.
ENDIF.
```

**Figure 18: Example: Creating an Instance of the SAP HTML Viewer**

To create an **SAP HTML Viewer** instance, you need a reference variable declared with reference to class CL\_GUI\_HTML\_VIEWER.

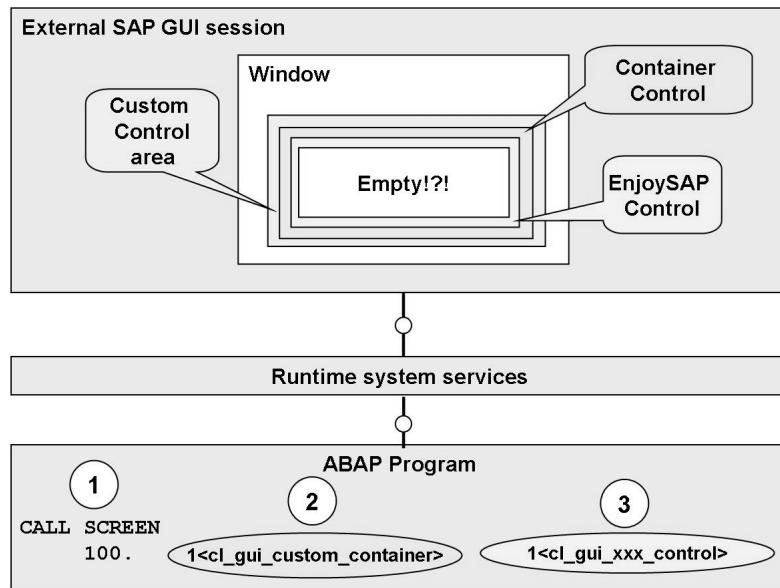
In the CREATE OBJECT statement, you must fill the parameter PARENT with the reference to the container control instance that you have already created.

If an error occurs when the system tries to create the instance, your program should react with a termination (type A) message.

The instance of the SAP HTML Viewer Control is linked to the container control whose reference variable you specified as the parent control in the CREATE OBJECT statement. This also links it to the container screen.

The procedure for creating a SAP Picture Control is similar: The appropriate reference variable is created with reference to the class CL\_GUI\_PICTURE.

## Result: Screen, Container, and EnjoySAP Control



**Figure 19: Result: Screen, Container, and EnjoySAP Control**

- The result of the steps up to this point is that we have now created an EnjoySAP control on a screen. However, we have still not seen anything in the SAP GUI window, since the controls are not displaying anything. We still need to supply the presentation server control with data that it can display.

# Exercise 1: Controls and Screens

## Exercise Objectives

After completing this exercise, you will be able to:

- Create custom control areas (that is, screen elements) in the Screen Painter
- Create and use reference variables for instances of ABAP Objects
- Create instances of container controls and picture controls
- Assign container control instances to custom controls
- Link picture control instances with containers
- Find information on global classes in the Class Library in the Class Builder.

## Business Example

Copy a template consisting of an ABAP program that calls a screen. The screen has a GUI status, plus the standard navigation functions.

This program will be used later to display a picture control.

**Program:** ZBC412\_##\_CFW\_EX1

**Template:** SAPBC412\_CFWT\_EXERCISE1

**Model solution:** SAPBC412\_CFWS\_EXERCISE1

where ## is the group number.

## Task 1:

Perform the following tasks:

1. Copy the template SAPBC412\_CFWT\_EXERCISE1 (including all its sub-objects) to the program ZBC412\_##\_CFW\_EX1 and get to know the features of this copy.
2. On screen 100, create a custom control area with the following attributes:

Attribute	Value	Suggested value
Element type	Custom control	
Name	CONTROL_AREA1	
Row	Any	2
columns	Any	2
defLength/ visLength	Any	50
Height	Any	11

*Continued on next page*

Attribute	Value	Suggested value
Resizing	yes	
Rows (minimum)	Any	3
Columns (minimum)	Any	5

Activate the screen.

### Task 2:

Create a container instance in a PBO module for screen 100:

1. Create a reference variable for the container instance (We suggest you use the name *ref\_container*). Use the global class type *cl\_gui\_custom\_container*.
2. Screen 100 already has a PBO module *init\_control\_processing*. Extend this module so that the system generates a container control instance when it executes the program for the first time. Pass the name of the custom control area you created to the interface parameter *container\_name*. Catch the generic exception, OTHERS, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing. Use the **termination message 010** from the **message class bc412**.

### Task 3:

Create a picture control instance. Encapsulate this code in the PBO module *init\_control\_processing*, which you already used in Task 2 of this exercise.

1. Create a reference variable for the picture control instance (We suggest you use the name *ref\_picture*). Use the global class type *cl\_gui\_picture*.
2. Extend the *init\_control\_processing* module so that the system generates a picture control instance when it executes the program for the first time. Pass your container control reference to the interface parameter *parent*. Catch the generic exception, OTHERS, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing. Use the **termination message 011** from the **message class bc412**.

### Task 4:

Activate and test.

1. Activate and test your program. Even if your program runs correctly, you will still see an empty screen (and an empty control area). If an error occurs, the system either displays a type A message or a runtime error occurs.

*Continued on next page*

The message class BC412 has already been set globally for this program (see the REPORT statement).

The buttons that are displayed on the screen attached to the template do not yet have functions. Assigning these functions is part of a subsequent exercise in this unit.

# Solution 1: Controls and Screens

## Task 1:

Perform the following tasks:

1. Copy the template SAPBC412\_CFWT\_EXERCISE1 (including all its sub-objects) to the program ZBC412\_##\_CFW\_EX1 and get to know the features of this copy.
  - a) –
2. On screen 100, create a custom control area with the following attributes:

Attribute	Value	Suggested value
Element type	Custom control	
Name	CONTROL_AREA1	
Row	Any	2
columns	Any	2
defLength/ visLength	Any	50
Height	Any	11
Resizing	yes	
Rows (minimum)	Any	3
Columns (minimum)	Any	5

Activate the screen.

- a) –

## Task 2:

Create a container instance in a PBO module for screen 100:

1. Create a reference variable for the container instance (We suggest you use the name *ref\_container*). Use the global class type *cl\_gui\_custom\_container*.
  - a)
 

```
DATA: ref_container TYPE REF TO cl_gui_custom_container.
```
2. Screen 100 already has a PBO module *init\_control\_processing*. Extend this module so that the system generates a container control instance when it executes the program for the first time. Pass the name of the custom control area you created to the interface parameter *container\_name*. Catch the generic exception, OTHERS, instead of evaluating each exception

*Continued on next page*

individually. When an exception occurs, make sure your program stops processing. Use the **termination message 010** from the **message class bc412**.

a)

```
IF ref_container IS INITIAL." prevent re-processing on ENTER
  CREATE OBJECT ref_container
    EXPORTING
      container_name = 'CONTROL_AREA1'
    EXCEPTIONS
      others = 1.
    IF sy-subrc NE 0.
      MESSAGE a010.          " cancel program processing
    ENDIF.
  ENDIF.
```

### Task 3:

Create a picture control instance. Encapsulate this code in the PBO module *init\_control\_processing*, which you already used in Task 2 of this exercise.

1. Create a reference variable for the picture control instance (We suggest you use the name *ref\_picture*). Use the global class type *cl\_gui\_picture*.
  - a)  
DATA: ref\_picture TYPE REF TO cl\_gui\_picture.
2. Extend the *init\_control\_processing* module so that the system generates a picture control instance when it executes the program for the first time. Pass your container control reference to the interface parameter *parent*. Catch the generic exception, OTHERS, instead of evaluating each exception

*Continued on next page*

individually. When an exception occurs, make sure your program stops processing. Use the **termination message 011** from the **message class bc412**.

a)

```

IF ref_container IS INITIAL.      " prevent re-processing on ENTER
*      create container object and link to screen area
CREATE OBJECT ref_container
EXPORTING
  container_name = 'CONTROL_AREA1'
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
  MESSAGE a010.                  " cancel program processing
ENDIF.

*      create picture control and link to container object
CREATE OBJECT ref_picture
EXPORTING
  parent = ref_container
EXCEPTIONS
  others = 1.
IF sy-subrc NE 0.
  MESSAGE a011.                  " cancel program processing
ENDIF.
ENDIF.
```

## Task 4:

Activate and test.

1. Activate and test your program. Even if your program runs correctly, you will still see an empty screen (and an empty control area). If an error occurs, the system either displays a type A message or a runtime error occurs.

The message class BC412 has already been set globally for this program (see the REPORT statement).

The buttons that are displayed on the screen attached to the template do not yet have functions. Assigning these functions is part of a subsequent exercise in this unit.

a) -

*Continued on next page*

## Result

### Screen flow logic

#### SCREEN 100

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.
  MODULE init_control_processing.
PROCESS AFTER INPUT.
  MODULE exit_command_0100 AT EXIT-COMMAND.
  MODULE user_command_0100.
```

### ABAP program

#### *Data Declarations*

```
*&-----*
*& Report  SAPBC412_CFWS_EXERCISE1 *
*&
*&-----*
```

```
REPORT  sapbc412_cfws_exercise1 MESSAGE-ID bc412.
* data declarations
*   screen specific
DATA:
  ok_code      TYPE sy-ucomm,          " command field
  copy_ok_code LIKE ok_code,          " copy of ok_code
  l_answer     TYPE c,                " return flag (used in
                                         " standard user dialogs)

*   control specific: object references
  ref_container TYPE REF TO cl_gui_custom_container,
  ref_picture    TYPE REF TO cl_gui_picture,
```

### Main ABAP Program: Event Blocks

```
* start of main program
START-OF-SELECTION.

CALL SCREEN 100.           " container screen for SAP-Enjoy
                           " controls

* end of main program
```

### Modules

*Continued on next page*

```

*-----*
*&      Module  INIT_CONTROL_PROCESSING  OUTPUT
*-----*
*      Implementation of EnjoySAP control processing:
*      1)  create container object and link to screen area
*      2)  create picture object and link to container
*-----*
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.          " prevent re-processing on ENTER
*      create container object and link to screen area
  CREATE OBJECT ref_container
    EXPORTING
      container_name = 'CONTROL_AREA1'
  EXCEPTIONS
    others = 1.

  IF sy-subrc NE 0.
    MESSAGE a010.                      " cancel program processing
  ENDIF.

*      create picture control and link to container object
  CREATE OBJECT ref_picture
    EXPORTING
      parent = ref_container
  EXCEPTIONS
    others = 1.

  IF sy-subrc NE 0.
    MESSAGE a011.                      " cancel program processing
  ENDIF.

  ENDIF.
ENDMODULE.                                " INIT_CONTROL_PROCESSING  OUTPUT

*-----*
*&      Module  USER_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type ' ':
*      - push buttons on the screen
*      - GUI functions
*-----*
MODULE user_command_0100 INPUT.
  copy_ok_code = ok_code.
  CLEAR ok_code.

```

*Continued on next page*

```

CASE copy_ok_code.
  WHEN 'BACK'.
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *          DEFAULTOPTION = 'Y'
        textline1   = text-004
        textline2   = text-005
        titel       = text-007
        cancel_display = ' '
      IMPORTING
        answer      = l_answer.

CASE l_answer.
  WHEN 'J'.
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
  WHEN 'N'.
    SET SCREEN sy-dynnr.
ENDCASE.

WHEN 'STRETCH'.           " picture operation: stretch to fit area
*   to be implemented later
WHEN 'NORMAL'.            " picture operation: fit to normal size
*   to be implemented later
WHEN 'NORMAL_CENTER'.    " picture operation: center normal size
*   to be implemented later
WHEN 'FIT'.               " picture operation: zoom picture
*   to be implemented later
WHEN 'FIT_CENTER'.       " picture operation: zoom and center
*   to be implemented later

ENDCASE.

ENDMODULE.                  " USER_COMMAND_0100  INPUT

*&-----*
*&     Module STATUS_0100  OUTPUT
*&-----*
*     Set GUI for screen 0100
*&-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'STATUS_NORM_0100'.
  SET TITLEBAR 'TITLE_NORM_0100'.

```

*Continued on next page*

```

ENDMODULE.                                     " STATUS_0100  OUTPUT

*-----*
*&      Module  EXIT_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type 'E'.
*-----*
MODULE exit_command_0100 INPUT.

CASE ok_code.

WHEN 'CANCEL'.           " cancel current screen processing
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
        EXPORTING
            *
            DEFAULTOPTION = 'Y'
            textline1     = text-004
            textline2     = text-005
            titel         = text-006
            cancel_display = ' '
        IMPORTING
            answer        = l_answer.

CASE l_answer.

WHEN 'J'.
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.

WHEN 'N'.
    CLEAR ok_code.
    SET SCREEN sy-dynnr.
ENDCASE.

WHEN 'EXIT'.                  " leave program
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
        EXPORTING
            *
            DEFAULTOPTION = 'Y'
            textline1     = text-001
            textline2     = text-002
            titel         = text-003
            cancel_display = 'X'
        IMPORTING
            answer        = l_answer.

CASE l_answer.

WHEN 'J' OR 'N'.             " no data to update
    PERFORM free_control_ressources.
    LEAVE PROGRAM.

WHEN 'A'.

```

*Continued on next page*

```
CLEAR ok_code.  
SET SCREEN sy-dynnr.  
ENDCASE.  
ENDCASE.  
ENDMODULE.                                     " EXIT_COMMAND_0100  INPUT
```

## Subroutines

```
*&-----  
*&      Form  free_control_ressources  
*&-----  
*      Here you should implement: Free all control related  
*      ressources.  
*-----  
*      no interface  
*-----  
FORM free_control_ressources.  
* to be implemented later  
ENDFORM.                                         " free_control_ressources
```



## Lesson Summary

You should now be able to:

- Work with controls and screens
- Explain the relationship between screen, container and EnjoySAP control
- Display data in a SAP Picture control and SAP HTML Viewer on a screen

# Lesson: Changing the Attributes of a Control

## Lesson Overview

This lesson presents the SAP HTML Viewer and Picture controls.



## Lesson Objectives

After completing this lesson, you will be able to:

- Understand and use the attributes of a control

## Business Example

The project team of which you are a member has decided to implement an application using EnjoySAP Controls Picture and HTML Viewer. You need to understand the features of these controls and the methods available to use them.

## Pointers to Data in the Form of URLs

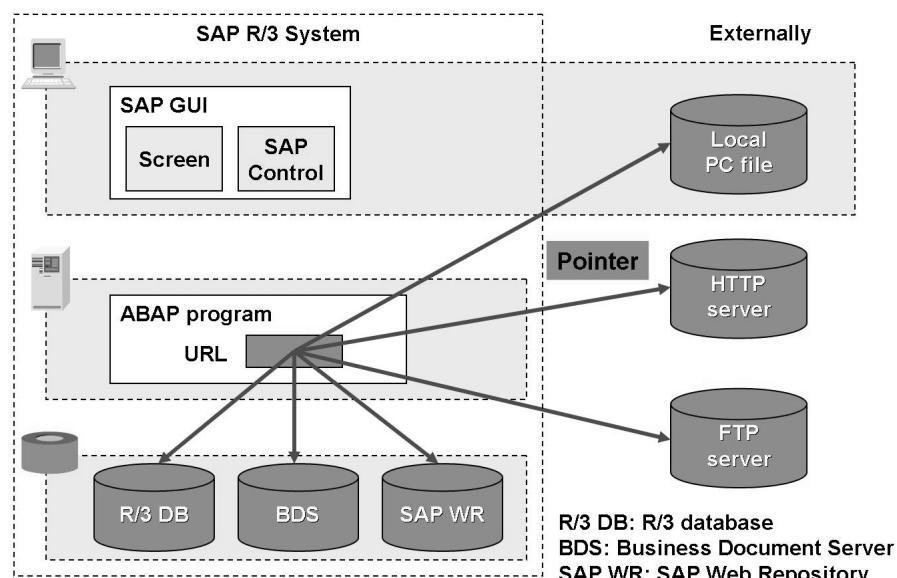


Figure 20: Pointers to Data in the Form of URLs

Both the SAP Picture Control and SAP HTML Viewer Control access the data they display using URLs (uniform resource locators).

URLs point to data sources. In ABAP, you can administer them using a type C variable.

The data can be stored either in the SAP System or externally.

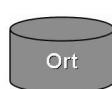
Within the SAP System, you can save files in the R/3 database as a cluster table, in the **Business Document Server** (BDS), in the **SAP Web Repository**, and in the **MIME Repository**.

The BDS is stored in *Office → Business Documents → Find*. The class CL\_BDS\_DOCUMENT\_SETS contains appropriate methods. For further information, refer to the keyword documentation for the CLASS statement. (*Basis → Basis Services/Communications Interface → SAP Archive Link → SAP Archive Link → Information for All → Business Document Navigator*). The SAP Web Repository is stored in: *Tools → Web Development → Web Repository*. For further information, refer to the online documentation ((*Basis Frontend Services → ITS/SAP@WebStudio → IAC Programming → WebRFC Programming → WebRFC Application Development → Web-Object Maintenance*)).

External data sources could be files on your PC (presentation server), shared directories accessible from the PC, or files on an HTTP or FTP server.

Since WAS 6.10 the MIME Repository stores all MIME objects (style sheets, graphics, icons and so on) in the SAP System. MIMEs are created as objects in the SAP database and can be referred to as part of different application technologies, such as BSP applications and ABAP Web Dynpro.

## The Structure of URLs for External Data Sources



### General Structure:

service://host.domain[:port]/pathname/filename



### Examples:

file:///c:/public/pictures/LogonSequence.gif

file:///c:/public/htmlpages/home.html



http://www.sap.com/images/Enjoy\_new.gif

http://www.sap.com/index.html



ftp://sapserv3/usr/picture.gif

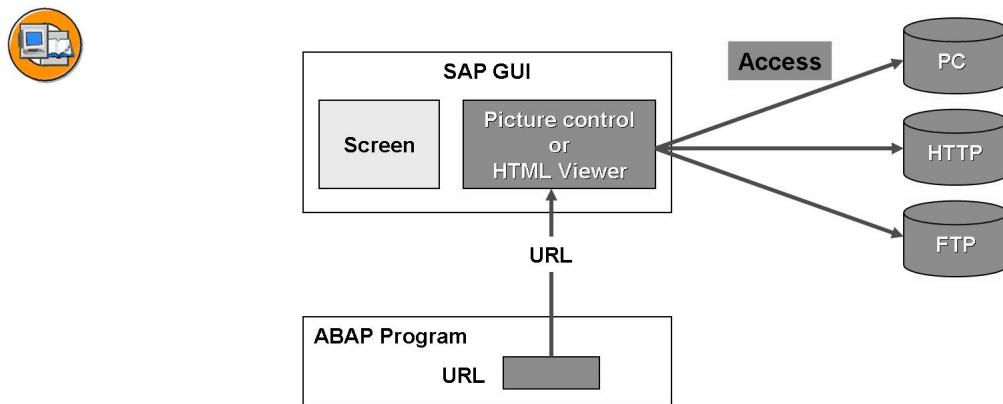
ftp://sapserv3/usr/home.html

Figure 21: The Structure of URLs for External Data Sources

URLs for external data sources are structured as follows:

- Name of the service (file for local PC files, http for files on an HTTP server, and ftp for files on an FTP server)
- :// followed by the host name.domain name and the port number (if the port number differs from the standard port number).
- /directory path/filename

## Accessing External Data by URL



**Figure 22: Accessing External Data by URL**

To display external data in an SAP Picture Control or SAP HTML Viewer Control, you must pass the URL of the data source to the presentation server control (using a method call on the proxy object).

The data source is accessed by the presentation server control itself.



## Accessing Internal Data by URL

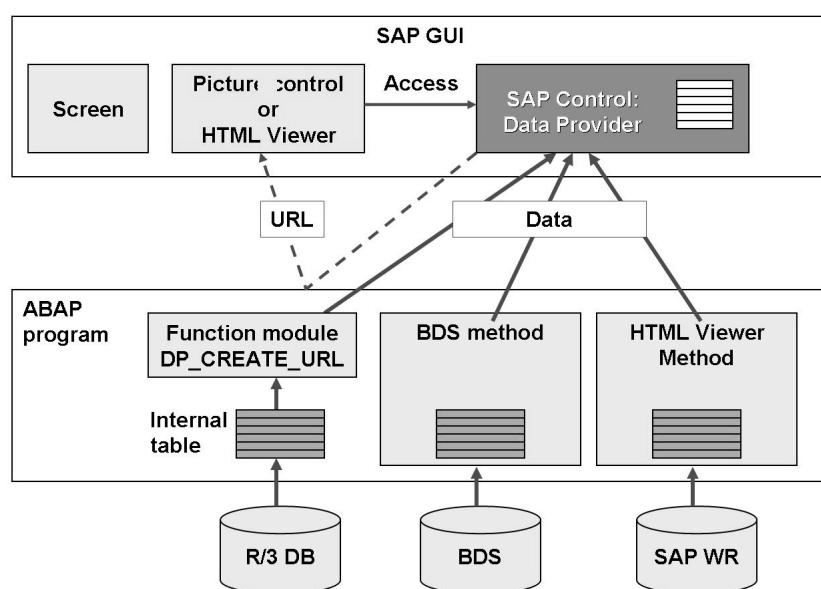


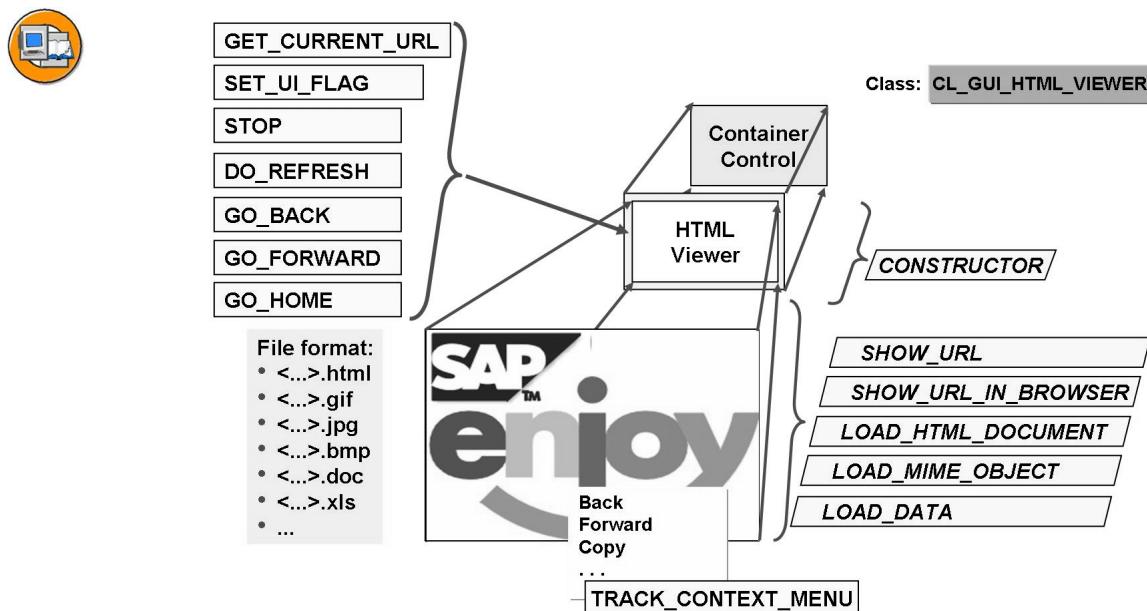
Figure 23: Accessing Internal Data by URL

To access internal data sources in R/3, you also pass a URL to the control. At runtime, this URL points to a data source in the Data Provider. The SAP Data Provider is a control in the SAPGUI. You can pass data to it that is then administered locally by the control instance. When data is sent to the Data Provider, it returns a URL pointing to this data. The SAP Picture Control and the SAP HTML Viewer Control can use this URL to access the data. The Data Provider behaves like an external data source when interacting with these controls.

- Data stored in the SAP Web Repository can be loaded into the Data Provider using instance method `LOAD_HTML_DOCUMENT` of the class `cl_gui_html_viewer`. The Data Provider then returns a URL pointing to the data. The method returns a URL to the data in the Data Provider.
- Data stored in the BDS can be loaded into the Data Provider using the instance method `GET_WITH_URL` on an instance of the class `CL_BDS_DOCUMENT_SET`. The method returns a URL to the data in the Data Provider.
- Data from a cluster table in the R/3 database must be read into an internal table. You can then send them to the Data Provider using the function modules `DP_CREATE_URL_FROM_ITAB` or `DP_CREATE_URL`. The function module returns a URL to the data in the Data Provider.

Data transfer from the internal R/3 data sources is implemented in different ways:

## The SAP HTML Viewer: Features



**Figure 24: The SAP HTML Viewer: Features**

The SAP HTML Viewer is a control developed by SAP for use within the SAP GUI. SAP does not have its own Web browser. Instead, the SAP HTML Viewer for Windows 95 and NT 4.0 uses the Microsoft Internet Explorer Web browser Control. It supports MS Internet Explorer from Version 4.0. Java Beans are available for all other platforms.

The interface of the SAP HTML Viewer is the same for all platforms, but the available functions differ depending on the HTML browser you are using.

You can display HTML pages, graphics, or pictures in your programs using the SAP HTML Viewer. If the SAP HTML Viewer is supported by your HTML browser, you can also use it as a generic container for Java applets and ActiveX controls, or as a generic viewer for any kind of document (for example, MIME contents).

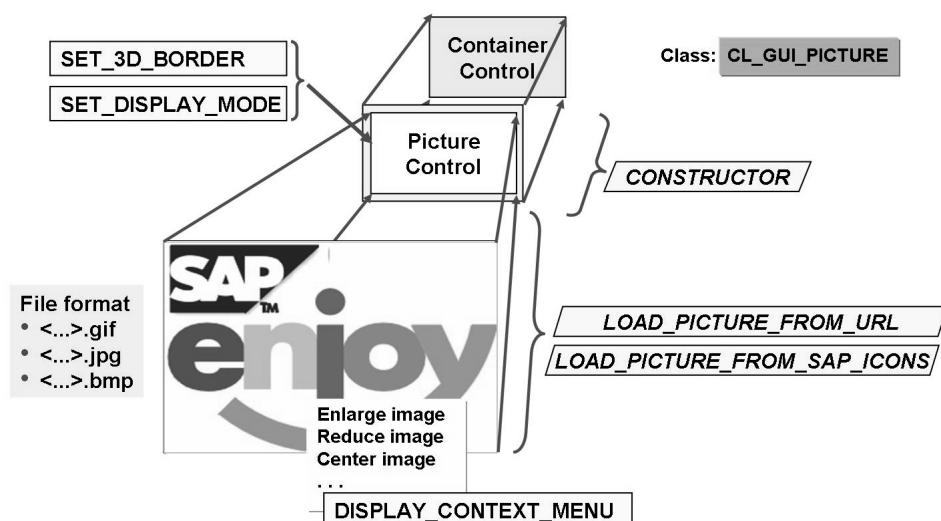
The following applies to Windows platforms: You can only use the SAP HTML Viewer if you have installed Microsoft Internet Explorer.

On all other platforms, the Java Bean is installed with the SAP GUI.

For further information, refer to the online documentation.



## The SAP Picture Control: Features



**Figure 25: The SAP Picture Control: Features**

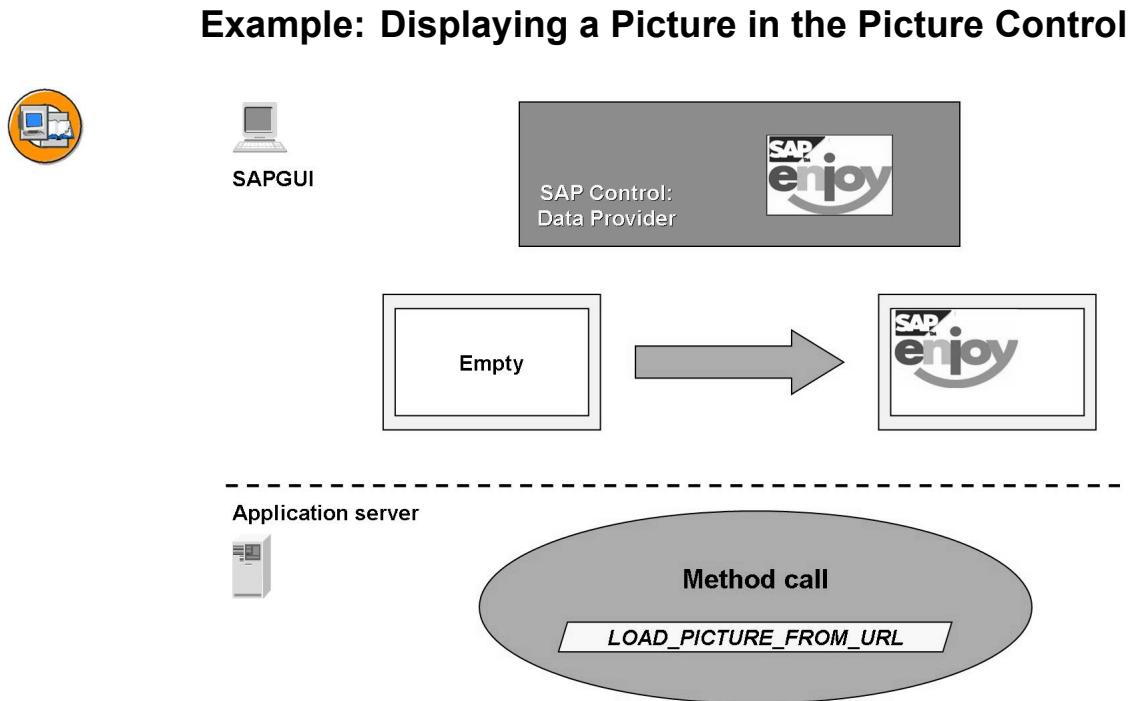
You can use the MIME types GIF (Graphic Interchange Format), JPG (Joint Photographic Experts Group) and BMP (Bitmap) with the SAP Picture Control as well as SAP icons. The corresponding class is called `CL_GUI_PICTURE`.

You can display pictures with the above MIME types using the method `LOAD_PICTURE_FROM_URL`. To display an SAP icon, use the method `LOAD_PICTURE_FROM_SAP_ICONS`. To stop the current data being displayed, use the `CLEAR_PICTURE` method.

The methods `SET_3D_BORDER` and `SET_DISPLAY_MODE` allow you to change the display attributes of the control.

You can also display context menus or perform Drag&Drop operations.

For further details on methods, refer to the online documentation for the SAP Picture Control.



**Figure 26: Example: Displaying a Picture in the Picture Control**

The system will not display a picture on the screen, even after you have created a container containing a SAP Picture Control on the screen, and have loaded a picture into the Data Provider.

To display the picture, you need to change the attributes of the SAP Picture Control – that is, call a method that passes the URL (returned by the Data Provider) for the picture that is to be displayed. OR that passes the URL pointing to the picture you want to display (returned by the Data Provider).

The method is called in your ABAP program. That is, you call a method belonging to the proxy object. This method call is then passed from the CFW to the Automation Controller, which then triggers the associated method of the GUI object.

## Displaying Data: Process Flow

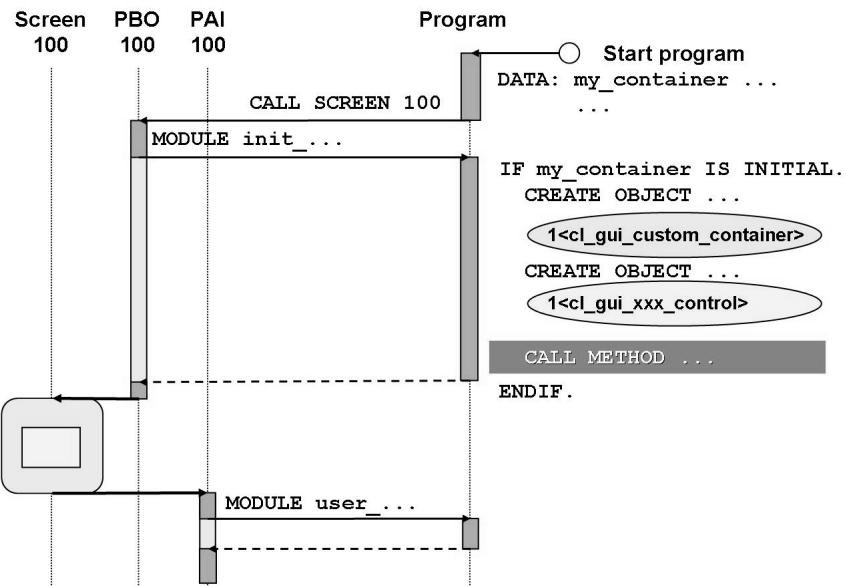


Figure 27: Displaying Data: Process Flow

Since you are calling an instance method of the proxy object, you must instantiate this object **first**.

## Example: Displaying a Picture in the Picture Control

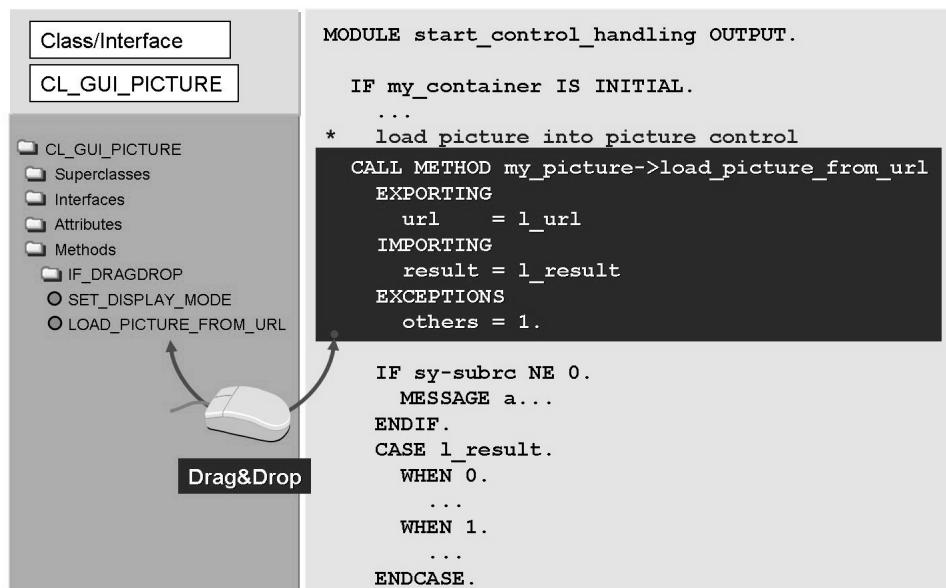


Figure 28: Example: Displaying a Picture in the Picture Control

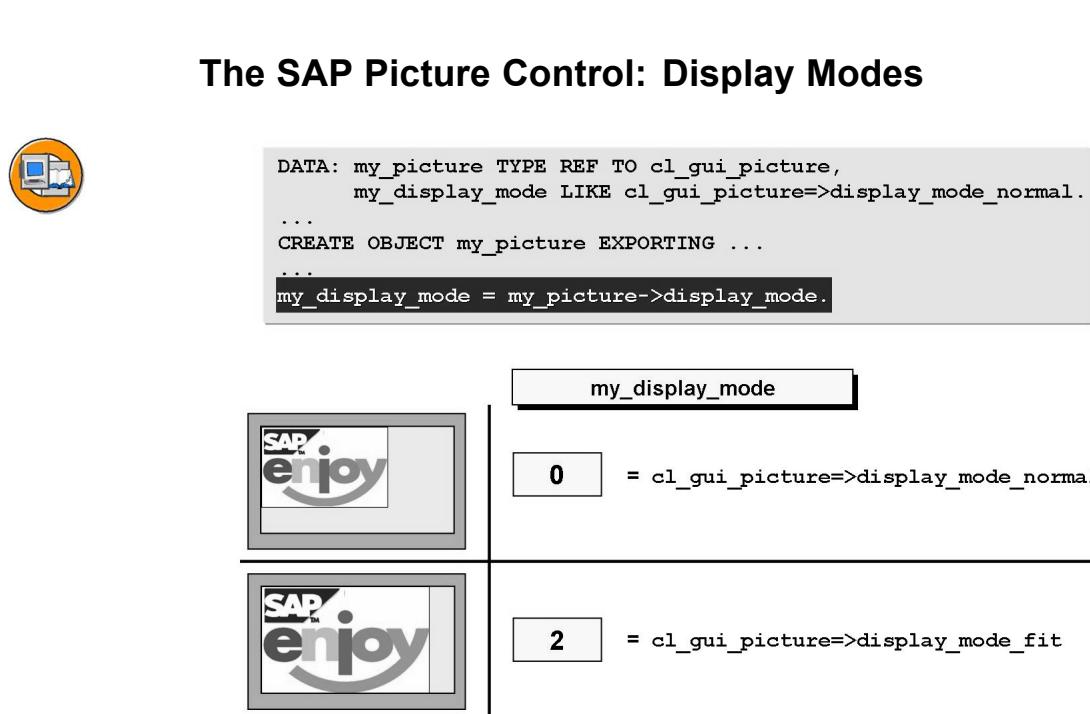
To display a picture in a SAP Picture Control, use the instance method LOAD\_PICTURE\_FROM\_URL. In the interface of the method, you pass the URL of the data source of the picture to the control. You can find out whether the method was successful or not from the return parameter RESULT and in the form of EXCEPTIONS.

If the method triggers an exception, you should react with a suitable termination message. The return parameter RESULT is useful for status messages concerning the load operation (For more details, refer to the online documentation).

For the SAP HTML Viewer Control, you must use one or more (usually two) methods of the control to display data in the browser. This depends on the actual data source. For external data sources: SHOW\_URL, SHOW\_URL\_IN\_BROWSER. For the SAP Web Repository: LOAD\_HTML\_DOCUMENT LOAD\_MIME\_OBJECT. For the SAP Data Provider: LOAD\_DATA, SHOW DATA.

For further information about the SAP HTML Viewer, refer to the online documentation.

To insert a method call in your source code, you can use the Drag&Drop functions in the Object Navigator. In the navigation area, navigate to the class you want and select the method (by clicking once). Then hold down the left mouse button and drag the mouse cursor to where you want to insert the method in the source code. Release the mouse button. The method call is then inserted in the source code at that point – all you need to do now is replace the place-holders with appropriate variable names.



**Figure 29: The SAP Picture Control: Display Modes**

The display mode of a picture in the SAP Picture Control is specified by the instance attribute DISPLAY\_MODE.

Attributes are internal data objects within a class.

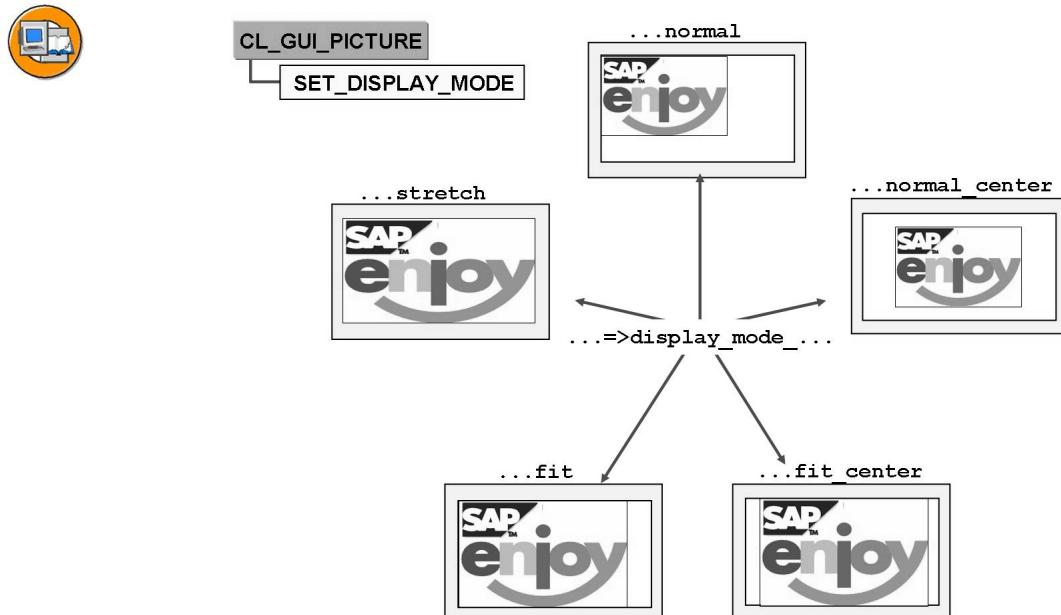
The contents of **instance attributes** determine the state of a particular object.

Instance attributes are created when the constructor is called. From outside the class, you can only address them using object references.

When you access them from outside the class, you must use the object component selector → as follows: <reference> → <attribute>. You can assign the result to a program variable.

In most cases, however, there are SET and GET methods available for reading and changing instance attributes.

## Setting the Picture Display Mode in the SAP Picture Control



**Figure 30: Setting the Picture Display Mode in the SAP Picture Control**

The SET\_DISPLAY\_MODE method allows you to determine how the SAP Picture Control displays the picture. The current display mode is always stored in the public instance attribute DISPLAY\_MODE.

There are five different display modes, corresponding to the five technical values that DISPLAY\_MODE can take. For each of these values there is a static constant with a relevant name:

- cl\_gui\_picture=>display\_mode\_normal: The picture is displayed in its original size. If the picture is larger than the area occupied by the control, the system provides scrollbars. If it is smaller, unused space remains blank.
- cl\_gui\_picture=>display\_mode\_normal\_center: The picture is displayed in its original size but centered within the control area.
- cl\_gui\_picture=>display\_mode\_stretch: The picture is always displayed so that it fills the entire control area. The original picture may appear stretched.
- cl\_gui\_picture=>display\_mode\_fit: The picture loaded into the control is displayed so that it fills the control area, but the length/width ratio is always maintained (similar to stretch but without distorting the image).
- cl\_gui\_picture=>display\_mode\_fit\_center: Similar to fit, but centered within the control area.



## Destroying Control Instances

```
MODULE exit_commands_0100 INPUT.
CASE ok_code.
WHEN 'EXIT'.                      " leave program
    PERFORM free_control_ressources.
    LEAVE PROGRAM.
WHEN 'CANCEL'.                     " cancel screen processing
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
...
ENDMODULE.

FORM free_control_ressources.
    CALL METHOD my_control->free.
    CALL METHOD my_container->free.
    FREE: my_control, my_container.
ENDFORM.
```

**Figure 31: Destroying Control Instances**

To release the resources occupied by a presentation server control, call the instance method FREE of the control you have finished with.

All control wrappers inherit this method from the class CL\_GUI\_OBJECT.

Call this method whenever you have presentation server controls that are no longer needed while your program is running.

By releasing the object references (CLEAR, FREE), you ensure that the resources allocated to the proxy objects in your program are released the next time the Garbage Collector is executed.

## Exercise 2: Changing the Attributes of a Control

### Exercise Objectives

After completing this exercise, you will be able to:

- Change attributes of controls by calling methods
- Display data in the picture control

### Business Example

You will extend the program you wrote in the first exercise so that it displays a picture in your picture control instance.

Use the Business Document Server (BDS) as your data source.

**Program:** ZBC412\_##\_CFW\_EX2

**Template:** SAPBC412\_CFWS\_EXERCISE1

**Model solution:** SAPBC412\_CFWS\_EXERCISE2

where ## is the group number.

### Task 1:

Copy the template.

1. Copy the template *ZBC412\_##\_CFW\_EX1* or the appropriate model solution *SAPBC412\_CFWS\_EXERCISE1* (including all its sub-objects) to the program **ZBC412\_##\_CFW\_EX2** and get to know the features of this copy.

### Task 2:

To display a picture in a picture control instance, you need a URL pointing to the picture. Data objects used in this course that can be displayed in a picture control are stored in the BDS. Use them in your program. Use the function module *BC412\_BDS\_GET\_PIC\_URL* to create a valid URL for a picture in the BDS.

1. Create an ABAP field for the URL. Find out about the interface of the function module you will be using, *BC412\_BDS\_GET\_PIC\_URL*. The export parameter *url* has the ABAP Dictionary type *BAPIURI-URI*. Assign the same type to your field (we suggest the name *l\_url*).
2. Call the function module *BC412\_BDS\_GET\_PIC\_URL*. This function module encapsulates technical details for accessing the picture in the BDS. Fill the *IMPORTING* parameter *NUMBER* with a 1 (default value). The *EXPORT* parameter *URL* returns a valid URL. This URL is valid for the

*Continued on next page*

duration of the program. To make your program clearer, call the function module at the START-OF-SELECTION event, before you call screen 100. If the function module cannot return a valid URL, have the program terminate, using the LEAVE PROGRAM statement.

3. Now you have a URL pointing to a picture. Get the program to display this picture in your control instance. To do this, use the instance method `load_picture_from_url` in the class `cl_gui_picture`. Catch the generic exception, OTHERS, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing. Use **the message 012** from the **message class bc412**.

## Solution 2: Changing the Attributes of a Control

### Task 1:

Copy the template.

1. Copy the template *ZBC412\_##\_CFW\_EX1* or the appropriate model solution *SAPBC412\_CFWS\_EXERCISE1* (including all its sub-objects) to the program **ZBC412\_##\_CFW\_EX2** and get to know the features of this copy.
  - a) —

### Task 2:

To display a picture in a picture control instance, you need a URL pointing to the picture. Data objects used in this course that can be displayed in a picture control are stored in the BDS. Use them in your program. Use the function module *BC412\_BDS\_GET\_PIC\_URL* to create a valid URL for a picture in the BDS.

1. Create an ABAP field for the URL. Find out about the interface of the function module you will be using, *BC412\_BDS\_GET\_PIC\_URL*. The export parameter *url* has the ABAP Dictionary type *BAPIURI-URI*. Assign the same type to your field (we suggest the name *l\_url*).

a)

```
TYPES:      t_url          TYPE bapiuri-uri.
DATA:
...
l_url      TYPE t_url.           URL of picture to be shown
```

2. Call the function module *BC412\_BDS\_GET\_PIC\_URL*. This function module encapsulates technical details for accessing the picture in the BDS. Fill the IMPORTING parameter *NUMBER* with a 1 (default value). The EXPORT parameter *URL* returns a valid URL. This URL is valid for the duration of the program. To make your program clearer, call the function

*Continued on next page*

module at the START-OF-SELECTION event, before you call screen 100. If the function module cannot return a valid URL, have the program terminate, using the LEAVE PROGRAM statement.

a)

```
CALL FUNCTION 'BC412_BDS_GET_PIC_URL' fetch URL of first picture
*      EXPORTING                                     " from BDS
*          NUMBER          = 1
IMPORTING
    url          = l_url
EXCEPTIONS
    OTHERS        = 1.

IF sy-subrc <> 0.           " no picture --> end of program
    LEAVE PROGRAM.
ENDIF.
```

3. Now you have a URL pointing to a picture. Get the program to display this picture in your control instance. To do this, use the instance method `load_picture_from_url` in the class `cl_gui_picture`. Catch the generic exception, OTHERS, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing. Use the message 012 from the message class `bc412`.

a)

```
*      load picture into picture control
CALL METHOD ref_picture->load_picture_from_url
EXPORTING
    url      = l_url
EXCEPTIONS
    OTHERS   = 1.

IF sy-subrc NE 0.
    MESSAGE a012.
ENDIF.
```

### ABAP program

#### *Data Declarations*

```
*-----*
*& Report  SAPBC412_CFW_S_EXERCISE2
*&
*-----*
```

*Continued on next page*

```

REPORT  sapbc412_cfw_s_exercise2 MESSAGE-ID bc412.
* data types
TYPES:
  t_url      TYPE bapiuri-uri.
* data declarations
* screen specific
DATA:
  ok_code      TYPE sy-ucomm,          " command field
  copy_ok_code LIKE ok_code,          " copy of ok_code
  l_answer     TYPE c,                " return flag (used in
                                         " standard user dialogs)

* control specific: object references
  ref_container TYPE REF TO cl_gui_custom_container,
  ref_picture    TYPE REF TO cl_gui_picture,

* control specific: auxiliary fields
  l_url         TYPE t_url.          " URL of picture to be shown

```

## Result

### Main ABAP Program: Event Blocks

```

* start of main program
START-OF-SELECTION.

  CALL FUNCTION 'BC412_BDS_GET_PIC_URL' fetch URL of first picture
*   EXPORTING                               " from BDS
*     NUMBER        = 1
  IMPORTING
    url          = l_url
  EXCEPTIONS
    OTHERS        = 1.

  IF sy-subrc <> 0.                      " no picture --> end of program
    LEAVE PROGRAM.
  ENDIF.

  CALL SCREEN 100.                         " container screen for SAP-Enjoy
                                         " controls

* end of main program

```

*Continued on next page*

## Modules

```

*-----*
*&      Module  INIT_CONTROL_PROCESSING  OUTPUT
*-----*
*      Implementation of EnjoySAP control processing:
*      1)  create container object and link to screen area
*      2)  create picture object and link to container
*      3)  load picture into picture control
*-----*
MODULE init_control_processing OUTPUT.

IF ref_container IS INITIAL.      " prevent re-processing on ENTER
*      create container object and link to screen area
CREATE OBJECT ref_container
EXPORTING
  container_name = 'CONTROL_AREA1'
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
MESSAGE a010.                      " cancel program processing
ENDIF.

*      create picture control and link to container object
CREATE OBJECT ref_picture
EXPORTING
  parent = ref_container
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
MESSAGE a011.                      " cancel programn processing
ENDIF.

*      load picture into picture control
CALL METHOD ref_picture->load_picture_from_url
EXPORTING
  url    = l_url
EXCEPTIONS
  OTHERS = 1.

IF sy-subrc NE 0.
MESSAGE a012.

```

*Continued on next page*

```
ENDIF.  
  
ENDIF.  
ENDMODULE.                                     " INIT_CONTROL_PROCESSING  OUTPUT
```



## Lesson Summary

You should now be able to:

- Understand and use the attributes of a control

# Lesson: Automation Queue

## Lesson Overview

This lesson presents how the Automation Queue works and how you could optimize the runtime performance.



## Lesson Objectives

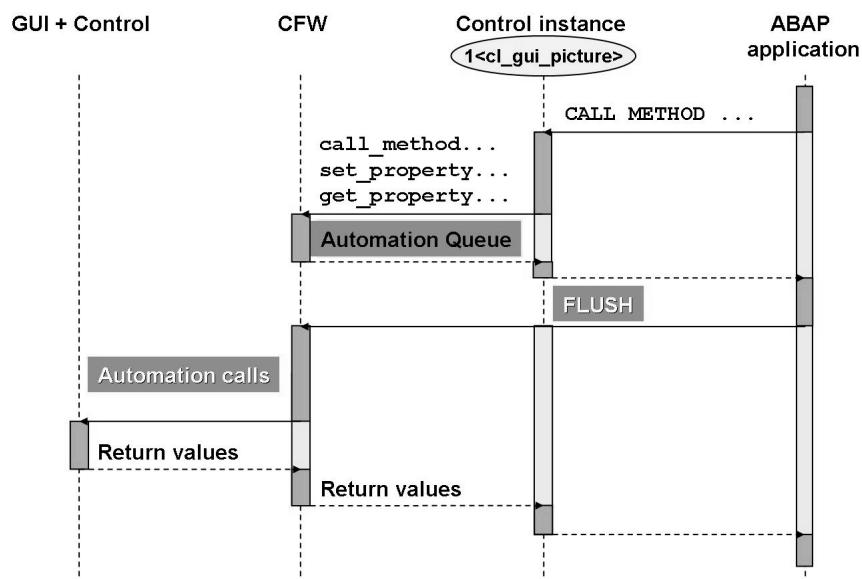
After completing this lesson, you will be able to:

- Understand how the actions are transferred to the presentation server
- Describe the Automation Queue and its Runtime Performance

## Business Example

The project team of which you are a member has decided to implement an application using EnjoySAP Controls Picture and HTML Viewer. You want to understand how the Automation Queue works so you can optimize the runtime performance

## Method Calls and the Automation Queue



**Figure 32: Method Calls and the Automation Queue**

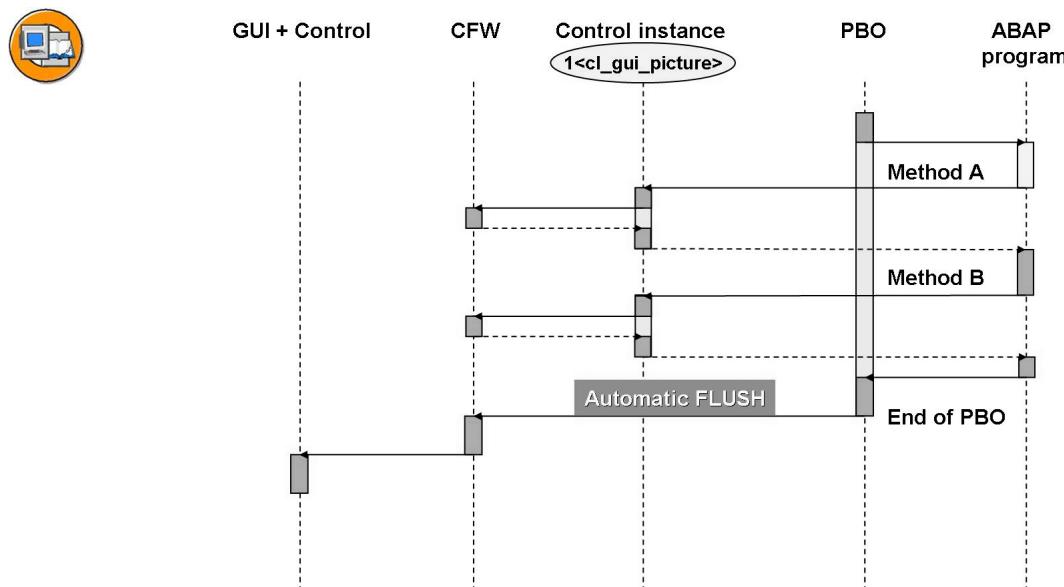
Usually, when you call a method or read attributes within a proxy instance, these activities are buffered in the CFW in the Automation Queue. This helps reduce demands on the system at runtime.

The actions are not transferred to the presentation server control until a particular Control Framework method called **FLUSH** is called.

In the Control Framework, there is a single automation queue for all controls. When the flush occurs, all of the actions stored for all presentation server controls are transferred.

**Result values** are also transferred from the presentation server controls back to the Control Framework only when the flush takes place.

## Passing the Automation Queue Automatically

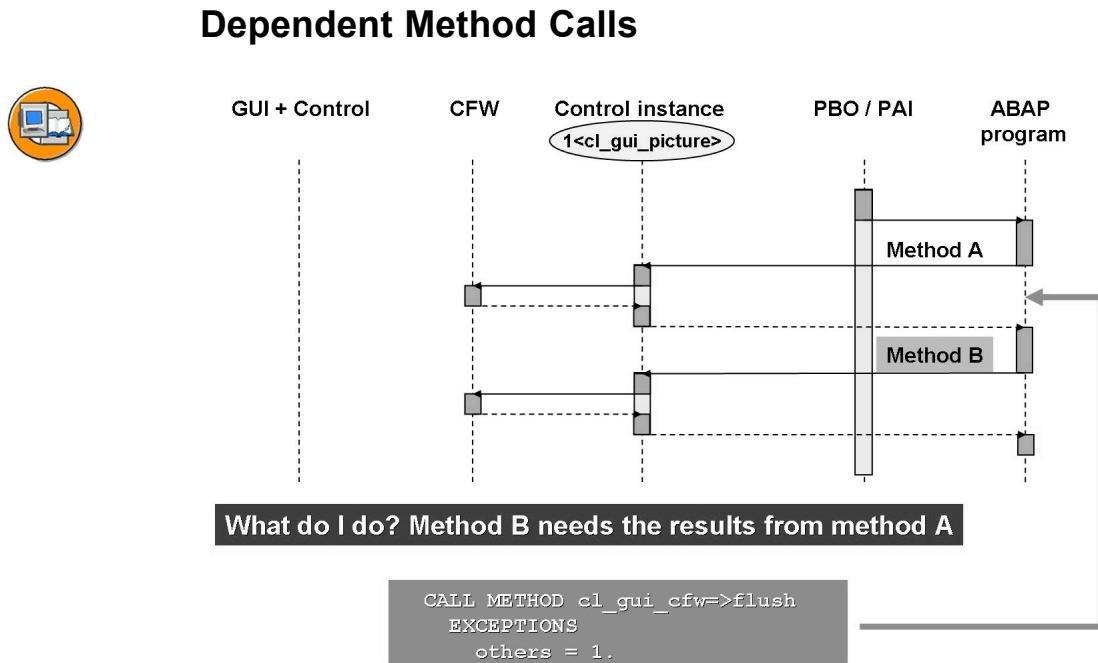


**Figure 33: Passing the Automation Queue Automatically**

**At the end of the PBO event** for a given screen, the **runtime system** **FLUSHes** the Automation Queue.

In compound screens, the FLUSH is automatically triggered **once** only at the end of the “top-level” PBO event, immediately before the SAP GUI information is sent to the Presentation Server.

If a screen has one or more subscreens, the FLUSH takes place only when the PBO of the main screen containing these subscreens has been processed. The same applies to hierarchically-nested screens (a screen containing a subscreen that itself includes a subscreen).



**Figure 34: Dependent Method Calls**

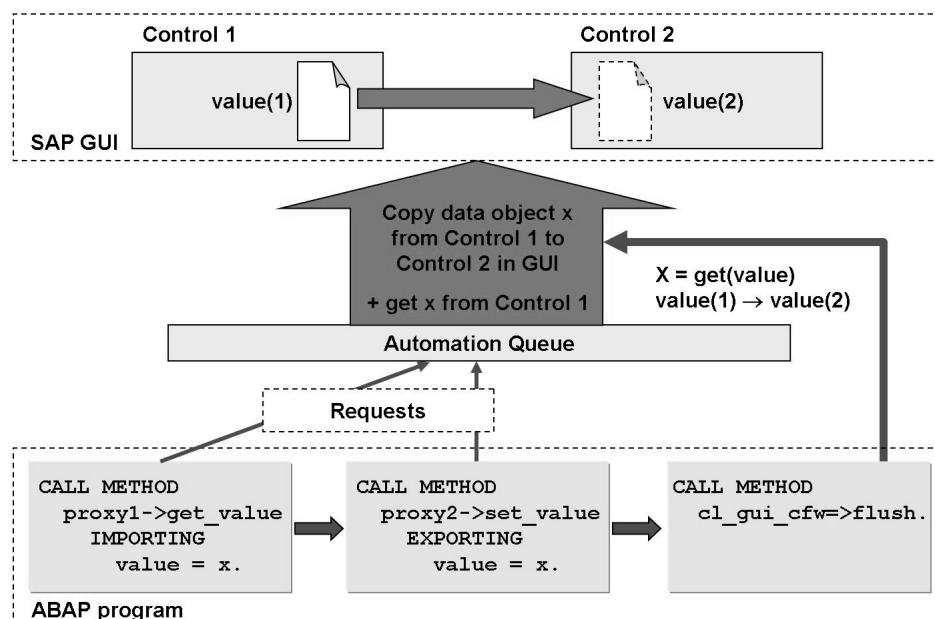
If you call two instance methods one after the other, and the second method needs results provided by the first, you must ensure that the Automation Queue is transferred to the presentation server after the first method call.

You can do this by calling the static method FLUSH of the class CL\_GUI\_CFW. This method passes all the data collected in the Automation Queue (method A in our example above) at this point in time to the presentation server control.

**Note however:**

- You do not always need to use explicit flushes, because of the way the Automation Queue is handled in the CFW: If queries for the values of control attributes can be answered using data already in the Automation Queue, then there is no need for any communication with the presentation server. **Example:** Suppose method A sets attribute a1 of object O to value v1. If method B of object O is executed and queries the value of a1, the Control Framework will return the value v1 **from the Automation Queue**.
- The system load is increased very time the Automation Queue is passed. This means that you should call the FLUSH method explicitly only when absolutely essential.

## Optimizing Runtime Performance in the Automation Queue



**Figure 35: Optimizing Runtime Performance in the Automation Queue**

As outlined previously, the Automation Queue is implemented so as to optimize runtime performance. This concept is explained in more detail below.

To move or copy data between controls, you must call two methods: The first method queries the first control for the data to be moved; the second moves this data to the second control. The implementation of the Automation Queue optimizes this process by making the copying process take place on the presentation server, **without** involving the application server – that is, avoiding unnecessary round trips between the SAP GUI and the application server (from the system's point of view). You should always use this method to exchange large volumes of data between controls.

The system transports the data queried from control 1 to the declared variable **at the end of executing the method cl\_gui\_cfw=>flush**.

If the automation queue is empty, the system does not perform a round trip between the application server and the SAP GUI.

## Summary: Recommendations for Calling Methods at PBO

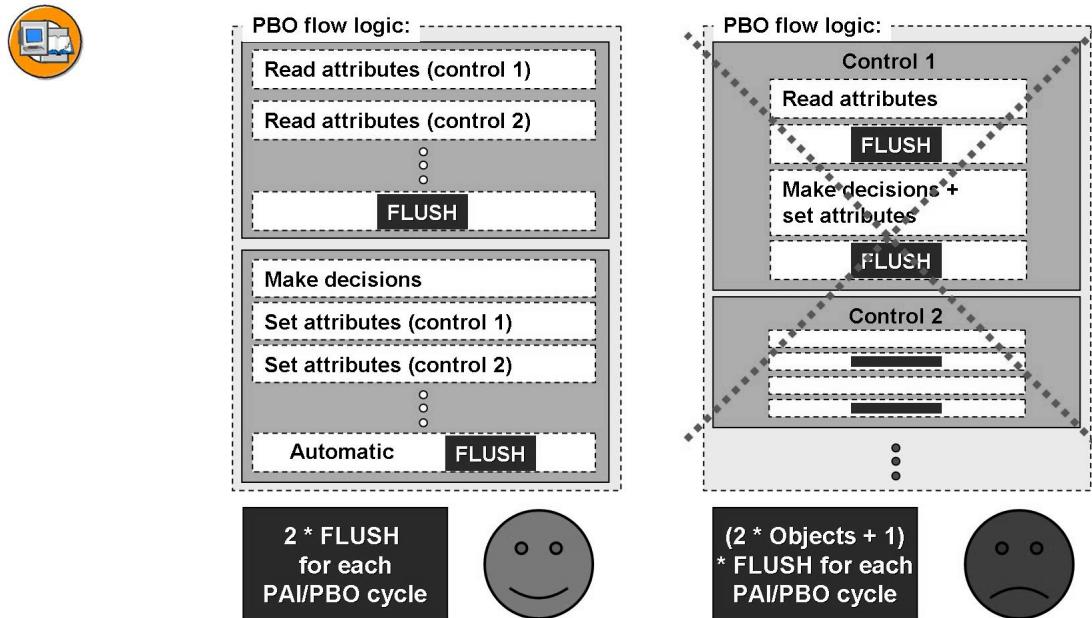


Figure 36: Summary: Recommendations for Calling Methods at PBO

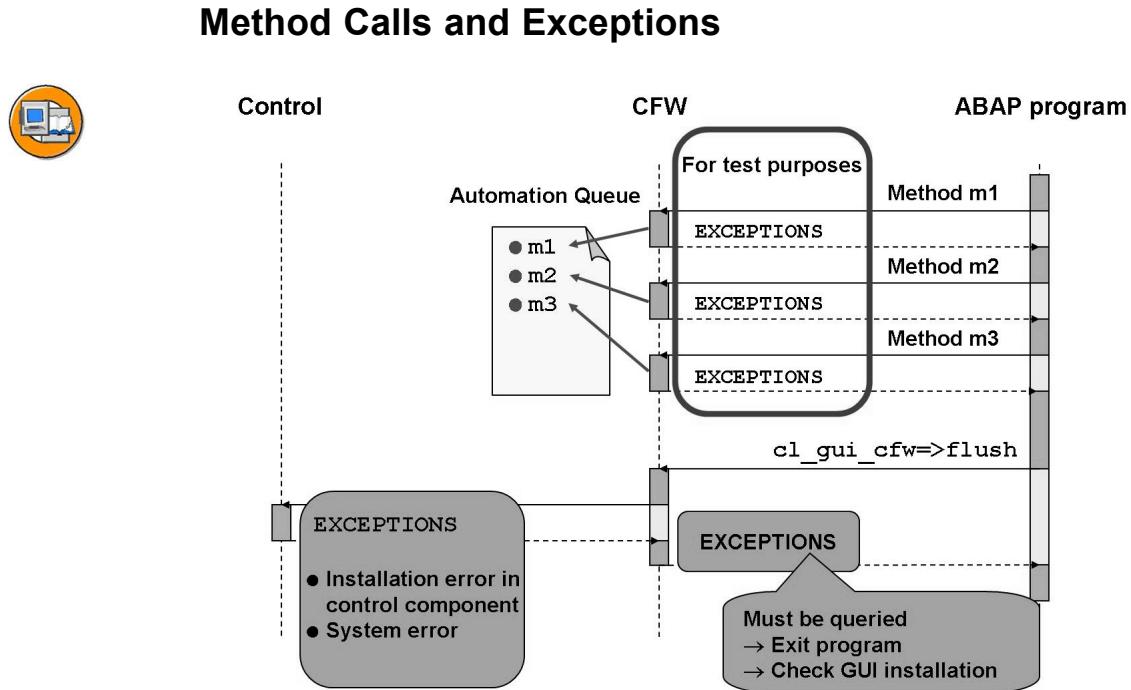
To improve performance in your programs, you should always try to minimize the number of round trips between the SAP GUI and the application server.

Example: A user is connected to an SAP R/3 System by satellite. Typically, it will take 0.5 seconds to set up a connection between the SAP GUI and the server. If your screen requires 10 round trips, the user will have to wait **10 seconds** before he or she can perform an action on the screen.

To minimize the number of round trips, you should structure your PBO modules in such a way that you call all methods only **after** obtaining the attributes of all the controls you are using. If other actions depend on the results of these reading operations, you must set a synchronization point (`cl_gui_cfw=>flush`). You can then call all the methods that change attributes, depending on the results.

The PBO module structure on the left is much better for runtime performance than the one on the right, where the method `cl_gui_cfw=>flush` is called twice for every control.

Note: When you use automatic synchronization points, make sure that the contents of the automation queue are transferred **after** the rest of the screen data. If all the screen fields that are filled with return parameters from method interfaces point to current values, you need another **FLUSH**.



**Figure 37: Method Calls and Exceptions**

Exceptions in the interfaces of proxy methods generally refer to errors in the presentation server control. Since method calls are buffered in the automation queue, the triggering of these exceptions is deferred. You should still implement exceptions, so that you can perform meaningful tests on the program in the Debugger. (The Debugger allows you to execute all methods synchronously. More details are available later in this course).

If an error occurs at the front end while the automation queue is being processed (at a synchronization point), the system aborts processing of the entire queue and triggers an exception. You should **always** implement exceptions in the interface of the method `cl_gui_cfw=>flush`.

Errors processing the automation queue on the presentation server generally occur when the control has been incorrectly installed. If such an error occurs, you should make sure that processing is terminated cleanly (with a termination ("A") or exit ("X") message), and that the presentation server installation is tested and repaired by an administrator.

## Testing the Application Server: ABAP Debugger

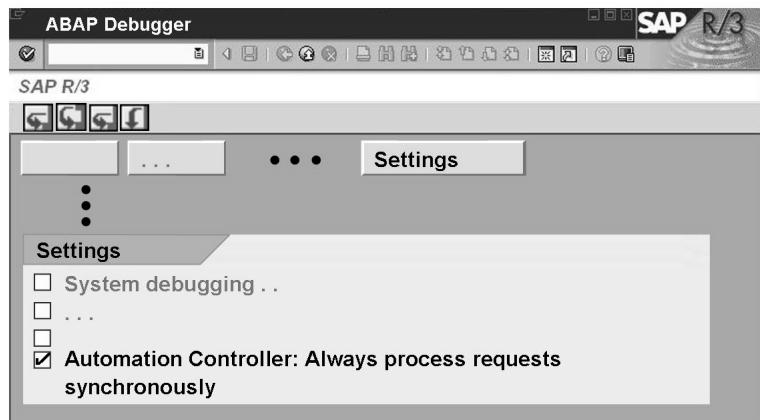


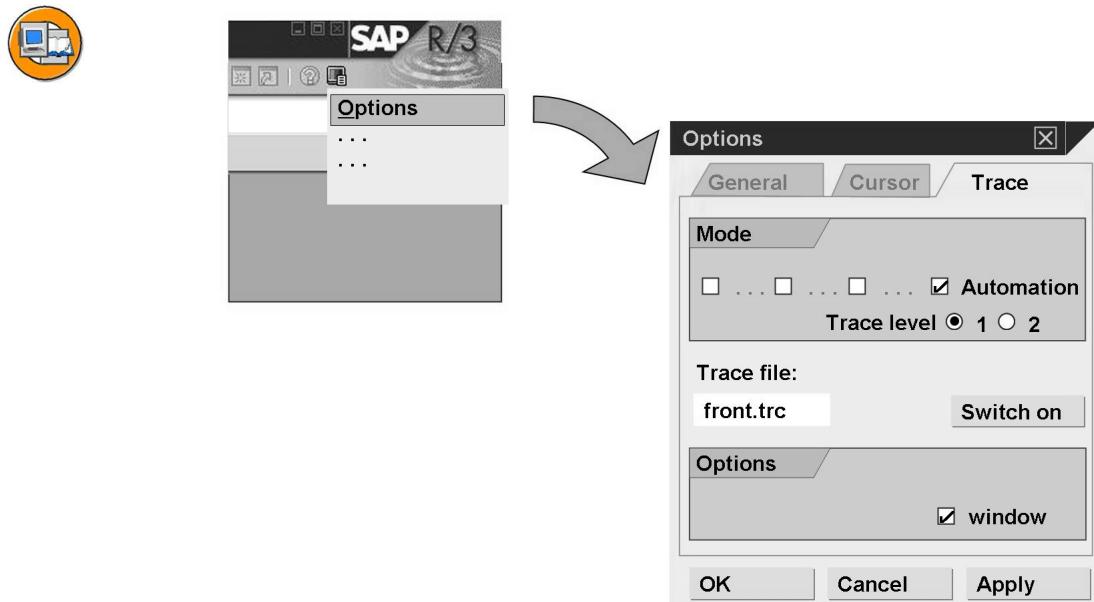
Figure 38: Testing the Application Server: ABAP Debugger

When you use buffered operations to communicate with controls, errors in a method call are not visible **until you synchronize the Automation Queue**. It is therefore a good idea to synchronize the automation queue after each method call when you are debugging.

You can do this by selecting Automation Controller: Always process requests synchronously in the “Classic” Debugger. This calls the static method flush from the class cl\_gui\_cfw, after each automation method.

If the error no longer occurs when you run the program in this way, you have called the method cl\_gui\_cfw=>flush at the wrong point somewhere in your program. You should include appropriate error handling after every method call (query value of sy-subrc). Note, however, that the error handling is normally only processed within Debugging.

## Testing the Presentation Server: Automation Trace



**Figure 39: Testing the Presentation Server: Automation Trace**

You can set up a trace for the Automation Queue. To do this, choose Automation in the Trace group of the SAP GUI settings. Subsequently, all calls with their Automation Queue parameters are logged in a trace file.

If an error occurs, it is also logged in the trace file (HRESULT error\_code).

You can also check the number of synchronizations (times the automation queue is flushed) that your application uses in each PAI/PBO block. You can then eliminate any redundant synchronizations.

When you follow the method calls in the trace file that the ABAP method names, these names generally **differ** from the method names in the trace. The method names in the trace are those of the automation calls to the control.

A single ABAP method call may cause **more than one** automation method call.

# Exercise 3: Changing the Attributes of a Control Dynamically

## Exercise Objectives

After completing this exercise, you will be able to:

- Destroy control instances on the presentation server
- Release reference variables to proxy objects
- Change attributes of a control by calling methods (controlled by pushbuttons on a screen)
- Ascertain the attributes of a control (read access)
- You let users show the current display mode for the picture control

## Business Example

Building on the previous exercise, make sure that all presentation server instances are always destroyed before the program ends. In addition, make sure that the user can change the picture display mode by choosing pushbuttons on the screen.

**Program:** ZBC412\_##\_CFW\_EX3

**Template:** SAPBC412\_CFWS\_EXERCISE2

**Model solution:** SAPBC412\_CFWS\_EXERCISE3

where ## is the group number.

## Task 1:

Copy the template.

1. Copy your solution from the previous exercise (*ZBC412\_##\_CFW\_EX2*) or the appropriate model solution (*SAPBC412\_CFWS\_EXERCISE2*) with all their sub-objects to the name **ZBC412\_##\_CFW\_EX3**. Get to know how your copy of the program works.
2. Make sure that your control instances at the presentation server are destroyed before the program ends. To do this, implement the subroutine `free_control_ressources`. In addition, call the instance method `free` and release the ABAP reference variables.



**Hint:** The subroutine definition and the statement to call it are already part of your program.

*Continued on next page*

## Task 2:

Make sure that users can specify how the picture control instance is to be displayed, using buttons on screen 100.

1. Implement the functions “STRETCH”, “NORMAL”, “NORMAL\_CENTER”, “FIT”, and “FIT\_CENTER”, defined by the pushbuttons. These functions are already implemented in the PAI module `user_command_0100`. To do this, use the instance method `set_display_mode` in the class `cl_gui_picture`.

To fill the interface parameter `display_mode` of the `set_display_mode` method, use the class constants:

```
display_mode_stretch,  
display_mode_normal,  
display_mode_normal_center,  
display_mode_fit and display_mode_fit_center  
of the class cl_gui_picture. Access them using  
cl_gui_picture=>display_mode....
```

If an exception occurs while the program is running, make the system display the message 015 from the message class BC412 (message type S or I).

2. Test the mode in which the control displays the picture by resizing the window on the subscreen container.

## Task 3: Optional

**Program:** ZBC412\_##\_CFW\_EX3

**Template:** SAPBC412\_CFWS\_EXERCISE3

**Model solution:** SAPBC412\_CFWS\_EXERCISE3A

where ## is the group number.

Extend container screen 100 by adding a button, which – when clicked – displays the current display mode for the picture control, as a technical value in an information message.

1. On screen 100, create a pushbutton with the following attributes:

Attribute	Value	Suggested value
Element type	Pushbutton	
Name	Any	BUT6
Text	Any	Query mode

*Continued on next page*

Attribute	Value	Suggested value
Icon	Any	ICON_INFORMATION
Function Code	MODE_INFO	
Function Code Type	'' (blank)	

2. Implement the associated function (function code MODE\_INFO) in the module user\_command\_0100. Read the display\_mode attribute of your picture control proxy instance and display the content using the information message 025 (message class bc412).

## Solution 3: Changing the Attributes of a Control Dynamically

### Task 1:

Copy the template.

1. Copy your solution from the previous exercise (*ZBC412\_##\_CFW\_EX2*) or the appropriate model solution (*SAPBC412\_CFW\_EXERCISE2*) with all their sub-objects to the name **ZBC412\_##\_CFW\_EX3**. Get to know how your copy of the program works.
  - a) –
2. Make sure that your control instances at the presentation server are destroyed before the program ends. To do this, implement the subroutine `free_control_ressources`. In addition, call the instance method `free` and release the ABAP reference variables.



**Hint:** The subroutine definition and the statement to call it are already part of your program.

a)

```
FORM free_control_ressources.
  CALL METHOD ref_picture->free.
  CALL METHOD ref_container->free.
  FREE: ref_picture, ref_container.
ENDFORM.                                     " free_control_ressources
```

### Task 2:

Make sure that users can specify how the picture control instance is to be displayed, using buttons on screen 100.

1. Implement the functions “STRETCH”, “NORMAL”, “NORMAL\_CENTER”, “FIT”, and “FIT\_CENTER”, defined by the pushbuttons. These functions are already implemented in the PAI module `user_command_0100`. To do this, use the instance method `set_display_mode` in the class `cl_gui_picture`.

To fill the interface parameter `display_mode` of the `set_display_mode` method, use the class constants:

```
display_mode_stretch,
display_mode_normal,
```

*Continued on next page*

display\_mode\_normal\_center,  
 display\_mode\_fit and display\_mode\_fit\_center  
 of the class cl\_gui\_picture. Access them using  
 cl\_gui\_picture=>display\_mode....

If an exception occurs while the program is running, make the system display the message 015 from the message class BC412 (message type S or I).

a)

```
CASE copy_ok_code.
...
  WHEN 'STRETCH'.           " picture operation: stretch to fit area
    CALL METHOD ref_picture->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_stretch
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.
  WHEN 'NORMAL'.            " picture operation: fit to normal size
    CALL METHOD ref_picture->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_normal
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'NORMAL_CENTER'.    " picture operation: center normal size
    CALL METHOD ref_picture->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_normal_center
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'FIT'.               " picture operation: zoom picture
    CALL METHOD ref_picture->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_fit
```

*Continued on next page*

```

EXCEPTIONS
  OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'FIT_CENTER'.      " picture operation: zoom and center
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_fit_center
EXCEPTIONS
  OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.
ENDCASE.

```

2. Test the mode in which the control displays the picture by resizing the window on the subscreen container.
  - a) -

### Task 3: Optional

**Program:** ZBC412\_##\_CFW\_EX3

**Template:** SAPBC412\_CFWS\_EXERCISE3

**Model solution:** SAPBC412\_CFWS\_EXERCISE3A

where ## is the group number.

Extend container screen 100 by adding a button, which – when clicked – displays the current display mode for the picture control, as a technical value in an information message.

1. On screen 100, create a pushbutton with the following attributes:

Attribute	Value	Suggested value
Element type	Pushbutton	
Name	Any	BUT6
Text	Any	Query mode

*Continued on next page*

Attribute	Value	Suggested value
Icon	Any	ICON_INFORMATION
Function Code	MODE_INFO	
Function Code Type	'' (blank)	

- a) –
2. Implement the associated function (function code MODE\_INFO) in the module user\_command\_0100. Read the display\_mode attribute of your picture control proxy instance and display the content using the information message 025 (message class bc412).
- a)

```
CASE copy_ok_code.
...
WHEN 'MODE_INFO'.
MESSAGE i025 WITH ref_picture->display_mode.
ENDCASE.
```

## Result

### ABAP program

#### Modules

```
*&-----*
*&     Module  USER_COMMAND_0100  INPUT
*&-----*
*     Implementation of user commands of type ' ':
*     - push buttons on screen 100
*     - GUI functions
*-----*
MODULE user_command_0100 INPUT.
copy_ok_code = ok_code.
CLEAR ok_code.

CASE copy_ok_code.
WHEN 'BACK'.                               " back to program, leave screen
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
        EXPORTING
            DEFAULTOPTION  = 'Y'
            textline1      = text-004
```

*Continued on next page*

```

textline2      = text-005
titel         = text-007
cancel_display = ' '
IMPORTING
answer        = l_answer.

CASE l_answer.
WHEN 'J'.
  PERFORM free_control_ressources.
  LEAVE TO SCREEN 0.
WHEN 'N'.
  SET SCREEN sy-dynnr.
ENDCASE.

WHEN 'STRETCH'.    " picture operation: stretch to fit area
CALL METHOD ref_picture->set_display_mode
EXPORTING
  display_mode = cl_gui_picture->display_mode_stretch
EXCEPTIONS
  OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'NORMAL'.    " picture operation: fit to normal size
CALL METHOD ref_picture->set_display_mode
EXPORTING
  display_mode = cl_gui_picture->display_mode_normal
EXCEPTIONS
  OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'NORMAL_CENTER'.    " picture operation: center normal size
CALL METHOD ref_picture->set_display_mode
EXPORTING
  display_mode = cl_gui_picture->display_mode_normal_center
EXCEPTIONS
  OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'FIT'.    " picture operation: zoom picture
CALL METHOD ref_picture->set_display_mode

```

*Continued on next page*

```

EXPORTING
  display_mode = cl_gui_picture->display_mode_fit
EXCEPTIONS
  OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'FIT_CENTER'.      " picture operation: zoom and center
CALL METHOD ref_picture->set_display_mode
EXPORTING
  display_mode = cl_gui_picture->display_mode_fit_center
EXCEPTIONS
  OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

ENDCASE.

ENDMODULE.      " USER_COMMAND_0100  INPUT

```

### ***Subroutines***

```

*&-----*
*&      Form free_control_ressources
*&-----*
*      Free all control related ressources.
*-----*
*      no interface
*-----*
FORM free_control_ressources.
  CALL METHOD ref_picture->free.
  CALL METHOD ref_container->free.
  FREE: ref_picture, ref_container.
ENDFORM.      " free_control_ressources

```

### **ABAP program (Optional Part)**

#### ***Data Declarations***

```

*&-----*
*& Report SAPBC412_CFWS_EXERCISE3A
*&

```

*Continued on next page*

```
*&-----*
REPORT sapbc412_cfw_exercise3a MESSAGE-ID bc412.
* data types
TYPES:
  t_url      TYPE bapiuri-uri.
* data declarations
* screen specific
DATA:
  ok_code      TYPE sy-ucomm,          " command field
  copy_ok_code LIKE ok_code,          " copy of ok_code
  l_answer     TYPE c,                " return flag (used in
                                         " standard user dialogs)

* control specific: object references
  ref_container TYPE REF TO cl_gui_custom_container,
  ref_picture    TYPE REF TO cl_gui_picture,

* control specific: auxiliary fields
  l_url         TYPE t_url.          " URL of picture to be shown
```

### ***Modules***

```
*&-----*
*&     Module  USER_COMMAND_0100  INPUT
*&-----*
*     Implementation of user commands of type ' ':
*     - push buttons on screen 100
*     - GUI functions
*-----*
MODULE user_command_0100 INPUT.
  copy_ok_code = ok_code.
  CLEAR ok_code.

CASE copy_ok_code.
  WHEN 'BACK'.                         " back to program, leave screen
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *           DEFAULTOPTION  = 'Y'
        textline1    = text-004
        textline2    = text-005
        titel       = text-007
        cancel_display = ' '
```

*Continued on next page*

```

IMPORTING
    answer          = l_answer.

CASE l_answer.
    WHEN 'J'.
        PERFORM free_control_ressources.
        LEAVE TO SCREEN 0.
    WHEN 'N'.
        SET SCREEN sy-dynnr.
    ENDCASE.

WHEN 'STRETCH'.      " picture operation: stretch to fit area
    CALL METHOD ref_picture->set_display_mode
    EXPORTING
        display_mode = cl_gui_picture->display_mode_stretch
    EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
        MESSAGE s015.
    ENDIF.

WHEN 'NORMAL'.       " picture operation: fit normal size
    CALL METHOD ref_picture->set_display_mode
    EXPORTING
        display_mode = cl_gui_picture->display_mode_normal
    EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
        MESSAGE s015.
    ENDIF.

WHEN 'NORMAL_CENTER'. " picture operation: center normal size
    CALL METHOD ref_picture->set_display_mode
    EXPORTING
        display_mode = cl_gui_picture->display_mode_normal_center
    EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
        MESSAGE s015.
    ENDIF.

WHEN 'FIT'.           " picture operation: zoom picture
    CALL METHOD ref_picture->set_display_mode
    EXPORTING
        display_mode = cl_gui_picture->display_mode_fit
    EXCEPTIONS

```

*Continued on next page*

```
OTHERS = 1.  
IF sy-subrc NE 0.  
MESSAGE s015.  
ENDIF.  
  
WHEN 'FIT_CENTER'.      " picture operation: zoom and center  
CALL METHOD ref_picture->set_display_mode  
EXPORTING  
    display_mode = cl_gui_picture=>display_mode_fit_center  
EXCEPTIONS  
    OTHERS = 1.  
IF sy-subrc NE 0.  
MESSAGE s015.  
ENDIF.  
  
WHEN 'MODE_INFO'.  
MESSAGE i025 WITH ref_picture->display_mode.  
  
ENDCASE.  
ENDMODULE.           " USER_COMMAND_0100  INPUT
```



## Lesson Summary

You should now be able to:

- Understand how the actions are transferred to the presentation server
- Describe the Automation Queue and its Runtime Performance

# Lesson: Control Events

## Lesson Overview

This lesson presents the events triggered by the SAP Controls.



## Lesson Objectives

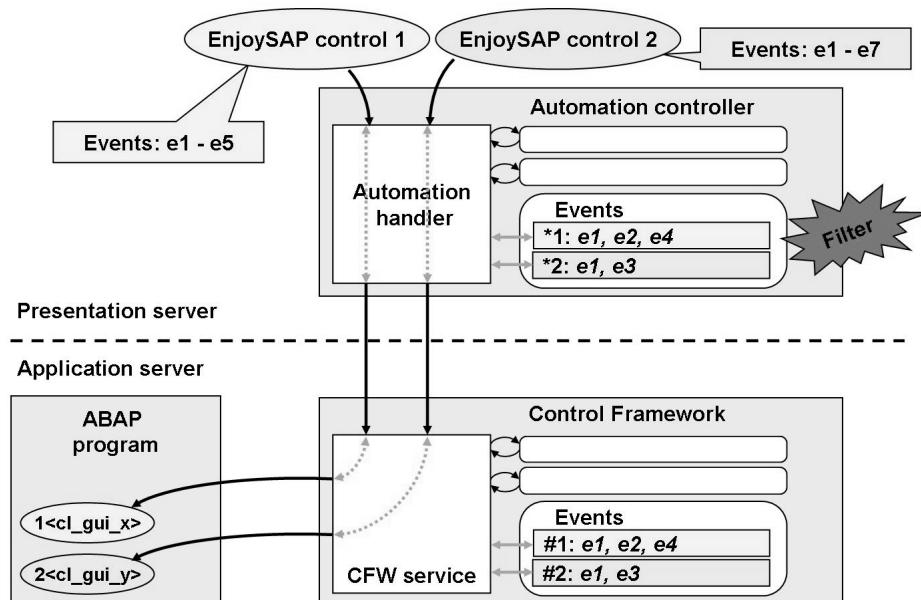
After completing this lesson, you will be able to:

- Describe the events triggered by the EnjoySAP Controls and learn the steps necessary to react to them

## Business Example

The project team of which you are a member has decided to implement an application using EnjoySAP Controls Picture and HTML Viewer. These controls can trigger events that you need to know to be able to handle them.

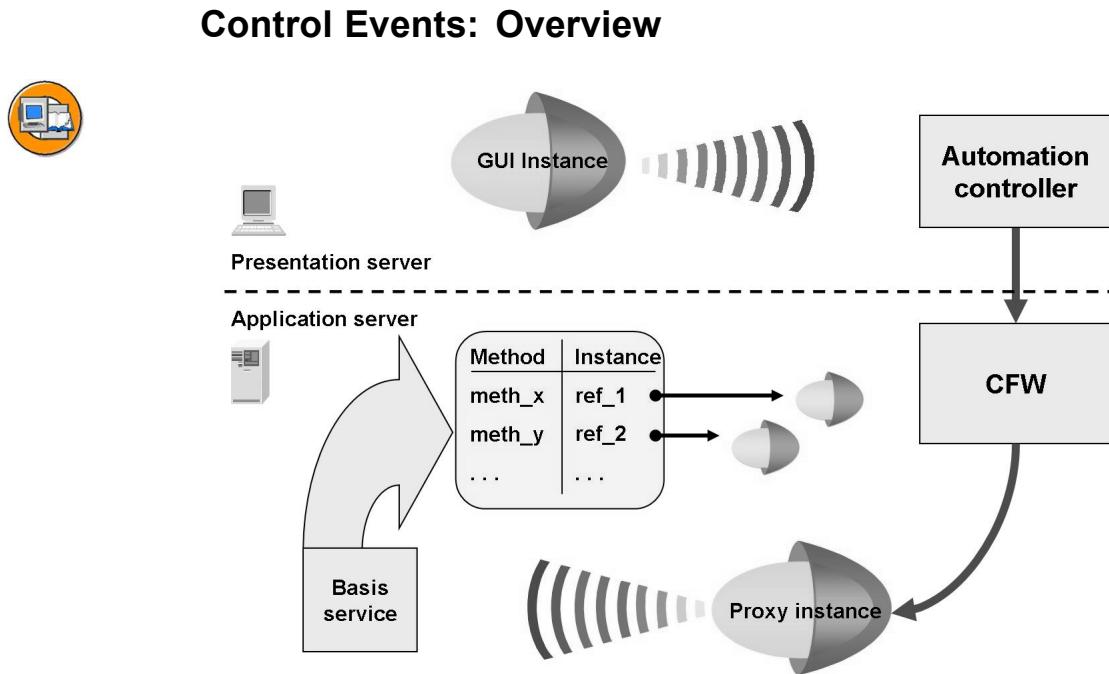
## Recapitulation: Control Events



**Figure 40: Recapitulation: Control Events**

EnjoySAP Controls can trigger events.

The Control Framework directs them to the appropriate proxy objects in your ABAP program. You can then process them using the event concept implemented in ABAP Objects.



**Figure 41: Control Events: Overview**

An event that is passed from a presentation server control passes through various levels.

**SAP GUI/Automation Controller:** The event triggered by the presentation server control is received by the Automation Controller in the SAP GUI. The Automation Control passes information about this event to the Basis services of the runtime system in the form of a special function code and some additional data.

The Basis services pass the information on to the proxy class in the ABAP program.

The proxy object in the ABAP program triggers an ABAP Objects event. This informs the components that are registered for the event that the event has occurred.

## Control Events: Flow Diagram

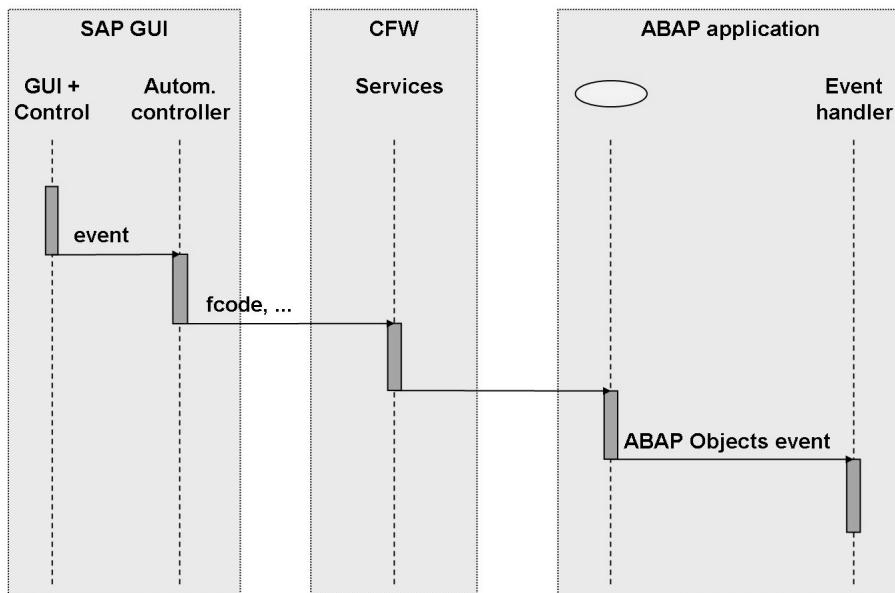


Figure 42: Control Events: Flow Diagram

At each level:

- Certain settings are **required** for the whole chain to function.
- Certain settings **can** be made to affect how the chain works.
- The following slides describe which settings must be made, or can be made, and where.

## Response to the ABAP Objects Event

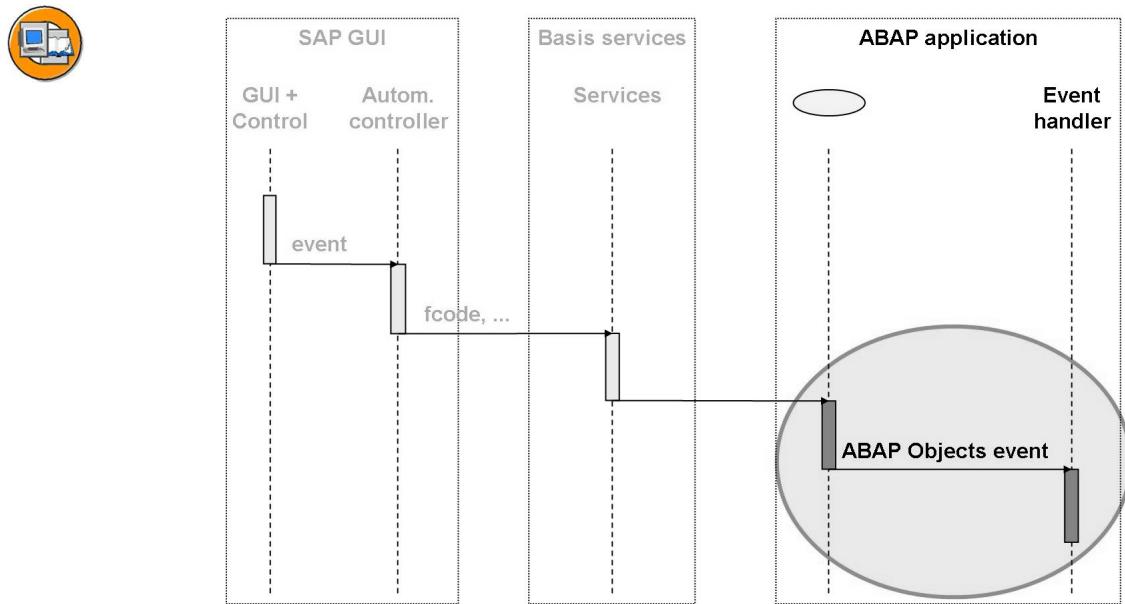


Figure 43: Response to the ABAP Objects Event

You will learn the steps necessary to make your application react to events triggered by a proxy object.

## Recapitulation: ABAP Objects Events

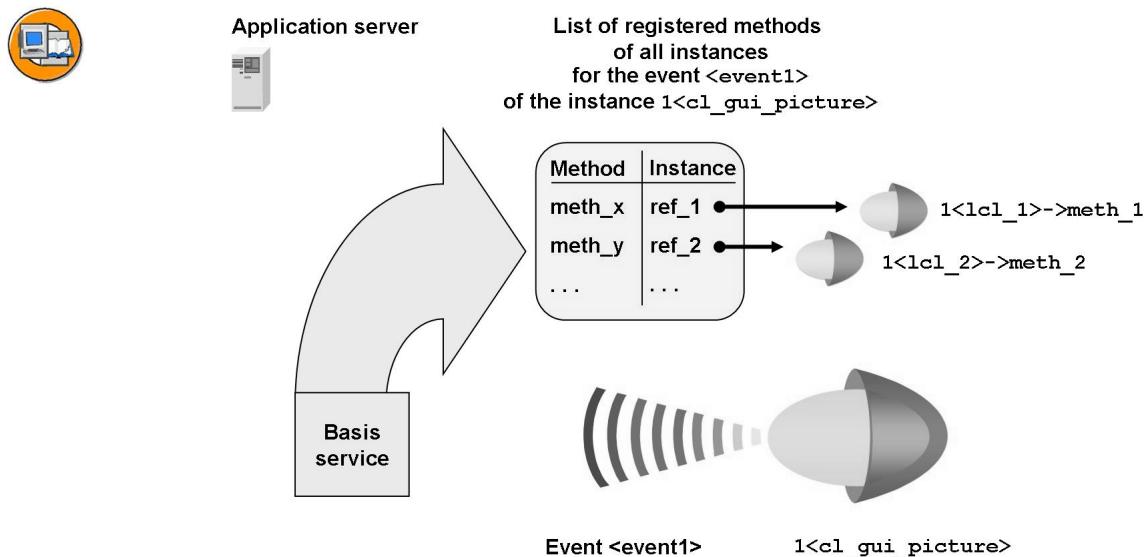


Figure 44: Recapitulation: ABAP Objects Events

An object can announce that its state has changed by triggering an event.

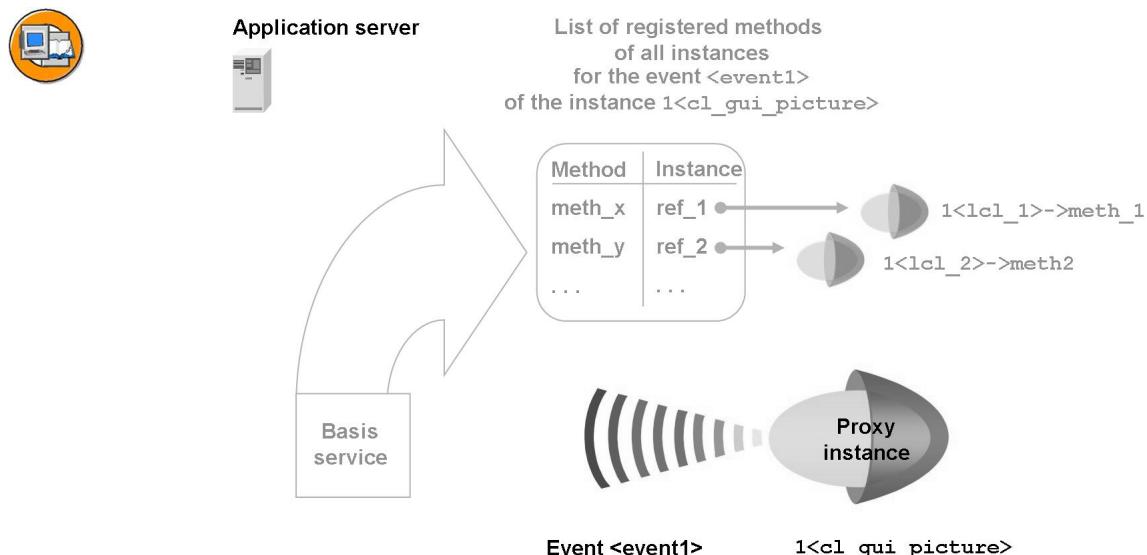
Other objects can contain handler methods, which are processed when the event occurs.

The object instances concerned must be entered in a list of objects that can trigger the event, so that the handler methods of these instances can be executed.

When the event is triggered, a Basis service takes responsibility for executing the methods registered in this list.

In an explicit method call, the caller has control of the program and knows what methods it is calling. When an event is triggered, the trigger does not know what methods (if any) will be called. This applies both to the point at which the event is defined and the point at which the event occurs at runtime.

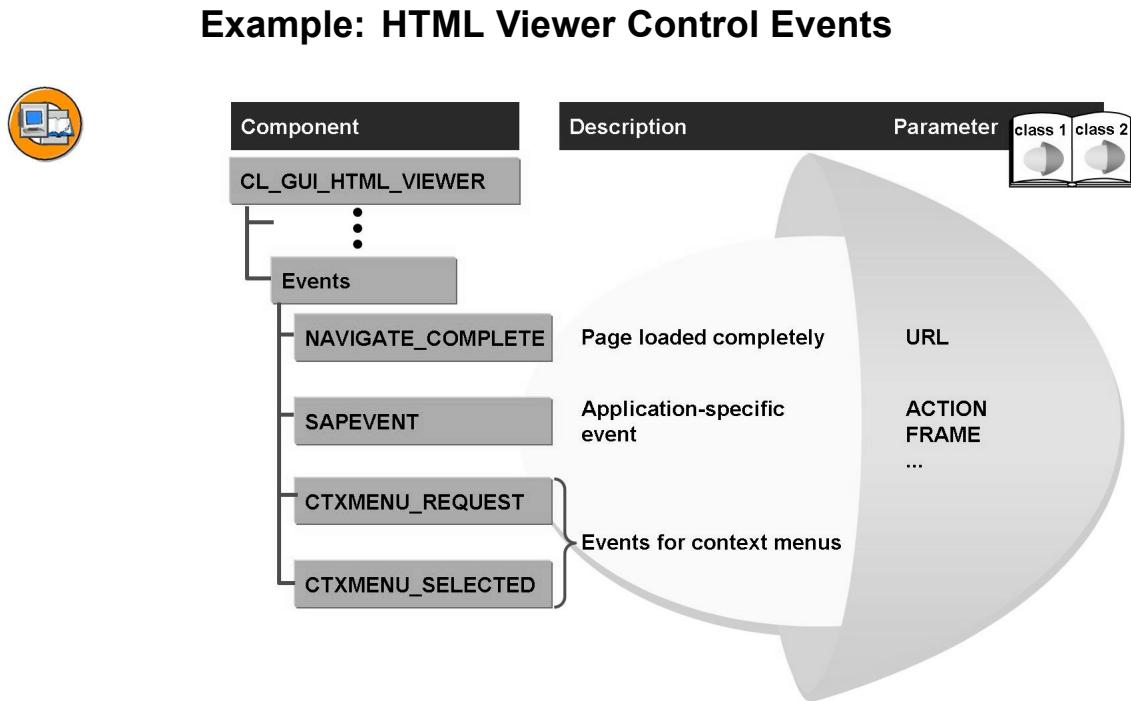
## Events of ABAP Objects Instances



**Figure 45: Events of ABAP Objects Instances**

If you want to react to events from an EnjoySAP control in your program, you first need to find out what events can be triggered by the object. You can do this in the definition of the proxy class of the control.

For more information on proxy classes, see the online documentation for the relevant control.



**Figure 46: Example: HTML Viewer Control Events**

The SAP HTML Viewer Control has four events:

- **NAVIGATE\_COMPLETE**: This event is triggered each time a page is fully loaded in the browser. The URL of the page is returned in the identically-named event parameter.
- **SAPEVENT**: You can trigger this event in an HTML page using the SAPEVENT:<action>?<data> tag. The event passes information, such as action code or frame name through the event parameters ACTION and FRAME.
- **CTXMENU\_REQUEST** and **CTXMENU\_SELECTED**: The events are triggered when context menus are used.
- For further information about events, refer to the online documentation.

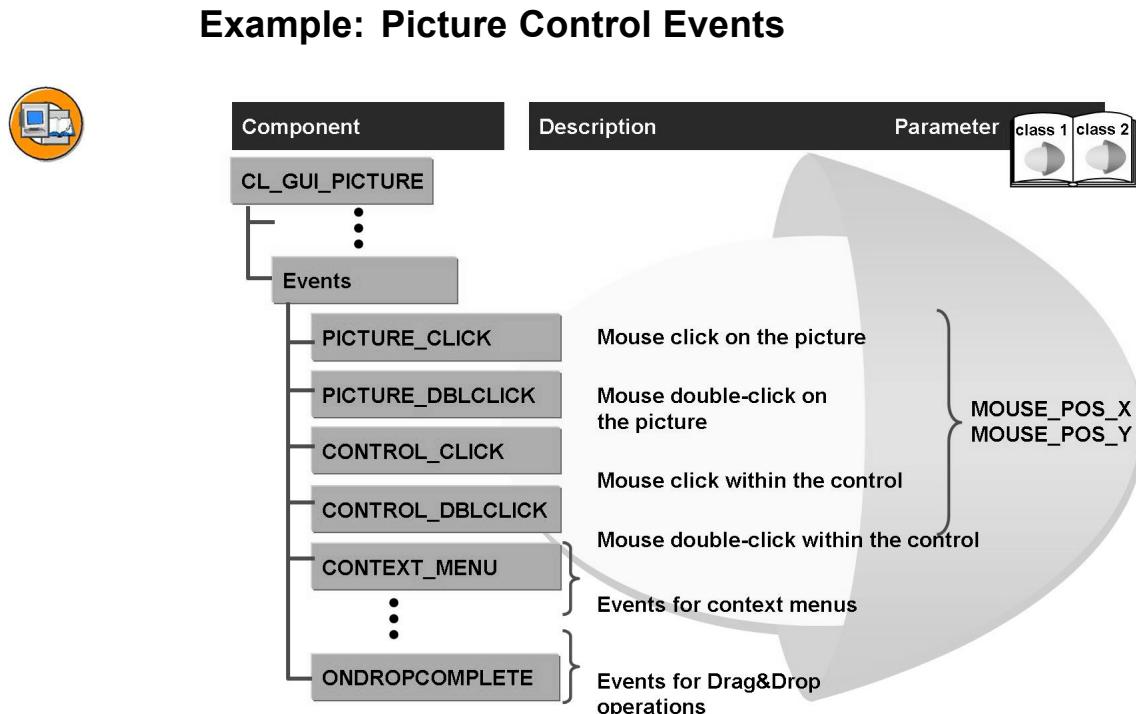


Figure 47: Example: Picture Control Events

The class for the SAP Picture Control has the following events:

- Four events are triggered by mouse operations on the control contents or the control area:
  - PICTURE\_CLICK: A single-click on the contents of the control (picture or icon).
  - PICTURE\_DBCLICK: A double-click on the contents of the control (picture or icon).
  - CONTROL\_CLICK: A single-click on the control area (but not on the contents).
  - CONTROL\_DBCLICK: A double-click on the control area (but not on the contents).
- Two events are triggered in conjunction with the context menu: CONTEXT\_MENU and CONTEXT\_MENU\_SELECTED.
- Four events are triggered in conjunction with Drag&Drop operations: ONDRAG, ONDROP, ONDROPCOMPLETE, and ONGETFLAVOUR.
- When the mouse click events are triggered, the position of the mouse is passed in the event parameters MOUSE\_POS\_X and MOUSE\_POS\_Y.
- Note that you cannot use the single- and double-click events together, since a single-click is always triggered before a double-click. The second event is lost when it is passed from the presentation to the application server.
- For further information about events, refer to the online documentation.

## Recapitulation: Generating Handler Methods

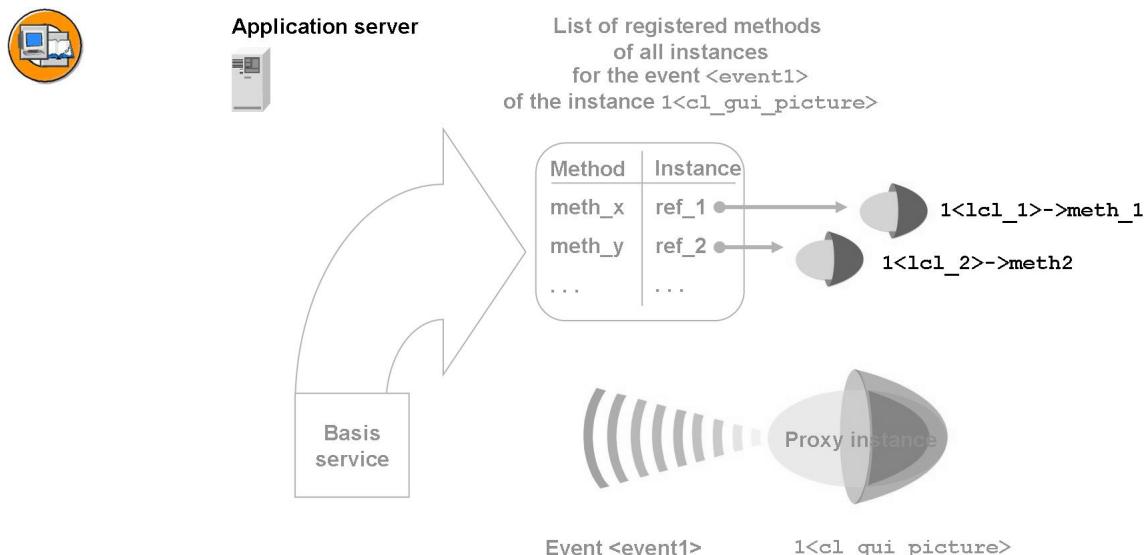


Figure 48: Recapitulation: Generating Handler Methods



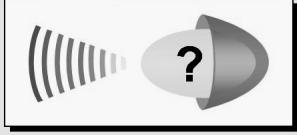
If you want to react to events from EnjoySAP controls, you need to define handler methods that are executed when the event is triggered. We will now look at the steps required to do this.

## Syntax: Handler Method for an Event

```

CLASS <class_name> DEFINITION.
  PUBLIC| PROTECTED| PRIVATE SECTION.
    [CLASS-]METHODS:
      <method_name> FOR EVENT <event_name> OF <class_name>
        IMPORTING <p1> ... <pn> [SENDER],
        <method_name1> ...
    ENDCCLASS.

    CLASS <class_name> IMPLEMENTATION.
      METHOD <method_name>.
        ...
        " method implementation
      ENDMETHOD.
    ENDCCLASS.
  
```



**Figure 49: Syntax: Handler Method for an Event**

Any class can contain handler methods for events of other classes.

In the definition part of an event handler method, you must use the FOR EVENT <event\_name> OF <class\_name> addition. This specifies the event <event\_name> of class <class\_name> to which the method can react.

When assigning names to event handler methods, you should observe the following rules: **on\_<event\_name>**, where <event\_name> is the name of the event.

The interface of an event handler method may only contain formal parameters that are defined in the declaration of the corresponding event. The attributes of the parameters are inherited from the event definition. The event handler method does not have to import all of the parameters passed in the RAISE EVENT statement.

All events have an implicit parameter called SENDER, which you can include in the IMPORTING parameters. This parameter allows the handlers of instance events to access the object raising the event.

The example contains a static method, declared using CLASS-METHODS. This can run without an instance of the class having been created.

When you declare an event handler method in the class, the class or its instances are, **in principle**, able to handle the event.

## Syntax Example: Handlers as Instance Methods



```

DATA: ref_pic TYPE REF TO cl_gui_picture.
      ...
}

CLASS lcl_event_handler DEFINITION .
  PUBLIC SECTION.
    METHODS on_picture_click
      FOR EVENT picture_click
      OF cl_gui_picture ...
  ...
ENDCLASS.

CLASS lcl_event_handler IMPLEMENTATION .
  METHOD on_picture_click.
  ...
ENDMETHOD.
ENDCLASS.

...
DATA: ref_h TYPE REF TO lcl_event_handler.
      ...
CREATE OBJECT ref_pic.
CREATE OBJECT ref_h.
      
```

cl\_gui\_picture  
has the event  
picture\_click

The instance method  
on\_picture\_click  
can react to the event  
picture\_click

Implementing the  
instance method  
on\_picture\_click

Creating a reference  
variable for the handler  
object

Instances of the classes:  
cl\_gui\_picture and  
lcl\_eventhandler

**Figure 50: Syntax Example: Handlers as Instance Methods**

To create a handler object for an event, first define a local class. This class contains a public (PUBLIC SECTION) method that can react to an event.

After defining the class, you must implement it. In the implementation of the handler method, create the source code that you want to execute when the event is raised.

Then create a reference variable with reference to your local class.

Finally, create an appropriate object.



## Syntax Example: Handlers as Class Methods

```
CLASS lcl_event_handler DEFINITION.  
  PUBLIC SECTION.  
    CLASS-METHODS: on_picture_click  
      FOR EVENT picture_click OF cl_gui_picture  
      IMPORTING mouse_pos_x mouse_pos_y.  
    ...  
  ENDCLASS.  
  
CLASS lcl_event_handler IMPLEMENTATION.  
  METHOD on_picture_click.  
  ...  
  MESSAGE i016(bc412)  
    WITH mouse_pos_x mouse_pos_y.  
  ENDMETHOD.  
  ...  
ENDCLASS.
```

The static method `on_picture_click` can react to the event `picture_click`

Implementing the static method `on_picture_click`

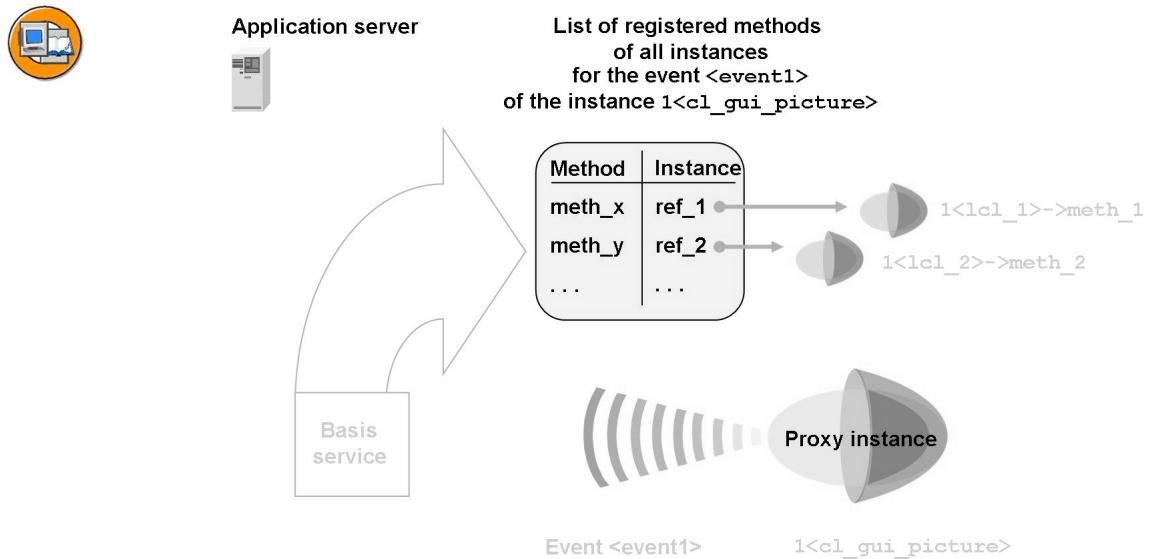
You do not need to create an instance of an object

Figure 51: Syntax Example: Handlers as Class Methods

A simple way of creating a handler method is to use a **static** method of a local class.

The method receives the coordinates of the mouse position from the event and displays an information message containing the coordinates and a text.

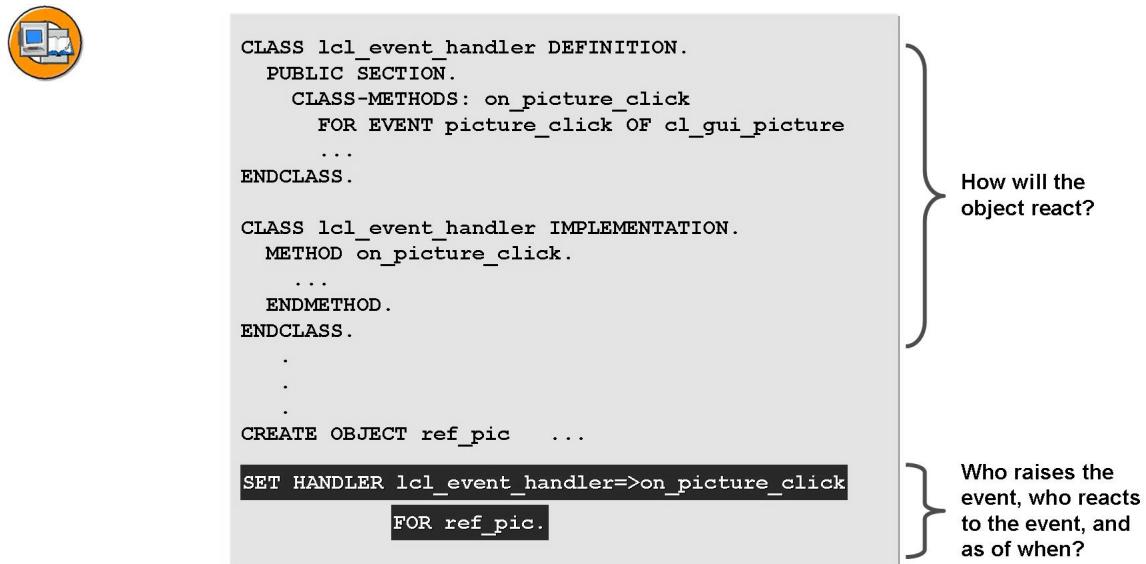
## Registering Handler Methods



**Figure 52: Registering Handler Methods**

When handling an event you must specify the **instances** to which the system will react, **how** it will react, and **when**.

## Creating the Link Between the Object Raising the Event and the Handler Method



**Figure 53: Creating the Link Between the Object Raising the Event and the Handler Method**

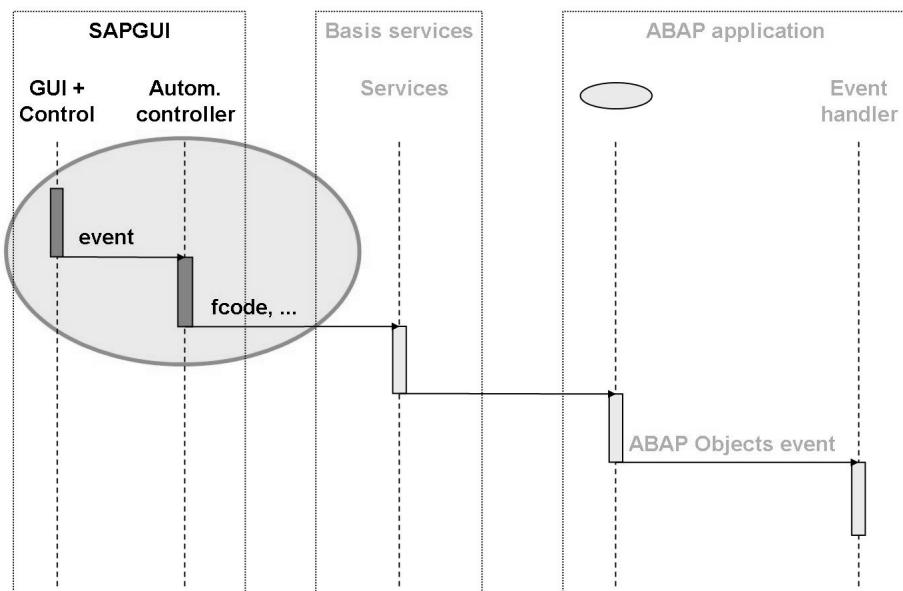
You must specify the triggering object at run time for an event handler method to react to an event.

The SET HANDLER statement links a list of handler methods with event triggers.

The graphic shows that the static method (not instance-specific) `on_picture_click` of class `lcl_event_handler` runs when the instance to which the object reference in `my_picture` is pointing raises the event `picture_click`.

For further information about registering handler methods for events, refer to the keyword documentation for the SET HANDLER statement.

## Raising the Event on the Presentation Server



**Figure 54: Raising the Event on the Presentation Server**

The following slides deal with the Automation Controller in the SAP GUI in more detail.

## The Automation Controller as an Event Filter

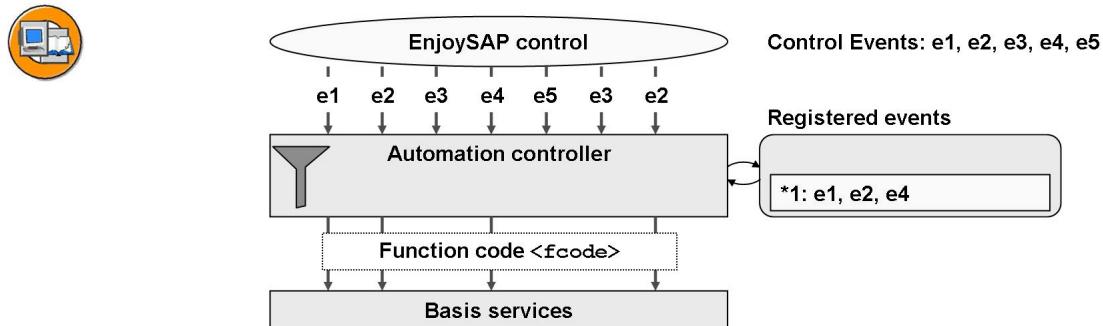


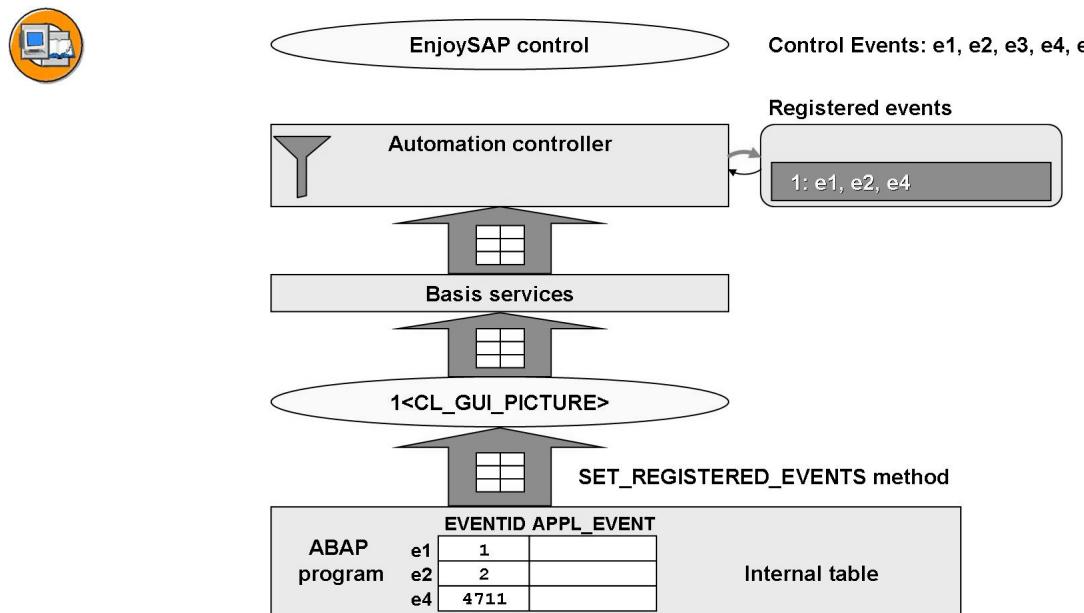
Figure 55: The Automation Controller as an Event Filter

In the SAP GUI, the Automation Controller's job is to act as a filter for the various control events. The Automation Controller has a list of all events registered for an instance. Only these events are passed from the Automation Controller to the Basis services (in the form of function codes).

The event list that the Automation Controller administers for each instance must be filled from the **ABAP application** that is going to handle the control events.

This process is known as **registration (in the Control Framework)**. However, it is a different process to the event registration in proxy instances that we have already seen in ABAP Objects.

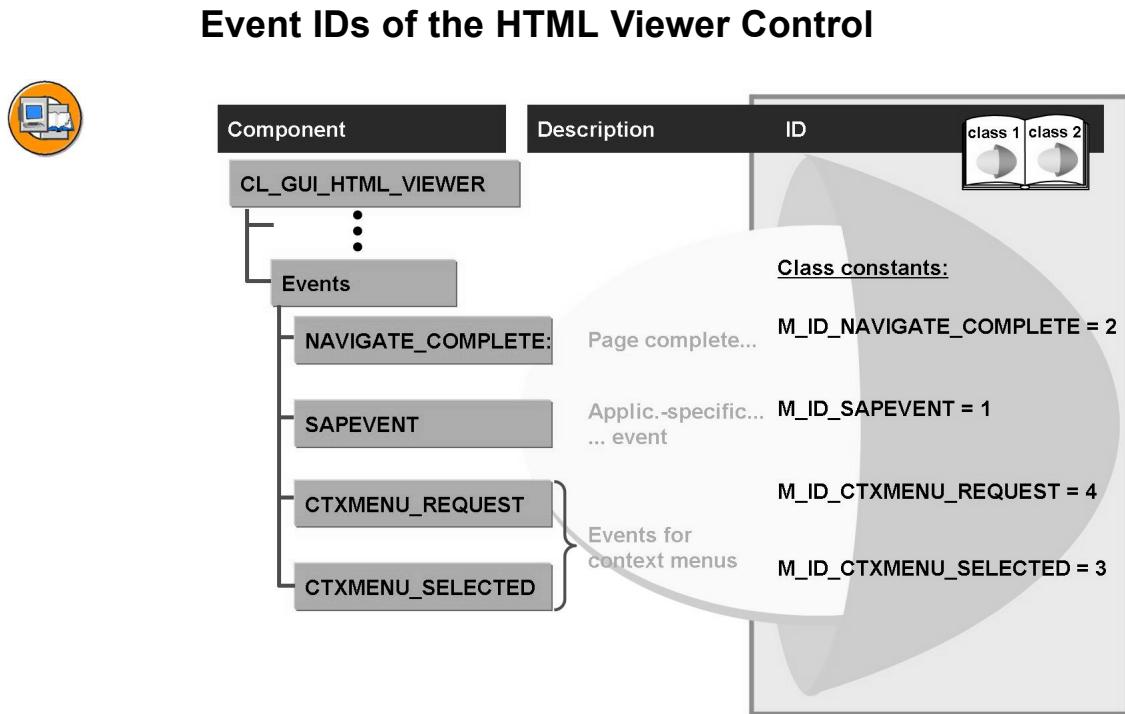
## Constructing Event Filters



**Figure 56: Constructing Event Filters**

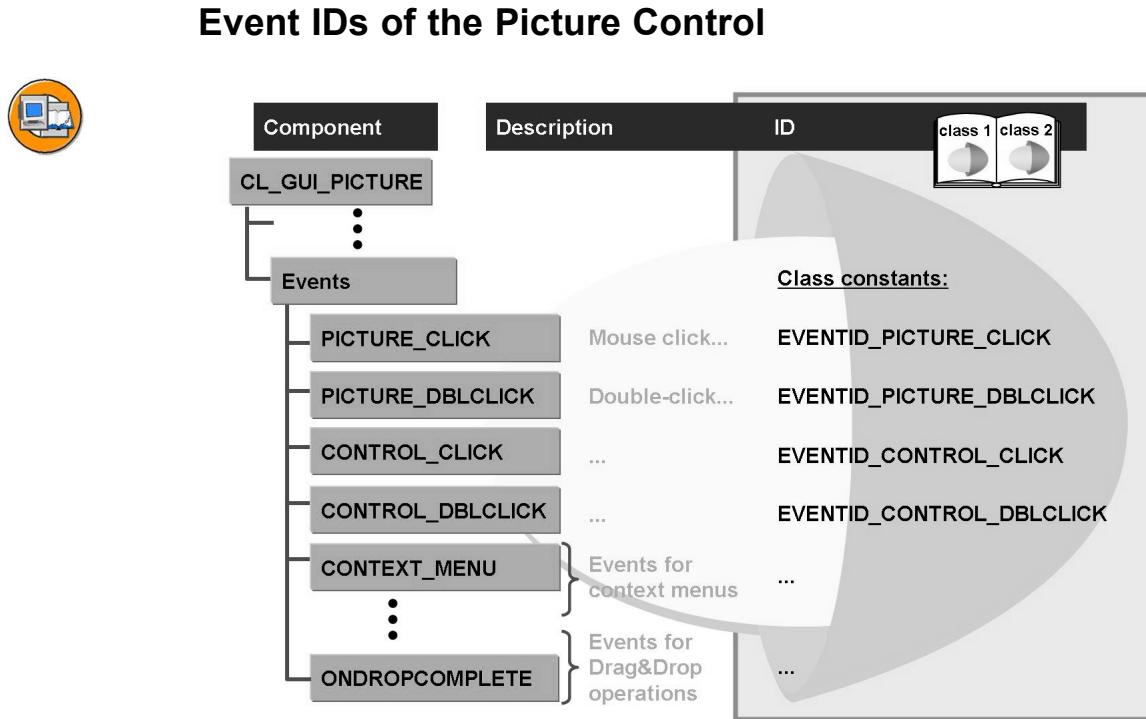
You construct the list of events to be registered in the Automation Controller as follows:

- In the ABAP program, an internal table is filled containing the event IDs of the events you want to register with the Automation Controller.
- This internal table is sent to the proxy object using the **SET\_REGISTERED\_EVENTS** instance method that every control wrapper offers. This method passes the data to the Automation Queue in the CFW.
- When the next synchronization (FLUSH) occurs, the list is automatically sent to the Automation Controller.
- You enter the IDs of the events in the **EVENTID** in the internal table (in the ABAP program).
- To find the event IDs associated with a particular EnjoySAP Control, see the online documentation for that control or search in the Class Builder in the Attributes of the class.



**Figure 57: Event IDs of the HTML Viewer Control**

Use the “self-addressing” class constants shown here when you construct the event filter in the Automation Controller.



**Figure 58: Event IDs of the Picture Control**

Use the “self-addressing” class constants shown here when you construct the event filter in the Automation Controller.

Note that naming conventions for the Picture Control are different than those used with the SAP HTML Viewer. This means that it is essential to study the attribute definitions of whatever controls you use, in the Class Builder.

## Syntax Example: Constructing Event Filters



```

TYPE-POOLS cntl.
DATA: it_events TYPE cntl_simple_events, " internal (event) table
      wa_events LIKE LINE OF it_events. " work area

* register events: 1) picture click
      wa_events-eventid      = cl_gui_picture->eventid_picture_click.

      INSERT wa_events INTO TABLE it_events.
*           2) control dblclick
      ...
      .

CALL METHOD my_picture->set_registered_events
      EXPORTING
          events = it_events
      EXCEPTIONS
          others = 1.

      IF sy-subrc NE 0.
*           MESSAGE a....
ENDIF.

```

**Figure 59: Syntax Example: Constructing Event Filters**

Create an internal table (above: it\_events) and assign it the type cntl\_simple\_events. You also need a work area for the internal table (above: wa\_events).

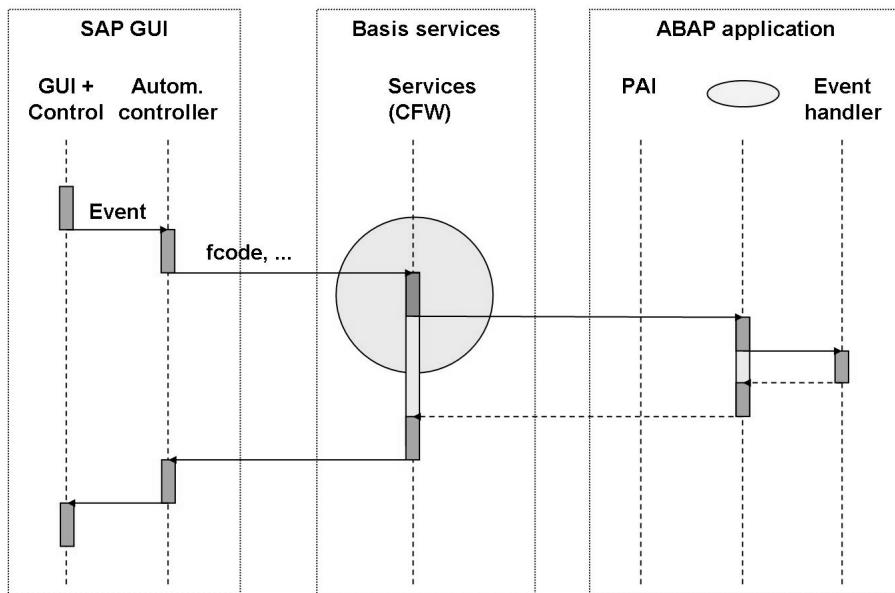
The cntl\_simple\_events table type is defined in the CNTL type pool. Add this type pool to your program.

Fill the internal table with the IDs of the events you want to register (eventid field).

Call the instance method SET\_REGISTERED\_EVENTS for the proxy instance, and pass the internal table to the interface parameter EVENTS.

If the method raises an exception, have the program display a termination message (type A).

## System Event



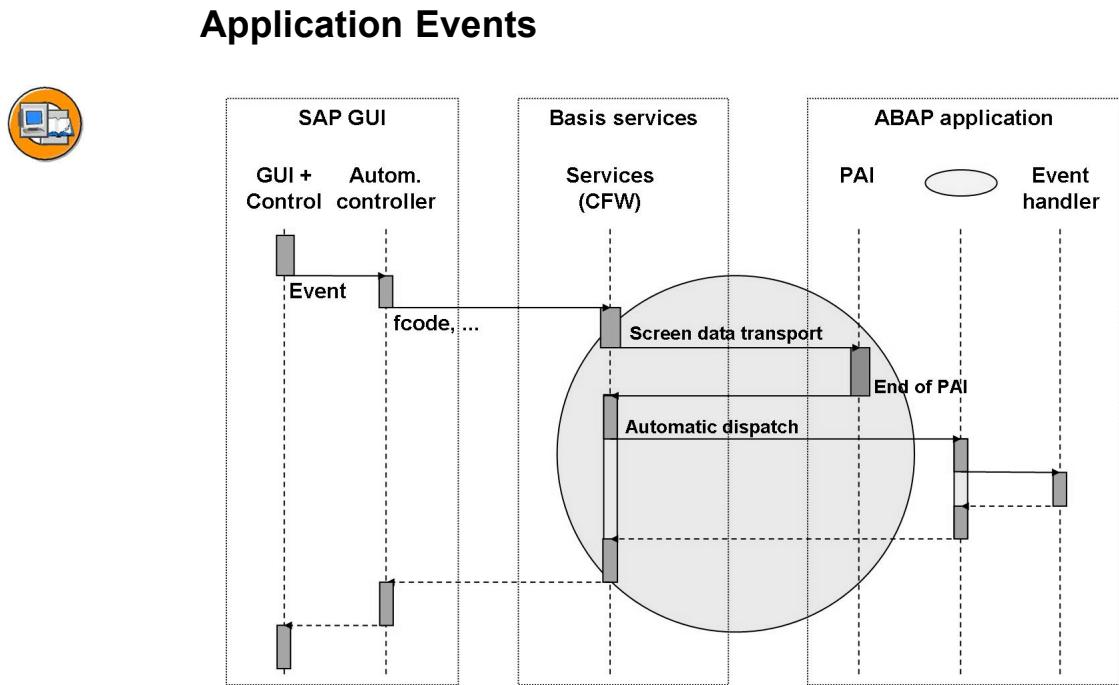
**Figure 60: System Event**

Control events that you register with the Automation Controller as we have just seen are known as **system events**.

With system events, the Basis services inform the proxy object about the event raised on the presentation server directly when the corresponding function code arrives.

The proxy object then reacts by raising the appropriate event in the ABAP program. The screen containing the presentation server control is not processed – that is, the PAI event is not triggered.

Consequently, no data from other screen elements is transferred back to the program. The event handling is restricted to the “object-oriented world” of the ABAP program.



**Figure 61: Application Events**

Application events are processed differently than system events: If a control triggers an application event, control is returned to its container screen.

The Automation Controller passes the information connected to the event (EXPORTING parameters and function code) to the Basis services on the application server.

The Basis services trigger the PAI event of the container screen.

At the beginning of the PAI event, the normal field transport takes place from the screen to the program.

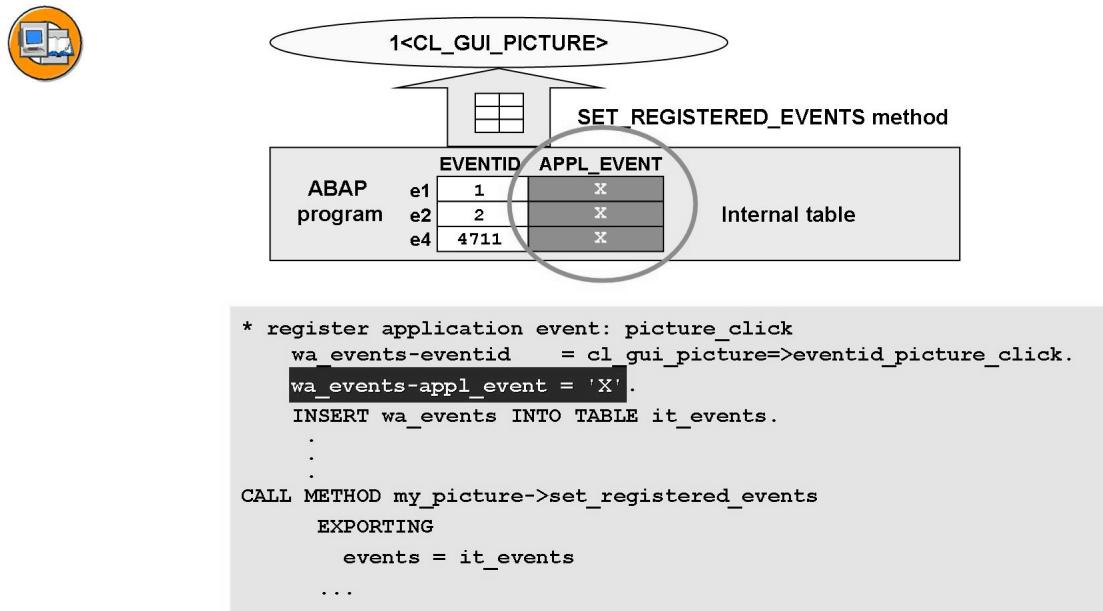
Since control is passed back the container screen of the control, the ABAP application determines itself when to pass the control on to the proxy object to complete the event handling, using the DISPATCH method of the class CL\_GUI\_CFW.

This method is called **by the CFW at the end of the PAI event**.

However, you can bring event handling **forward** within the PAI, by calling the method CALL METHOD cl\_gui\_cfw=>dispatch.

When this method is triggered, the Basis services pass the information the control event needs to the proxy object. The proxy event can then trigger the corresponding event for the ABAP application.

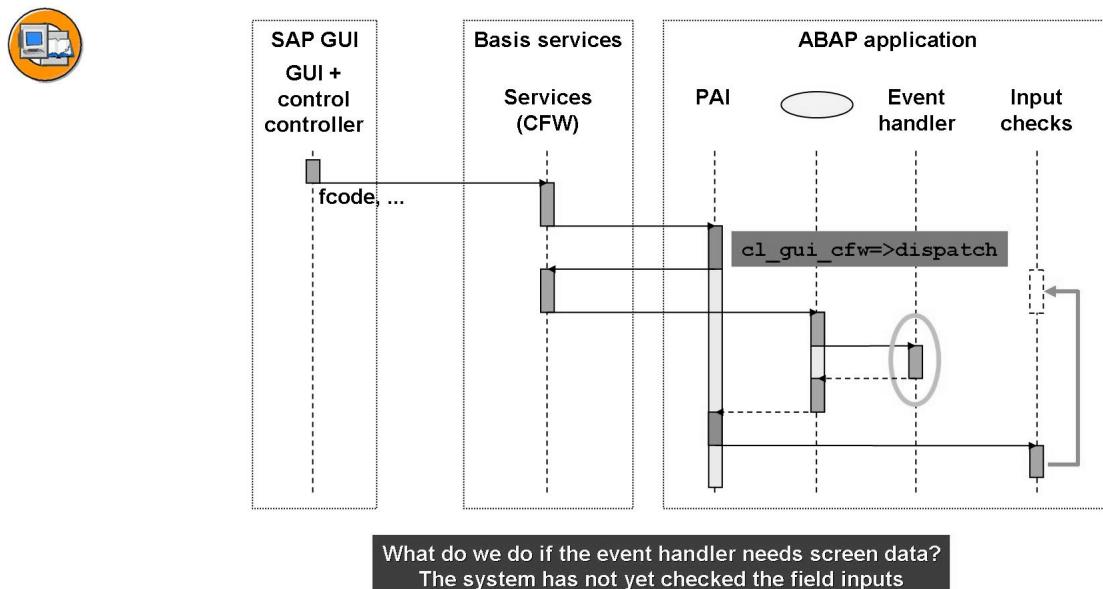
## Syntax Example: Registering an Application Event



**Figure 62: Syntax Example: Registering an Application Event**

You specify that an event should be an application event when you register it with the CFW: In your internal table, fill the appl\_event field with an “X”.

## Application Events and Screen Input Checks I



**Figure 63: Application Events and Screen Input Checks I**

If you use application events in your application, you can specify the order in which you want your own check modules and the event handler methods to run by calling the DISPATCH method:

- If the program is to handle the events **before** executing the check modules, bear in mind that you cannot rely on being able to access current screen data. Generally field inputs are only transported when the corresponding check module is called (the FIELD statement).
- Conversely, if you want first to execute the modules for screen-specific input checks, the DISPATCH method must be called **after** the check modules. You can either do this explicitly, or let the system call it automatically at the end of the PAI.

## Application Events and Screen Input Checks II

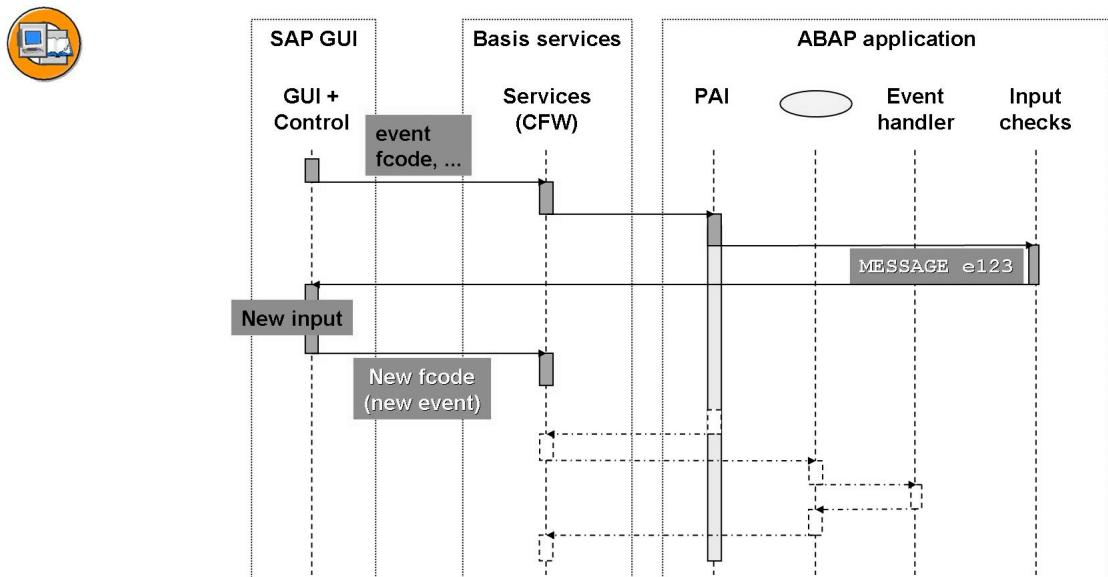


Figure 64: Application Events and Screen Input Checks II

If you perform the screen-specific checks before the event handling, it is possible for control events to be lost during an error dialog.

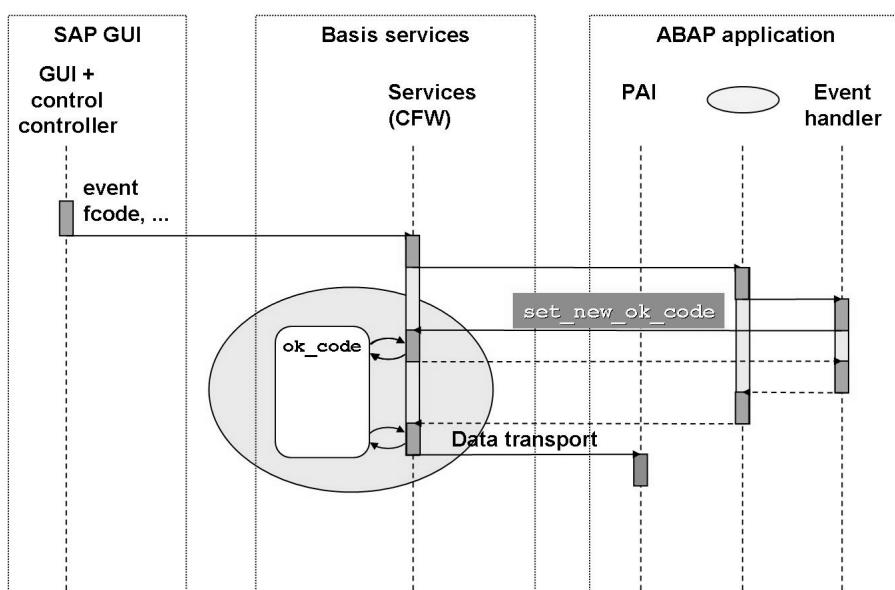
The presentation server control triggers an event that is passed to the application server as an application event. Control returns to the container screen, which processes the screen-specific check modules.

If an error or warning message (MESSAGE Ennn or MESSAGE Wnnn, where nnn is the message number) occurs during a check module, the user can change his or her entries on the screen and then trigger a different event with a different function code.

The new function code overwrites the function code linked to the event originally triggered by the user. The `cl_gui_cfw=>dispatch` call then does not lead to any event handling.

This does not occur if the new function is triggered by a screen element such as a pushbutton or menu entry. The original event is not lost.

## System Events That Trigger a PAI Event



**Figure 65: System Events That Trigger a PAI Event**

If you handle a control event as a system event and want to pass control to the container screen **after** the event handling, you must use the method `cl_gui_cfw=>set_new_ok_code`.

This method stores the function code – passed using the method interface – in the OK code field for the screen (the `OK_CODE` parameter). When the event handling has been completed, a PAI event with the new function code is triggered.

Note that the event handling takes place before any field transport, since the PAI is not triggered until **after** the event handling.

## Exercise 4: Control Events

### Exercise Objectives

After completing this exercise, you will be able to:

- Define and implement a static method of a local class as an event handler
- Register a static handler method for an ABAP Objects event
- Register control events with the Control Framework
- Find information on control events in the ABAP Workbench
- Use the PICTURE\_CLICK event

### Business Example

You should make it possible for your user to ascertain the coordinates of a point in a picture by clicking it. Implement this using a control event.

**Program:** ZBC412\_##\_CFW\_EX4

**Template:** SAPBC412\_CFWS\_EXERCISE3

**Model solution:** SAPBC412\_CFWS\_EXERCISE4

where ## is the group number.

#### Task 1:

Copy the Template.

1. Copy your solution from the previous exercise (ZBC412\_##\_CFW\_EX3) or the appropriate model solution (SAPBC412\_CFWS\_EXERCISE3) with all their sub-objects to the name **ZBC412\_##\_CFW\_EX4**. Get to know how your copy of the program works.

#### Task 2:

In your program, create a local class that contains a method for handling the PICTURE\_CLICK event for your picture control instance.

1. Use the *Class Builder* to find out about the PICTURE\_CLICK event interface of the `c1_gui_picture` class.
2. In the declaration part of your program, define a local class (suggested name: `lcl_event_handler`).
3. The local class should contain a public, static method, which the system will use as an event-handling method.

*Continued on next page*

Using the CLASS-METHODS statement, define a method for handling the PICTURE\_CLICK event in the cl\_gui\_picture class (suggested name: `on_picture_click`). Define the method interface so that the event passes the coordinates of the point clicked to the method.

4. Implement the method in the implementation part of the class. Have the program react to the event by displaying the coordinates in the information message 016.

### Task 3:

Register the event handler method.

1. Register the static method for the picture control instance on the application server. To do this, use the SET HANDLER statement in the `init_control_processing` module.

### Task 4:

Register the control event PICTURE\_CLICK of your control instance with the Control Framework.

1. In the declaration part of your program, create two more data objects:
  - An internal table, used to pass the events (which are to be registered with the CFW) to the instance method `set_registered_events` of the `cl_gui_picture` class. (We suggest the name: `it_events`).
  - A work area, which you will need to fill the internal table (suggested name: `wa_events`).
2. Assign the `cntl_simple_events` type definition in the type group `CNTL` to the internal table. Assign an appropriate work area for this internal table.
3. In the Class Builder, find out more about the names of the class constants for control events (Event IDs).
4. Fill the internal table with data for the PICTURE\_CLICK event. Register the event with the CFW using the instance method `set_registered_events`. Do this by extending the PBO module `init_control_processing`.

*Continued on next page*

## Task 5: Optional

**Program:** ZBC412\_##\_CFW\_EX4

**Template:** SAPBC412\_CFWS\_EXERCISE4

**Model solution:** SAPBC412\_CFWS\_EXERCISE4A

where ## is the group number.

Extend your program so that if users double-click the control area outside the picture, the program displays the cursor coordinates.

To do this, use the CONTROL\_DBCLICK event. Proceed as for steps 4-2 to 4-4 in exercise 4, for the new event.

1. Use the same local class. Add another public static method to handle CONTROL\_DBCLICK events of your picture control instance.
2. Have the system react to the event by displaying the information message 017.
3. Register this static method for the picture control instance on the application server. To do this, use the SET HANDLER statement in the init\_control\_processing module.
4. Register the control event CONTROL\_DBCLICK of your control instance with the Control Framework.

## Solution 4: Control Events

### Task 1:

Copy the Template.

1. Copy your solution from the previous exercise (ZBC412\_##\_CFW\_EX3) or the appropriate model solution (SAPBC412\_CFW\_EXERCISE3) with all their sub-objects to the name **ZBC412\_##\_CFW\_EX4**. Get to know how your copy of the program works.
  - a) –

### Task 2:

In your program, create a local class that contains a method for handling the PICTURE\_CLICK event for your picture control instance.

1. Use the *Class Builder* to find out about the PICTURE\_CLICK event interface of the cl\_gui\_picture class.
  - a) –
2. In the declaration part of your program, define a local class (suggested name: lcl\_event\_handler).
  - a)

```
CLASS lcl_event_handler DEFINITION.
```

```
ENDCLASS.
```

3. The local class should contain a public, static method, which the system will use as an event-handling method.

Using the CLASS-METHODS statement, define a method for handling the PICTURE\_CLICK event in the cl\_gui\_picture class (suggested name: on\_picture\_click). Define the method interface so that the event passes the coordinates of the point clicked to the method.

- a)

```
CLASS lcl_event_handler DEFINITION.
PUBLIC SECTION.
CLASS-METHODS: on_picture_click
  FOR EVENT picture_click OF cl_gui_picture
    IMPORTING mouse_pos_x mouse_pos_y.
ENDCLASS.
```

*Continued on next page*

4. Implement the method in the implementation part of the class. Have the program react to the event by displaying the coordinates in the information message 016.
- a)

```
CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_picture_click.
MESSAGE i016 WITH mouse_pos_x mouse_pos_y.
ENDMETHOD.
ENDCLASS.
```

### Task 3:

Register the event handler method.

1. Register the static method for the picture control instance on the application server. To do this, use the SET HANDLER statement in the `init_control_processing` module.

a)

```
SET HANDLER lcl_event_handler=>on_picture_click FOR ref_picture.
```

### Task 4:

Register the control event PICTURE\_CLICK of your control instance with the Control Framework.

1. In the declaration part of your program, create two more data objects:  
 An internal table, used to pass the events (which are to be registered with the CFW) to the instance method `set_registered_events` of the `cl_gui_picture` class. (We suggest the name: `it_events`).  
 A work area, which you will need to fill the internal table (suggested name: `wa_events`).  
 a) -
2. Assign the `cntl_simple_events` type definition in the type group `CNTL` to the internal table. Assign an appropriate work area for this internal table.  
 a)

```
DATA:
...     it_events      TYPE cntl_simple_events, " internal (event) table
...     wa_events      LIKE LINE OF it_events.   " work area
```

*Continued on next page*

3. In the Class Builder, find out more about the names of the class constants for control events ( Event IDs).
- a) –
4. Fill the internal table with data for the PICTURE\_CLICK event.  
Register the event with the CFW using the instance method  
`set_registered_events`. Do this by extending the PBO module  
`init_control_processing`.
- a)

```
wa_events-eventid      = cl_gui_picture->eventid_picture_click.  
wa_events-appl_event = ' '.  
INSERT wa_events INTO TABLE it_events.  
  
CALL METHOD ref_picture->set_registered_events  
  EXPORTING  
    events = it_events  
  EXCEPTIONS  
    OTHERS = 1.  
  
IF sy-subrc NE 0.  
  MESSAGE a012.  
ENDIF.
```

*Continued on next page*

## Task 5: Optional

**Program:** ZBC412\_##\_CFW\_EX4

**Template:** SAPBC412\_CFWS\_EXERCISE4

**Model solution:** SAPBC412\_CFWS\_EXERCISE4A

where ## is the group number.

Extend your program so that if users double-click the control area outside the picture, the program displays the cursor coordinates.

To do this, use the CONTROL\_DBLCHECK event. Proceed as for steps 4-2 to 4-4 in exercise 4, for the new event.

1. Use the same local class. Add another public static method to handle CONTROL\_DBLCHECK events of your picture control instance.

a)

```
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
    ...
      on_control_dbclick
        FOR EVENT control_dbclick OF cl_gui_picture
          IMPORTING mouse_pos_x mouse_pos_y.
    ENDCLASS.
```

2. Have the system react to the event by displaying the information message 017.

a)

```
CLASS lcl_event_handler IMPLEMENTATION.
...
  METHOD on_control_dbclick.
    MESSAGE i017 WITH mouse_pos_x mouse_pos_y.
  ENDMETHOD.
ENDCLASS.
```

*Continued on next page*

3. Register this static method for the picture control instance on the application server. To do this, use the SET HANDLER statement in the init\_control\_processing module.

a)

```
SET HANDLER lcl_event_handler=>on_control_dblclick
FOR ref_picture.
```

4. Register the control event CONTROL\_DBLC CLICK of your control instance with the Control Framework.

a)

```
wa_events-eventid      = cl_gui_picture=>eventid_control_dblclick.
wa_events-appl_event = ' '.
INSERT wa_events INTO TABLE it_events.
```

## Result

### Complete Solution to Exercises

#### Model solution SAPBC412\_CFW\_S\_EXERCISE4A

#### Screen flow logic

#### *SCREEN 0100*

```
PROCESS BEFORE OUTPUT.
```

```
MODULE status_0100.
```

```
MODULE init_control_processing.
```

```
PROCESS AFTER INPUT.
```

```
MODULE exit_command_0100 AT EXIT-COMMAND.
```

```
MODULE user_command_0100.
```

## ABAP program

### *Local Classes*

```
*&-----*
*& Report  SAPBC412_CFW_S_EXERCISE          *
*&                                         *
```

*Continued on next page*

```

*&-----*
REPORT  sapbc412_cfws_exercise MESSAGE-ID bc412.
*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
*      Definition of a local class containing event handler      *
*      methods for picture control objects                      *
*-----*_
CLASS lcl_event_handler DEFINITION.
PUBLIC SECTION.
CLASS-METHODS:
on_picture_click
    FOR EVENT picture_click OF cl_gui_picture
    IMPORTING mouse_pos_x mouse_pos_y,
*-----*
on_control_dblclick
    FOR EVENT control_dblclick OF cl_gui_picture
    IMPORTING mouse_pos_x mouse_pos_y.
ENDCLASS.

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
*      Corresponding class implementation                      *
*-----*
CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_picture_click.
MESSAGE i016 WITH mouse_pos_x mouse_pos_y.
ENDMETHOD.

METHOD on_control_dblclick.
MESSAGE i017 WITH mouse_pos_x mouse_pos_y.
ENDMETHOD.
ENDCLASS.

```

### ***Data Declarations***

```

* global types: Type Pool
TYPE-POOLS: cntl.
* data types
TYPES:
t_url          TYPE bapiuri-uri.

```

*Continued on next page*

```

* data declarations
* screen specific
DATA:
    ok_code      TYPE sy-ucomm,          " command field
    copy_ok_code LIKE ok_code,          " copy of ok_code
    l_answer     TYPE c,                " return flag (used in
                                         " standard user dialogs)

* control specific: object references
ref_container TYPE REF TO cl_gui_custom_container,
ref_picture   TYPE REF TO cl_gui_picture,

* control specific: auxiliary fields
l_url         TYPE t_url,           " URL of picture to be shown

* event handling
it_events      TYPE cntl_simple_events, " internal (event) table
wa_events     LIKE LINE OF it_events. " work area

```

### **Main ABAP Program: Event Blocks**

```

* start of main program
START-OF-SELECTION.

CALL FUNCTION 'BC412_BDS_GET_PIC_URL' fetch URL of first picture
*      EXPORTING                                " from BDS
*          NUMBER        = 1
          IMPORTING
          url          = l_url
          EXCEPTIONS
          OTHERS        = 1.

IF sy-subrc <> 0.                      " no picture --> end of program
  LEAVE PROGRAM.
ENDIF.

CALL SCREEN 100.                         " container screen for SAP-Enjoy
                                         " controls

* end of main program

```

### **Modules**

*Continued on next page*

```

*&-----*
*&      Module  INIT_CONTROL_PROCESSING  OUTPUT
*&-----*
*      Implementation of EnjoySAP control processing:
*      1)  create container object and link to screen area
*      2)  create picture object and link to container
*      3)  load picture into picture control
*      4)  event handling
*          a1) register event PICTURE_CLICK at CFW
*          a2) register event CONTROL_DBCLICK at CFW
*          b1) register event handler method ON_PICTURE_CLICK
*              for the picture control object
*          b2) register event handler method ON_CONTROL_DBCLICK
*              for the picture control object
*-----*
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.      " prevent re-processing on ENTER
*      create container object and link to screen area
  CREATE OBJECT ref_container
    EXPORTING
      container_name = 'CONTROL_AREA1'
  EXCEPTIONS
    others = 1.

  IF sy-subrc NE 0.
    MESSAGE a010.                  " cancel program processing
  ENDIF.

*      create picture control and link to container object
  CREATE OBJECT ref_picture
    EXPORTING
      parent = ref_container
  EXCEPTIONS
    others = 1.

  IF sy-subrc NE 0.
    MESSAGE a011.                  " cancel program processing
  ENDIF.

*      load picture into picture control
  CALL METHOD ref_picture->load_picture_from_url
    EXPORTING
      url     = l_url
  EXCEPTIONS

```

*Continued on next page*

```

        OTHERS = 1.

        IF sy-subrc NE 0.
          MESSAGE a012.
        ENDIF.

*     event handling
*     1. register events for control framework
  wa_events-eventid    = cl_gui_picture->eventid_picture_click.
  wa_events-appl_event = ' '.
  INSERT wa_events INTO TABLE it_events.

  wa_events-eventid    = cl_gui_picture->eventid_control_dblclick.
  wa_events-appl_event = ' '.
  INSERT wa_events INTO TABLE it_events.

CALL METHOD ref_picture->set_registered_events
  EXPORTING
    events = it_events
  EXCEPTIONS
    OTHERS = 1.

        IF sy-subrc NE 0.
          MESSAGE a012.
        ENDIF.

*     2. set event handler for ABAP object instance: ref_picture
  SET HANDLER lcl_event_handler=>on_picture_click
    FOR ref_picture.
  SET HANDLER lcl_event_handler=>on_control_dblclick
    FOR ref_picture.

ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING  OUTPUT

*-----*
*&      Module  USER_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type ' ':
*      - push buttons on screen 100
*      - GUI functions
*-----*

```

*Continued on next page*

```

MODULE user_command_0100 INPUT.
copy_ok_code = ok_code.
CLEAR ok_code.

CASE copy_ok_code.
  WHEN 'BACK'.                      " back to program, leave screen
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *           DEFAULTOPTION = 'Y'
        textline1     = text-004
        textline2     = text-005
        titel         = text-007
        cancel_display = ' '
      IMPORTING
        answer       = l_answer.

  CASE l_answer.
    WHEN 'J'.
      PERFORM free_control_ressources.
      LEAVE TO SCREEN 0.
    WHEN 'N'.
      SET SCREEN sy-dynnr.
  ENDCASE.

  WHEN 'STRETCH'.                  " picture operation: stretch to fit area
    CALL METHOD ref_picture->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture->display_mode_stretch
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'NORMAL'.                  " picture operation: fit normal size
    CALL METHOD ref_picture->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture->display_mode_normal
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

```

*Continued on next page*

```

WHEN 'NORMAL_CENTER'.      " picture operation: center normal size
CALL METHOD ref_picture->set_display_mode
  EXPORTING
    display_mode = cl_gui_picture->display_mode_normal_center
  EXCEPTIONS
    OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'FIT'.                  " picture operation: zoom picture
CALL METHOD ref_picture->set_display_mode
  EXPORTING
    display_mode = cl_gui_picture->display_mode_fit
  EXCEPTIONS
    OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'FIT_CENTER'.          " picture operation: zoom and center
CALL METHOD ref_picture->set_display_mode
  EXPORTING
    display_mode = cl_gui_picture->display_mode_fit_center
  EXCEPTIONS
    OTHERS = 1.
IF sy-subrc NE 0.
  MESSAGE s015.
ENDIF.

WHEN 'MODE_INFO'.
  MESSAGE i025 WITH ref_picture->display_mode.

ENDCASE.

ENDMODULE.        " USER_COMMAND_0100  INPUT
*
*-----*
*&      Module STATUS_0100  OUTPUT
*-----*
*      Set GUI for screen 0100
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'STATUS_NORM_0100'.
  SET TITLEBAR 'TITLE_NORM_0100'.

```

*Continued on next page*

```

ENDMODULE.                                     " STATUS_0100  OUTPUT
*&-----*
*&      Module  EXIT_COMMAND_0100  INPUT
*&-----*
*      Implementation of user commands of type 'E'.
*-----*
MODULE exit_command_0100 INPUT.
CASE ok_code.
  WHEN 'CANCEL'.           " cancel current screen processing
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *          DEFAULTOPTION = 'Y'
        textline1   = text-004
        textline2   = text-005
        titel       = text-006
        cancel_display = ' '
      IMPORTING
        answer      = l_answer.
  CASE l_answer.
    WHEN 'J'.
      PERFORM free_control_ressources.
      LEAVE TO SCREEN 0.
    WHEN 'N'.
      CLEAR ok_code.
      SET SCREEN sy-dynnrv.
  ENDCASE.

  WHEN 'EXIT'.                         " leave program
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *          DEFAULTOPTION = 'Y'
        textline1   = text-001
        textline2   = text-002
        titel       = text-003
        cancel_display = 'X'
      IMPORTING
        answer      = l_answer.
  CASE l_answer.
    WHEN 'J' OR 'N'.                  " no data to update
      PERFORM free_control_ressources.
      LEAVE PROGRAM.
    WHEN 'A'.
      CLEAR ok_code.

```

*Continued on next page*

```
      SET SCREEN sy-dynnrv.  
      ENDCASE.  
      ENDCASE.  
ENDMODULE.      " EXIT_COMMAND_0100  INPUT
```

### ***Subroutines***

```
*&-----  
*&      Form free_control_ressources  
*&-----  
*      Free all control related ressources.  
*-----  
*      no interface  
*-----  
FORM free_control_ressources.  
  CALL METHOD ref_picture->free.  
  CALL METHOD ref_container->free.  
  FREE: ref_picture, ref_container.  
ENDFORM.          " free_control_ressources
```



## Lesson Summary

You should now be able to:

- Describe the events triggered by the EnjoySAP Controls and learn the steps necessary to react to them



## Unit Summary

You should now be able to:

- Describe the Controll Framework architecture
- Work with controls and screens
- Explain the relationship between screen, container and EnjoySAP control
- Display data in a SAP Picture control and SAP HTML Viewer on a screen
- Understand and use the attributes of a control
- Understand how the actions are transferred to the presentation server
- Describe the Automation Queue and its Runtime Performance
- Describe the events triggered by the EnjoySAP Controls and learn the steps necessary to react to them

# Unit 3

## SAP Container

### Unit Overview

An EnjoySAP Control can be visualized only while belonging to a Container Control. This unit presents necessary basic information for all Container Control types, information valid for all flowing Controls.



### Unit Objectives

After completing this unit, you will be able to:

- Use Container Controls
- Present the inheritance hierarchy of the Container Classes
- Describe the graphical visualization of Containers
- Describe the Custom Container features
- Create an instance of the SAP Custom Container Control
- Relink an SAP Custom Container
- Describe the Dialog Box Container features
- Create an instance of the SAP Dialog Box Container Control
- Close the SAP Dialog Box Container
- Describe the Docking Container features
- Attach the SAP Docking Container to the screen
- Create an instance of the SAP Docking Container Control
- Read and Set the attributes
- Relink the SAP Docking Container
- Describe the Splitter Container features
- Create an instance of the SAP Splitter Container Control
- Use the SAP Splitter Container
- Find a cell reference
- Read and Set attributes
- Describe the Easy Splitter Container features
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container

## Unit Contents

Lesson: The Container Concept .....	121
Lesson: SAP Custom Container Control .....	126
Lesson: SAP Dialog Box Container Control.....	130
Lesson: SAP Docking Container Control .....	134
Exercise 5: SAP Docking Container .....	139
Exercise 6: (Optional) Changing the Docking Container .....	151
Lesson: SAP Splitter Container Control .....	159
Exercise 7: SAP Splitter Container Control .....	165
Lesson: SAP Easy Splitter Container Control.....	190

# Lesson: The Container Concept

## Lesson Overview

The Lesson presents the usage of a Container Control, the inheritance hierarchy of Container Classes and graphical visualization of Containers.



## Lesson Objectives

After completing this lesson, you will be able to:

- Use Container Controls
- Present the inheritance hierarchy of the Container Classes
- Describe the graphical visualization of Containers

## Business Example

You need to implement an application using the EnjoySAP Controls. Every control will be visualized in a Container and you need to understand the basics about Containers.

## Using Container Controls

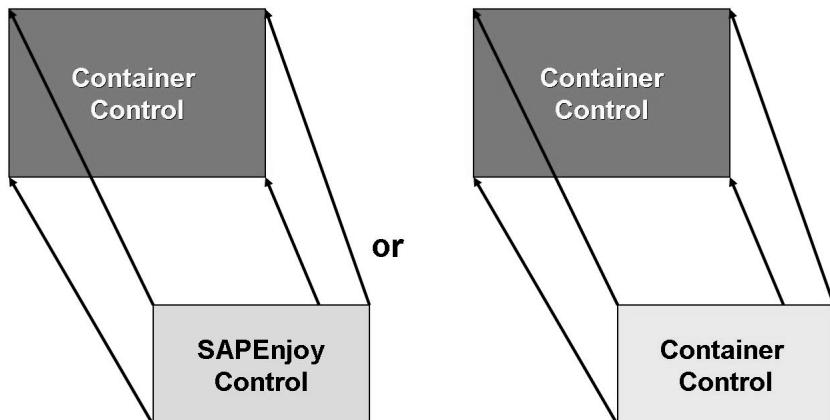


Figure 66: Using Container Controls

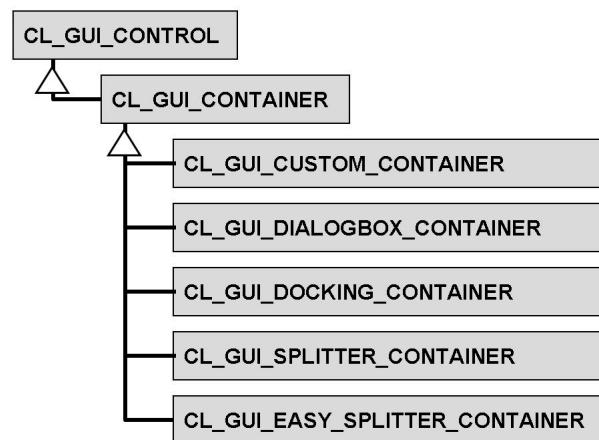
A SAP container is a control that can accommodate other controls, such as the SAP Tree Control, SAP Picture Control, or the SAP Text Edit Control. It maintains the controls in a logical collection, and provides an area in which they can be displayed.

A container is always assigned to a screen and takes over the role of communicating with the screen.

Each control lives in a container. Since containers are themselves controls, you can nest them. But the splitter containers **only** can be nested in other containers and they always need to be part of a container. Splitter containers can not be attached directly to a screen.

Controls within a container are usually displayed in the same size as the container itself.

## The Inheritance Hierarchy of Container Classes

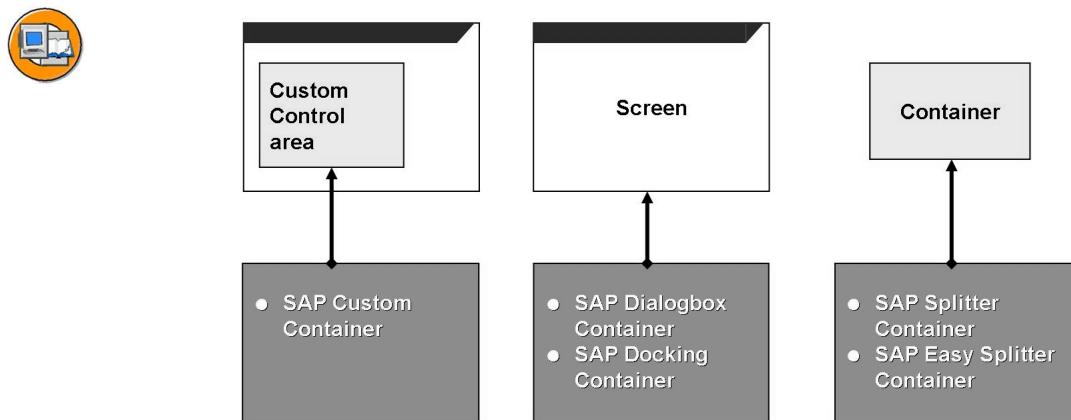


**Figure 67: The Inheritance Hierarchy of Container Classes**

Containers are known as the “**parents**” of the control that they contain.

All SAP Containers are derived from a common superclass, the global class CL\_GUI\_CONTROL. They are derived from this basic container and thus have a uniform object-oriented interface.

# Graphical Visualization of Containers



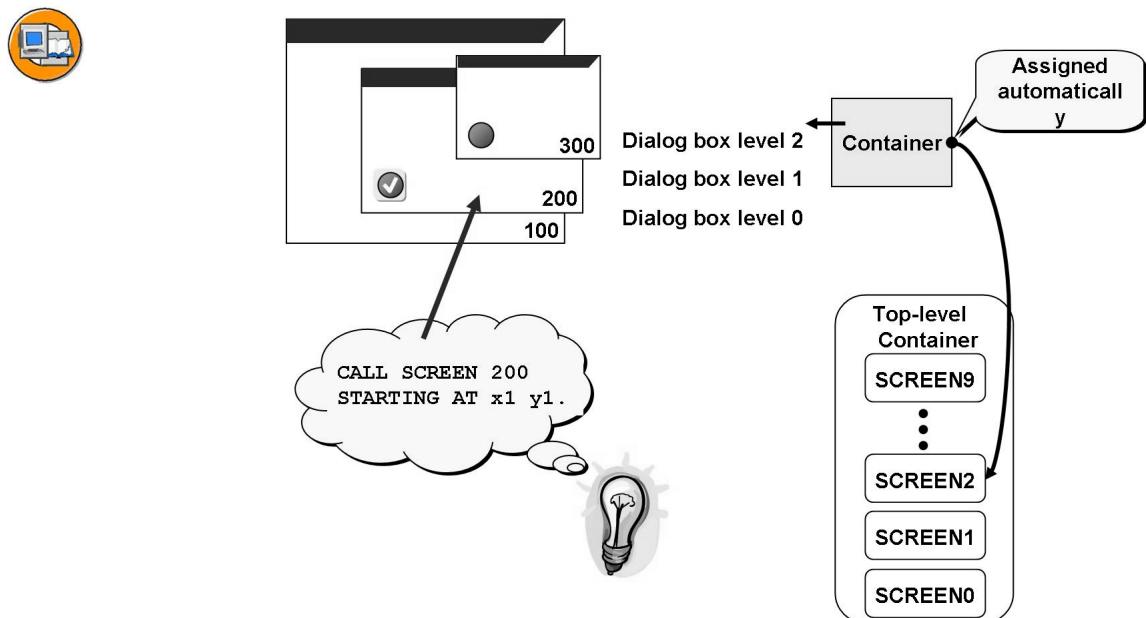
**Figure 68: Graphical Visualization of Containers**

SAP containers can contain:

- An area on a screen (SAP Custom Container)
  - An entire screen (SAP Docking Container, SAP Dialog Box Container)
  - Another container (SAP Splitter Container, SAP Easy Splitter Container)

The fact that you can nest controls provides further display possibilities.

## Dialog Box Level and the Top-Level Container



**Figure 69: Dialog Box Level and the Top-Level Container**

When you create an instance of a SAP container, you assign it to a dialog box level, which you cannot subsequently change.

A dialog box level is created in ABAP by executing the statement CALL SCREEN <screen\_no> STARTING AT <left\_column\_no> <top\_row\_no> [ENDING AT <right\_column\_no> <bottom\_row\_number>].

When you instantiate a container, the class constructor creates the container instances SCREEN0, SCREEN1, ..., SCREEN9. These are assigned to the different dialog box levels. Dialog box level 0 = SCREEN0, level 1 = SCREEN1, ... Dialog box level 9 = SCREEN9.

The container instances SCREEN0, ..., SCREEN9 are called **top-level containers**. They are used as “parents” for the SAP Custom Container, SAP Docking Container, and SAP Dialog Box Container. The top-level container is automatically assigned to the Container Control when it is instantiated. You **cannot** change this assignment at runtime.

The top-level container to which a container is assigned affects its visibility, and hence that of the other controls you display in it.

Controls in a container are only visible when the corresponding container is visible.

You need to know what the dialog box level has been assigned if you want to assign a container to a new screen (using the LINK method) or if a container is instantiated within a function module.



## Lesson Summary

You should now be able to:

- Use Container Controls
- Present the inheritance hierarchy of the Container Classes
- Describe the graphical visualization of Containers

# Lesson: SAP Custom Container Control

## Lesson Overview

This lesson presents the Custom Container Control.



## Lesson Objectives

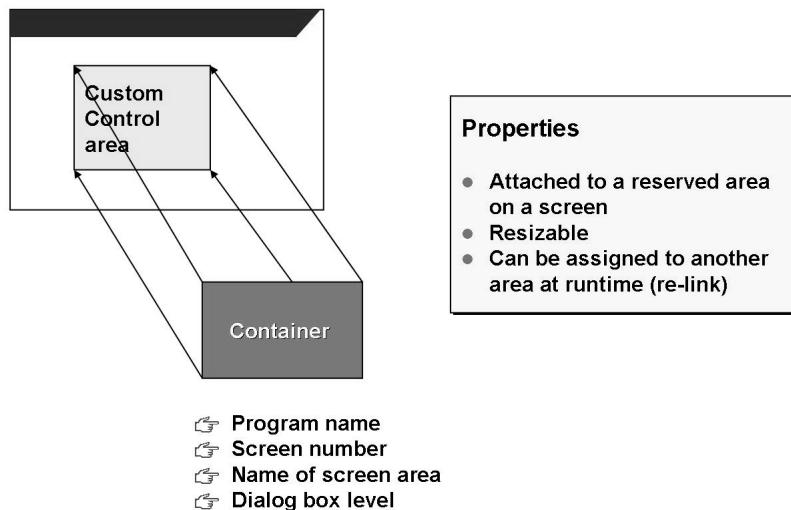
After completing this lesson, you will be able to:

- Describe the Custom Container features
- Create an instance of the SAP Custom Container Control
- Relink an SAP Custom Container

## Business Example

The application to be implemented needs a Custom Container to present an SAP Control. You need to know what is involved when using this Container type.

## SAP Custom Container: Features



**Figure 70: SAP Custom Container: Features**

Use the SAP Custom Container Control to attach a control to a reserved area on a screen.

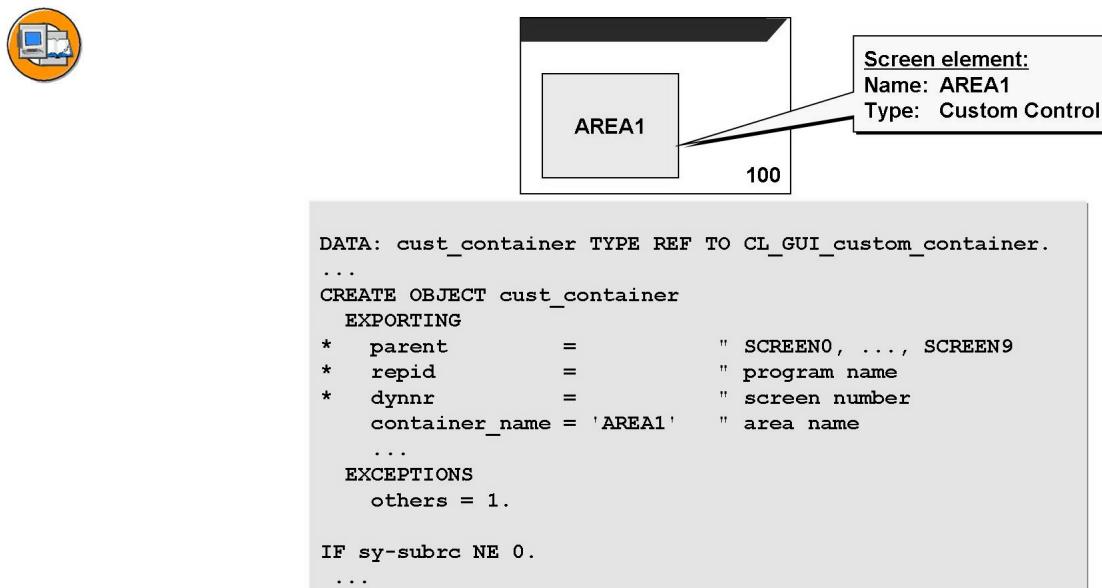
Create the area using the Custom Control element in the Screen Painter. The area can be resized.

You can assign a Custom Container Control instance to the area. This assigns the container to the screen. The container can be resized if the user changes the size of the SAP window.

The following parameters identify a custom control area uniquely at the CFW:

- Program name
- Screen number
- Name of the area
- Number of the dialog box level at which the container and screen can be displayed.

## Creating an Instance of the SAP Custom Container Control



**Figure 71: Creating an Instance of the SAP Custom Container Control**

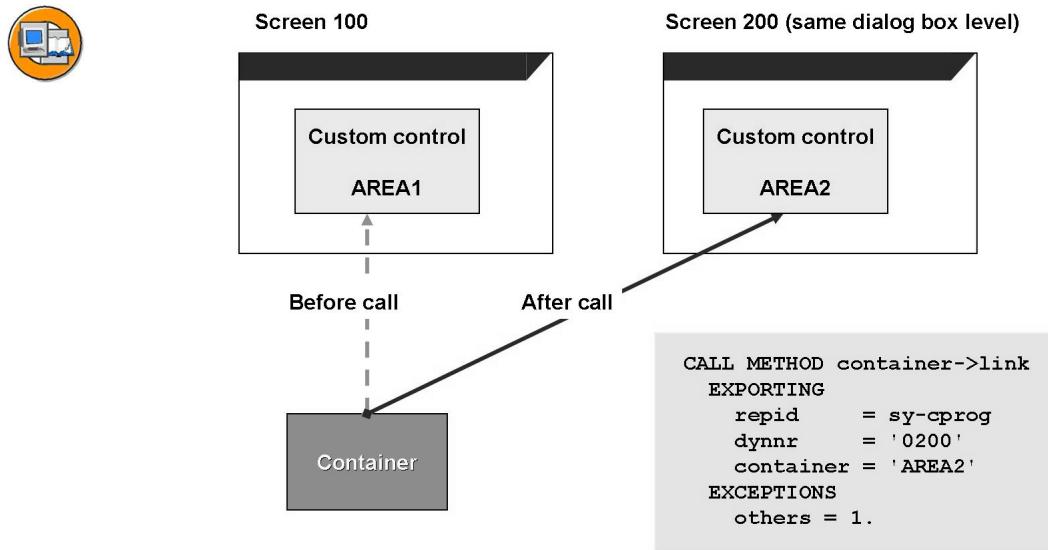
To create an instance of the SAP Custom Container Control, you need a data object that you declare using TYPE REF TO CL\_GUI\_CUSTOM\_CONTAINER.

To create the instance itself, use the statement CREATE OBJECT <object\_reference\_var>. In the statement, you must pass the relevant parameters to specify the attributes of the container. This is illustrated in the graphic. If you do not assign values to the parameters parent, repid, and dynnr, the system uses the current values at runtime (current dialog box level , current program name, current screen).

You must specify the name of the screen area in which the container should appear in the parameter container\_name.

For details of the other interface parameters in the constructor, refer to the online documentation.

## Relinking an SAP Custom Container



**Figure 72: Relinking an SAP Custom Container**

At runtime, you can assign a custom container instance to a different screen area (with the type Custom Control). The new area can be on the same screen, on another screen in the same program, or even on a screen in a different program.

The screen containing the new area to which you want to link the control can only be displayed in the same dialog box level.

To reassign an instance, you call the method LINK for the container instance you want to reassign. The method links the container instance to the new screen area and dissolves the link to the old one.



## Lesson Summary

You should now be able to:

- Describe the Custom Container features
- Create an instance of the SAP Custom Container Control
- Relink an SAP Custom Container

# Lesson: SAP Dialog Box Container Control

## Lesson Overview

This lesson presents the Dialog Box Container Control.



## Lesson Objectives

After completing this lesson, you will be able to:

- Describe the Dialog Box Container features
- Create an instance of the SAP Dialog Box Container Control
- Close the SAP Dialog Box Container

## Business Example

The application to be implemented needs a Dialog Box Container to present an SAP Control. You need to know what is involved when using this Container type.

## SAP Dialog Box Container: Features

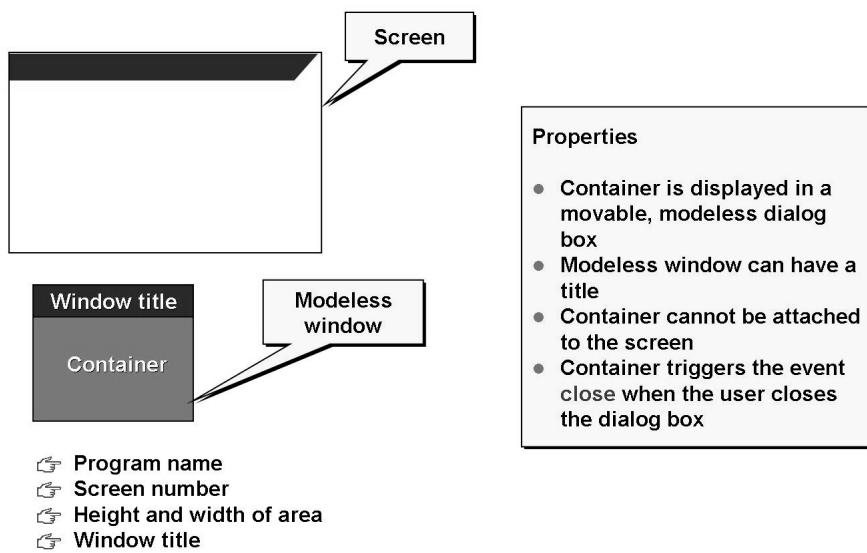


Figure 73: SAP Dialog Box Container: Features

The SAP Dialog Box Container allows you to display controls in a moveable modeless dialog box. You can also display controls in a full screen.

The modeless dialog box can have a title.

## Creating an Instance of the SAP Dialog Box Container Control



```

DATA: ref_box TYPE REF TO CL_GUI_dialogbox_container.
...
CREATE OBJECT ref_box
  EXPORTING
    * dynnr      =      " screen number
    * repid      =      " program name
    width       = 150
    height      = 150
    caption     = 'Fenstertitel'
...
  EXCEPTIONS
    others      = 1.

IF sy-subrc NE 0.
...

```

Creates a modeless dialog box  
with the title 'Window title'  
The window area provides a  
Container

**Figure 74: Creating an Instance of the SAP Dialog Box Container Control**

To create an instance of the SAP Dialog Box Container Control, you need a data object that you declare using TYPE REF TO cl\_gui\_dialogbox\_container.

To create the instance itself, use the statement CREATE OBJECT <object\_reference\_var>. In the statement, you must pass the relevant parameters to specify the attributes of the container. This is illustrated in the graphic. If you do not assign values to the REPID and DYNNR parameters, the system uses the current values at runtime (current dialog box level, current program name, current screen).

Use the WIDTH and HEIGHT parameters to specify the size of the dialog box. The user can also change the size of the dialog box at runtime.

Use the CAPTION parameter to give a title to the modeless dialog box.

You can also change the window title at runtime using the instance method SET\_CAPTION.

For details of the other interface parameters in the constructor, refer to the online documentation.

## Syntax: Closing the SAP Dialog Box Container



```
DATA: ref_box
      TYPE REF TO CL_GUI_dialogbox_container.

CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.

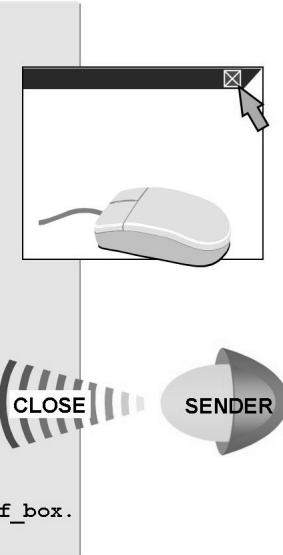
  CLASS-METHODS: on_close FOR EVENT close
    OF CL_GUI_dialogbox_container
    IMPORTING sender.

ENDCLASS.

CLASS lcl_event_handler IMPLEMENTATION.
  METHOD on_close.
    CALL METHOD sender->free.
    FREE ref_box.
  ENDMETHOD.

ENDCLASS.

CREATE OBJECT ref_box ...
SET HANDLER lcl_event_handler=>on_close FOR ref_box.
```



**Figure 75: Syntax: Closing the SAP Dialog Box Container**

The SAP Dialog Box Container Control possesses a CLOSE event. This is triggered when the user tries to close the modeless dialog box.

In the event handler method, you must close the window using the instance method free.

You can find out which instance raised the event using the optional import parameter of the event handler method.



## Lesson Summary

You should now be able to:

- Describe the Dialog Box Container features
- Create an instance of the SAP Dialog Box Container Control
- Close the SAP Dialog Box Container

# Lesson: SAP Docking Container Control

## Lesson Overview

This lesson presents the Docking Container Control.



## Lesson Objectives

After completing this lesson, you will be able to:

- Describe the Docking Container features
- Attach the SAP Docking Container to the screen
- Create an instance of the SAP Docking Container Control
- Read and Set the attributes
- Relink the SAP Docking Container

## Business Example

The application to be implemented needs a Docking Container to present an SAP Control. You need to know what is involved when using this Container type.

## SAP Docking Container: Features

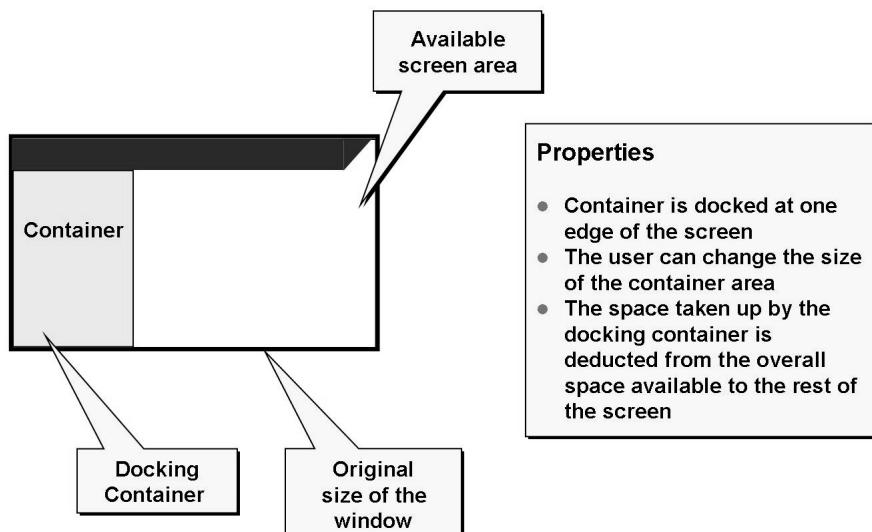
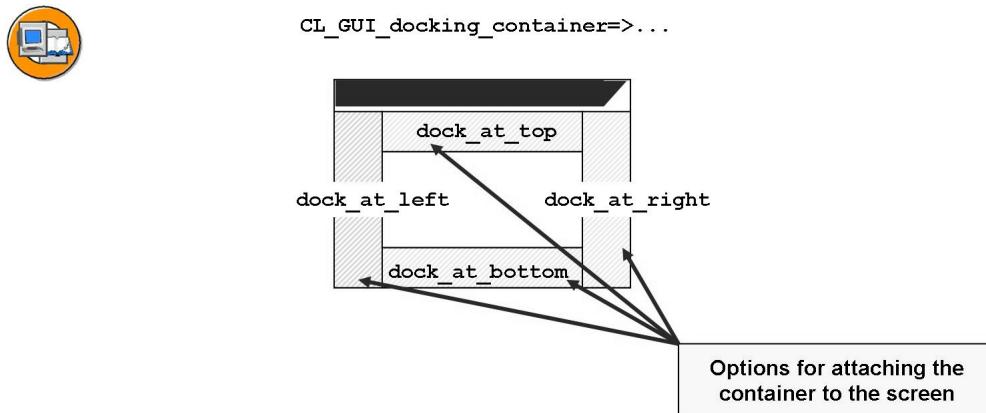


Figure 76: SAP Docking Container: Features

The SAP Docking Container allows you to attach one or more areas to a screen. You can attach the area to any of the four edges of the screen. The area available to the actual screen is reduced by the space taken up by the docking container.

The user can change the size of the docking container control.

## Attaching the SAP Docking Container to the Screen



**Figure 77: Attaching the SAP Docking Container to the Screen**

You can attach the container area of a SAP Docking Container Control to any of the four edges of the screen. To specify the edge of the screen technically, use one of the four class constants of the CL\_GUI.Docking.Container class.

Generally, you assign the area to one edge of the screen in the constructor (using the SIDE parameter). By default the container is docked on the **left**.

You can change this assignment at runtime by using the DOCK\_AT method. For further details, refer to the online documentation.



## Creating an Instance of the SAP Docking Container Control

```
DATA: dock_container TYPE REF TO CL_GUI_docking_container.  
....  
CREATE OBJECT dock_container  
    EXPORTING  
        ...  
        *      repid      =      " program name"  
        *      dynnr      =      " screen number"  
        side      = CL_GUI_docking_container->dock_at_right  
        extension = 50    " control extension"  
        ...  
        *      ratio      =      " percentage of screen"  
        ...  
    EXCEPTIONS  
        others = 1.  
  
IF sy-subrc NE 0.  
....
```

Creates a container control at the right edge of the screen

Figure 78: Creating an Instance of the SAP Docking Container Control

To create an instance of the SAP Docking Container Control, you need a data object that you declare using TYPE REF TO cl\_gui\_docking\_container.

To create the instance itself, use the statement CREATE OBJECT <object\_reference\_var>. In the statement, you must pass the relevant parameters to specify the attributes of the container. This is illustrated in the graphic. If you do not assign values to the REPID and DYNNR parameters, the system uses the current values at runtime (current dialog box level, current program name, current screen).

Use the SIDE parameter to determine the side of the screen to which the container will be attached.

For details of the other interface parameters in the constructor, refer to the online documentation.

## Reading and Setting Attributes

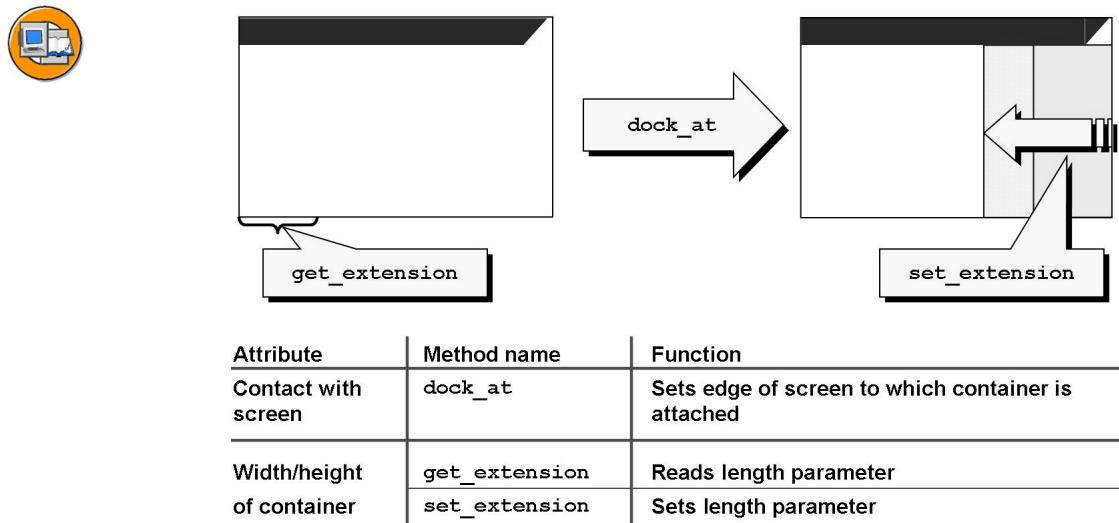


Figure 79: Reading and Setting Attributes

The methods shown above allow you to find out or set the values of attributes of your container control.

For more details on the exact functions and interfaces of the methods, see the online documentation.

## Relinking an SAP Docking Container

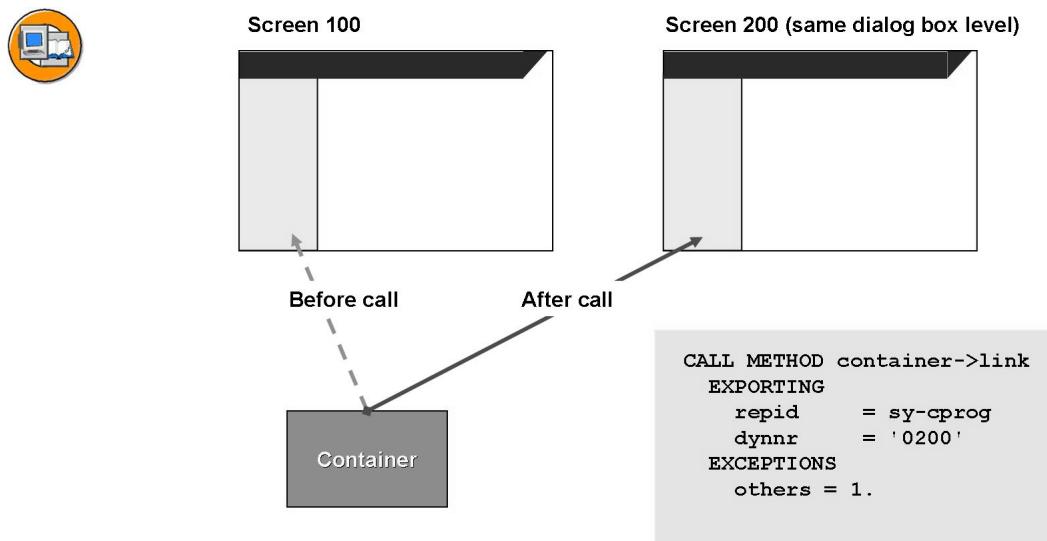


Figure 80: Relinking an SAP Docking Container

At runtime, you can dock a container to another screen at the same dialog box level (re-link it). To do this, use the instance method LINK.

In the interface, you need to specify the new screen number and the name of the program to which it belongs. If you do not set a new program name, the system assumes that the new screen belongs to the same program as the old one.

# Exercise 5: SAP Docking Container

## Exercise Objectives

After completing this exercise, you will be able to:

- Use instances of the Docking Container Control

## Business Example

You copy a template consisting of an ABAP program that calls a screen. At present, the program displays an image in a picture control on the screen using a Custom Container Control instance. Use a Docking Container control instead of a Custom Container Control.

**Program:** ZBC412\_##\_CON\_EX1

**Template:** SAPBC412\_CONT\_EXERCISE1

**Model solution:** SAPBC412\_CONS\_EXERCISE1

where ## is the group number.

### Task 1:

Copy the Template

1. Copy the program template SAPBC412\_CONT\_EXERCISE1 with all its sub-objects to **ZBC412\_##\_CON\_EX1** or use your solution from the last exercise in the previous chapter, “Introduction to the Control Framework”. This copy template has the same features as the model solution to this last exercise in “Introduction to the Control Framework”.

### Task 2:

Change your program so that it uses an SAP Docking Container instead of the SAP Custom Container.

1. Delete the Custom Control area from screen 100. You will not be using it again.
2. Change the type of your container reference variable (`ref_container`) to `cl_gui_docking_container`.



**Hint:** Use the class constant `dock_at_left` of the class `cl_gui_docking_container` to fill the interface parameter `side` in the constructor.

*Continued on next page*

3. In the `init_control_processing` module, create an instance of the SAP Docking Container Control. Attach the Docking Container instance to the left side of your screen.

**Task 3:**

Test your program.

## Solution 5: SAP Docking Container

### Task 1:

Copy the Template

1. Copy the program template SAPBC412\_CONT\_EXERCISE1 with all its sub-objects to **ZBC412\_##\_CON\_EX1** or use your solution from the last exercise in the previous chapter, “Introduction to the Control Framework”. This copy template has the same features as the model solution to this last exercise in “Introduction to the Control Framework”.
  - a) –

### Task 2:

Change your program so that it uses an SAP Docking Container instead of the SAP Custom Container.

1. Delete the Custom Control area from screen 100. You will not be using it again.
  - a) –
2. Change the type of your container reference variable (`ref_container`) to `cl_gui_docking_container`.



**Hint:** Use the class constant `dock_at_left` of the class `cl_gui_docking_container` to fill the interface parameter `side` in the constructor.

a)

```
DATA:  
...  
    ref_container TYPE REF TO cl_gui_docking_container,
```

*Continued on next page*

3. In the `init_control_processing` module, create an instance of the SAP Docking Container Control. Attach the Docking Container instance to the left side of your screen.

a)

```

IF ref_container IS INITIAL.      " prevent re-processing on ENTER
*      create container object and link to screen (left screen side)
CREATE OBJECT ref_container
EXPORTING
    side = cl_gui_docking_container->dock_at_left
EXCEPTIONS
    others = 1.

IF sy-subrc NE 0.
    MESSAGE a010.                  " cancel program processing
ENDIF.

```

## Task 3:

Test your program.

### Result

#### Screen flow logic

#### *SCREEN 100*

PROCESS BEFORE OUTPUT.

```

MODULE status_0100.
MODULE init_control_processing.

```

PROCESS AFTER INPUT.

```

MODULE exit_command_0100 AT EXIT-COMMAND.
MODULE user_command_0100.

```

### ABAP Program

#### *Local Classes*

```

REPORT sapbc412_cons_exercise1 MESSAGE-ID bc412.
*-----*
*      CLASS lcl_event_handler DEFINITION

```

*Continued on next page*

```

*-----*
*      Definition of a local class containing event handler      *
*      methods for picture control objects                      *
*-----*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      on_picture_click
        FOR EVENT picture_click OF cl_gui_picture
        IMPORTING mouse_pos_x mouse_pos_y,
*-----*
      on_control_dblclick
        FOR EVENT control_dblclick OF cl_gui_picture
        IMPORTING mouse_pos_x mouse_pos_y.
  ENDCLASS.

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
*      Corresponding class implementation                      *
*-----*
CLASS lcl_event_handler IMPLEMENTATION.
  METHOD on_picture_click.
    MESSAGE i016 WITH mouse_pos_x mouse_pos_y.
  ENDMETHOD.

  METHOD on_control_dblclick.
    MESSAGE i017 WITH mouse_pos_x mouse_pos_y.
  ENDMETHOD.
ENDCLASS.

```

### ***Data Declarations***

```

* global types: Type Pools
TYPE-POOLS: cntl.

* data types
TYPES:
  t_url          TYPE bapiuri-uri.

* data declarations
* screen specific
DATA:
  ok_code        TYPE sy-ucomm,           " command field
  copy_ok_code  LIKE ok_code,            " copy of ok_code
  l_answer       TYPE c,                 " return flag (used in

```

*Continued on next page*

```

          " standard user dialogs)

*   control specific: object references
ref_container TYPE REF TO cl_gui_docking_container,
ref_picture    TYPE REF TO cl_gui_picture,

*   control specific: auxiliary fields
l_url      TYPE t_url,           " URL of picture to be shown
current_mode LIKE cl_gui_picture=>display_mode,

*   event handling
it_events    TYPE cntl_simple_events, " internal (event) table
wa_events    LIKE LINE OF it_events. " work area

```

### Main ABAP Program: Event Blocks

```

* start of main program
START-OF-SELECTION.

      CALL FUNCTION 'BC412_BDS_GET_PIC_URL' " fetch URL of first picture
*      EXPORTING                                " from BDS
*          NUMBER        = 1
      IMPORTING
          url          = l_url
      EXCEPTIONS
          OTHERS        = 1.

      IF sy-subrc <> 0.                      " no picture --> end of program
         LEAVE PROGRAM.
      ENDIF.

      CALL SCREEN 100.                         " container screen for SAP-Enjoy
                                                " controls

* end of main program

```

### Modules

```

*&-----*
*&     Module INIT_CONTROL_PROCESSING OUTPUT
*&-----*
*     Implementation of EnjoySAP control processing:
*     1)  create container object and link to screen (left side)

```

*Continued on next page*

```

*      2) create picture object and link to container
*      3) load picture into picture control
*      4) event handling
*         a1) register event PICTURE_CLICK at CFW
*         a2) register event CONTROL_DBCLICK at CFW
*         b1) register event handler method ON_PICTURE_CLICK
*              for the picture control object
*         b2) register event handler method ON_CONTROL_DBCLICK
*              for the picture control object
*-----*
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.          " prevent re-processing on ENTER
*   create container object and link to screen (left screen side)
  CREATE OBJECT ref_container
    EXPORTING
      side = cl_gui_docking_container->dock_at_left
    EXCEPTIONS
      others = 1.

  IF sy-subrc NE 0.
    MESSAGE a010.                      " cancel program processing
  ENDIF.

*   create picture control and link to container object
  CREATE OBJECT ref_picture
    EXPORTING
      parent = ref_container
    EXCEPTIONS
      others = 1.

  IF sy-subrc NE 0.
    MESSAGE a011.                      " cancel program processing
  ENDIF.

*   load picture into picture control
  CALL METHOD ref_picture->load_picture_from_url
    EXPORTING
      url    = l_url
    EXCEPTIONS
      OTHERS = 1.

  IF sy-subrc NE 0.
    MESSAGE a012.
  ENDIF.

```

*Continued on next page*

```

*      event handling
*      1. register events for control framework
wa_events-eventid      = cl_gui_picture->eventid_picture_click.
wa_events-appl_event = ' '.
INSERT wa_events INTO TABLE it_events.

wa_events-eventid      = cl_gui_picture->eventid_control_dblclick.
wa_events-appl_event = ' '.
INSERT wa_events INTO TABLE it_events.

CALL METHOD ref_picture->set_registered_events
EXPORTING
      events = it_events
EXCEPTIONS
      OTHERS = 1.

IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*      2. set event handler for ABAP object instance: ref_picture
SET HANDLER lcl_event_handler=>on_picture_click
      FOR ref_picture.
SET HANDLER lcl_event_handler=>on_control_dblclick
      FOR ref_picture.

ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING  OUTPUT

*-----*
*&      Module  USER_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type ' ':
*      - push buttons on screen 100
*      - GUI functions
*-----*
MODULE user_command_0100 INPUT.
copy_ok_code = ok_code.
CLEAR ok_code.

```

*Continued on next page*

```

CASE copy_ok_code.
WHEN 'BACK'.                                " back to program, leave screen
  CLEAR l_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
    EXPORTING
      *           DEFAULTOPTION = 'Y'
      textline1     = text-004
      textline2     = text-005
      titel         = text-007
      cancel_display = ' '
    IMPORTING
      answer        = l_answer.

CASE l_answer.
WHEN 'J'.
  PERFORM free_control_ressources.
  LEAVE TO SCREEN 0.
WHEN 'N'.
  SET SCREEN sy-dynnrr.
ENDCASE.

WHEN 'STRETCH'.                            " picture operation: stretch to fit area
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_stretch
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'NORMAL'.                            " picture operation: fit to normal size
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_normal
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'NORMAL_CENTER'. " picture operation: center normal size
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_normal_center

```

*Continued on next page*

```

        EXCEPTIONS
          OTHERS = 1.
        IF sy-subrc NE 0.
          MESSAGE s015.
        ENDIF.

        WHEN 'FIT'.           " picture operation: zoom picture
        CALL METHOD ref_picture->set_display_mode
          EXPORTING
            display_mode = cl_gui_picture->display_mode_fit
        EXCEPTIONS
          OTHERS = 1.
        IF sy-subrc NE 0.
          MESSAGE s015.
        ENDIF.

        WHEN 'FIT_CENTER'.    " picture operation: zoom and center
        CALL METHOD ref_picture->set_display_mode
          EXPORTING
            display_mode = cl_gui_picture->display_mode_fit_center
        EXCEPTIONS
          OTHERS = 1.
        IF sy-subrc NE 0.
          MESSAGE s015.
        ENDIF.

        WHEN 'MODE_INFO'.
          current_mode = ref_picture->display_mode.
          MESSAGE i025 WITH current_mode.

        ENDCASE.
      ENDMODULE.           " USER_COMMAND_0100  INPUT

*-----*
*&      Module STATUS_0100  OUTPUT
*-----*
*      Set GUI for screen 0100
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'STATUS_NORM_0100'.
  SET TITLEBAR 'TITLE_NORM_0100'.
ENDMODULE.           " STATUS_0100  OUTPUT

```

*Continued on next page*

```

*&-----*
*&      Module  EXIT_COMMAND_0100  INPUT
*&-----*
*      Implementation of user commands of type 'E'.
*-----*
MODULE exit_command_0100 INPUT.
CASE ok_code.
  WHEN 'CANCEL'.           " cancel current screen processing
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *          DEFAULTOPTION = 'Y'
        textline1   = text-004
        textline2   = text-005
        titel       = text-006
        cancel_display = ' '
      IMPORTING
        answer      = l_answer.
  CASE l_answer.
    WHEN 'J'.
      PERFORM free_control_ressources.
      LEAVE TO SCREEN 0.
    WHEN 'N'.
      CLEAR ok_code.
      SET SCREEN sy-dynnrv.
  ENDCASE.

  WHEN 'EXIT'.              " leave program
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *          DEFAULTOPTION = 'Y'
        textline1   = text-001
        textline2   = text-002
        titel       = text-003
        cancel_display = 'X'
      IMPORTING
        answer      = l_answer.
  CASE l_answer.
    WHEN 'J' OR 'N'.         " no data to update
      PERFORM free_control_ressources.
      LEAVE PROGRAM.
    WHEN 'A'.
      CLEAR ok_code.

```

*Continued on next page*

```
SET SCREEN sy-dynnrv.  
ENDCASE.  
ENDCASE.  
ENDMODULE.                                     " EXIT_COMMAND_0100  INPUT
```

### ***Subroutines***

```
*-----  
*&      Form free_control_ressources  
*&-----  
*      Free all control related ressources.  
*-----  
*      no interface  
*-----  
FORM free_control_ressources.  
  CALL METHOD ref_picture->free.  
  CALL METHOD ref_container->free.  
  FREE: ref_picture, ref_container.  
ENDFORM.                                         " free_control_ressources
```

## Exercise 6: (Optional) Changing the Docking Side of a Docking Container

### Exercise Objectives

After completing this exercise, you will be able to:

- Attach instances of the SAP Docking Container Control to different sides of a screen.

### Business Example

Allow the user to choose whether the Picture Control appears on the left or right side of the screen.

**Program:** ZBC412\_##\_CON\_EX1A

**Template:** SAPBC412\_CONS\_EXERCISE1

**Model solution:** SAPBC412\_CONS\_EXERCISE1A

where ## is the group number.

#### Task 1:

Copy the Template

1. Copy your solution from the previous exercise (ZBC412\_##\_CON\_EX1) or the appropriate model solution (SAPBC412\_CONS\_EXERCISE1) with all their sub-objects to the name **ZBC412\_##\_CON\_EX1A**. Get to know how your copy of the program works.

#### Task 2:

Allow the user to choose which side of the screen the Picture Control appears on by providing a checkbox on the standard selection screen.

1. Extend your program to include a PARAMETERS statement that generates a checkbox field on a selection screen. (We suggest the name p\_side). To display this parameter as a checkbox, use the AS CHECKBOX addition. Provide a selection text (*Dock on right side of screen* or similar).
2. In the data declaration section of your program, create another variable, which you will use to evaluate the parameter from this selection screen. This variable should contain the technical name of the side of the screen where the Picture Control is to be attached. (We suggest the name docking\_side). Assign a type to the variable using LIKE cl\_gui\_docking\_container=>dock at right.

*Continued on next page*

3. Evaluate the parameter p\_side in the init\_control\_processing module before you create an instance of the container:  
If the parameter p\_side is not selected, assign the value cl\_gui\_docking\_container=>dock\_at\_left to the variable docking\_side. Otherwise, assign the value cl\_gui\_docking\_container=>dock\_at\_right.
4. Use the docking\_side variable in the interface of the constructor for the container instance.

### Task 3:

Test your program.

## Solution 6: (Optional) Changing the Docking Side of a Docking Container

### Task 1:

Copy the Template

1. Copy your solution from the previous exercise (ZBC412\_##\_CON\_EX1) or the appropriate model solution (SAPBC412\_CONS\_EXERCISE1) with all their sub-objects to the name **ZBC412\_##\_CON\_EX1A**. Get to know how your copy of the program works.
  - a) -

### Task 2:

Allow the user to choose which side of the screen the Picture Control appears on by providing a checkbox on the standard selection screen.

1. Extend your program to include a PARAMETERS statement that generates a checkbox field on a selection screen. (We suggest the name p\_side). To display this parameter as a checkbox, use the AS CHECKBOX addition. Provide a selection text (*Dock on right side of screen* or similar).

a)

```
PARAMETERS: p_side AS CHECKBOX.
```

2. In the data declaration section of your program, create another variable, which you will use to evaluate the parameter from this selection screen. This variable should contain the technical name of the side of the screen where the Picture Control is to be attached. (We suggest the name docking\_side). Assign a type to the variable using LIKE cl\_gui\_docking\_container=>dock\_at\_right.

a)

```
DATA:  
...  
docking_side  LIKE cl_gui_docking_container=>dock_at_right.
```

3. Evaluate the parameter p\_side in the init\_control\_processing module before you create an instance of the container:

*Continued on next page*

If the parameter p\_side is not selected, assign the value cl\_gui\_docking\_container=>dock\_at\_left to the variable docking\_side. Otherwise, assign the value cl\_gui\_docking\_container=>dock\_at\_right.

a)

```
IF ref_container IS INITIAL.      " prevent re-processing on ENTER
*      create container object and link to screen
IF p_side IS INITIAL.
    docking_side = cl_gui_docking_container=>dock_at_left.
ELSE.                           " dock to right side selected
    docking_side = cl_gui_docking_container=>dock_at_right.
ENDIF.
```

4. Use the docking\_side variable in the interface of the constructor for the container instance.

a)

```
...
CREATE OBJECT ref_container
EXPORTING
    side = docking_side
EXCEPTIONS
    others = 1.

IF sy-subrc NE 0.
    MESSAGE a010.          " cancel program processing
ENDIF.
...
```

## Task 3:

Test your program.

## Result

### ABAP Program

#### *Data Declarations and Selection Screen*

```
* global types
TYPE-POOLS: cntl.
* data types
TYPES:
    t_url      TYPE bapiuri-uri.
```

*Continued on next page*

```

* data declarations
*   screen specific
DATA:
  ok_code      TYPE sy-ucomm,          " command field
  copy_ok_code LIKE ok_code,          " copy of ok_code
  l_answer     TYPE c,                " return flag (used in
                                         " standard user dialogs)

*   control specific: object references
  ref_container TYPE REF TO cl_gui_docking_container,
  ref_picture    TYPE REF TO cl_gui_picture,

*   control specific: auxiliary fields
  l_url         TYPE t_url,           " URL of picture to be shown
  current_mode   LIKE cl_gui_picture=>display_mode,
  docking_side  LIKE cl_gui_docking_container=>dock_at_right,

*   event handling
  it_events      TYPE cntl_simple_events, " internal (event) table
  wa_events      LIKE LINE OF it_events.  " work area

* selection screen definition
PARAMETERS: p_side AS CHECKBOX.

```

### **Modules**

```

*&-----*
*&     Module INIT_CONTROL_PROCESSING OUTPUT
*&-----*
*     Implementation of EnjoySAP control processing:
*     1) create container object and link to screen (left side)
*     2) create picture object and link to container
*     3) load picture into picture control
*     4) event handling
*        a1) register event PICTURE_CLICK at CFW
*        a2) register event CONTROL_DBCLICK at CFW
*        b1) register event handler method ON_PICTURE_CLICK
*             for the picture control object
*        b2) register event handler method ON_CONTROL_DBCLICK
*             for the picture control object
*-----*
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.      " prevent re-processing on ENTER
*     create container object and link to screen

```

*Continued on next page*

```

IF p_side IS INITIAL.
  docking_side = cl_gui_docking_container->dock_at_left.
ELSE.                                     " dock to right side selected
  docking_side = cl_gui_docking_container->dock_at_right.
ENDIF.

CREATE OBJECT ref_container
  EXPORTING
    side = docking_side
  EXCEPTIONS
    others = 1.

IF sy-subrc NE 0.
  MESSAGE a010.                         " cancel program processing
ENDIF.

*      create picture control and link to container object
CREATE OBJECT ref_picture
  EXPORTING
    parent = ref_container
  EXCEPTIONS
    others = 1.

IF sy-subrc NE 0.
  MESSAGE a011.                         " cancel program processing
ENDIF.

*      load picture into picture control
CALL METHOD ref_picture->load_picture_from_url
  EXPORTING
    url      = l_url
  EXCEPTIONS
    OTHERS = 1.

IF sy-subrc NE 0.
  MESSAGE a012.
ENDIF.

*      event handling
*      1. register events for control framework
wa_events-eventid      = cl_gui_picture->eventid_picture_click.
wa_events-appl_event = ' '.
INSERT wa_events INTO TABLE it_events.

```

*Continued on next page*

```
wa_events-eventid      = cl_gui_picture->eventid_control dblclick.  
wa_events-appl_event = ' ' .  
INSERT wa_events INTO TABLE it_events.  
  
CALL METHOD ref_picture->set_registered_events  
      EXPORTING  
            events = it_events  
      EXCEPTIONS  
            OTHERS = 1 .  
  
IF sy-subrc NE 0 .  
MESSAGE a012 .  
ENDIF .  
*      2. set event handler for ABAP object instance: ref_picture  
SET HANDLER lcl_event_handler=>on_picture_click  
      FOR ref_picture .  
SET HANDLER lcl_event_handler=>on_control dblclick  
      FOR ref_picture .  
ENDIF .  
ENDMODULE .  
          " INIT_CONTROL_PROCESSING OUTPUT
```



## Lesson Summary

You should now be able to:

- Describe the Docking Container features
- Attach the SAP Docking Container to the screen
- Create an instance of the SAP Docking Container Control
- Read and Set the attributes
- Relink the SAP Docking Container

# Lesson: SAP Splitter Container Control

## Lesson Overview

This lesson presents the Splitter Container Control.



## Lesson Objectives

After completing this lesson, you will be able to:

- Describe the Splitter Container features
- Create an instance of the SAP Splitter Container Control
- Use the SAP Splitter Container
- Find a cell reference
- Read and Set attributes

## Business Example

The application to be implemented needs a Splitter Container to present an SAP Control. You need to know what is involved when using this Container type.

## SAP Splitter Container: Features

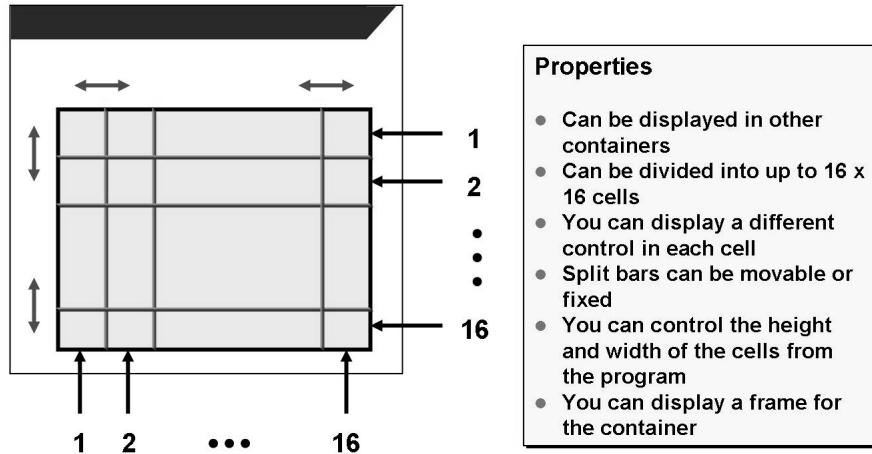


Figure 81: SAP Splitter Container: Features

The SAP Splitter Container Control allows you to display a group of controls in individual cells. The splitter container regulates the cells and displays them. The user can change the size of the individual cells using split bars. Increasing the size of one cell decreases that of the adjacent cell.

You can make the splitter bars immovable from within your program.

You can only place a Splitter Container Control in another container such as a Custom Container or a Docking Container.

You can nest splitter container instances.

The grid of a Splitter Container Control instance is initially set to 0 x 0. The maximum division is 16 x 16. You can specify the size of the lines and columns either absolutely in pixels or relatively as a percentage. The default setting is relative.

## Creating an Instance of the SAP Splitter Container Control



```

DATA: split_container TYPE REF TO CL_GUI_splitter_container,
      cust_container  TYPE REF TO CL_GUI_custom_container.
...
CREATE OBJECT split_container
  EXPORTING
    parent  = cust_container
*   repid   = "program name"
*   dynnr   = "screen number"
    rows    = 2
    columns = 2
  ...
  EXCEPTIONS
    others  = 1.

IF sy-subrc NE 0.
...

```

Parameter	Meaning
parent	Reference variable to relevant container object
rows	Number of rows in grid
columns	Number of columns in grid

Creates a splitter control instance  
with a 2 x 2 grid  
(four cells = four areas for  
other controls)  
Custom control contains  
Container Control

**Figure 82: Creating an Instance of the SAP Splitter Container Control**

To create an instance of the SAP Splitter Container Control, you need a data object that you declare using TYPE REF TO cl\_gui\_splitter\_container as well as a reference to another container instance.

To create the instance itself, use the statement CREATE OBJECT <object\_reference\_var>. In the statement, you must pass the relevant parameters to specify the attributes of the container. This is illustrated in the graphic. If you do not assign values to the REPID and DYNNR parameters, the system uses the current values at runtime (current dialog box level, current program name, current screen).

Use the rows and columns parameters to specify the number of ROWS and COLUMNS that your grid should have.

Use the PARENT parameter to assign your splitter control instance to another container.

For details of the other interface parameters in the constructor, refer to the online documentation.

## SAP Splitter Container: Use

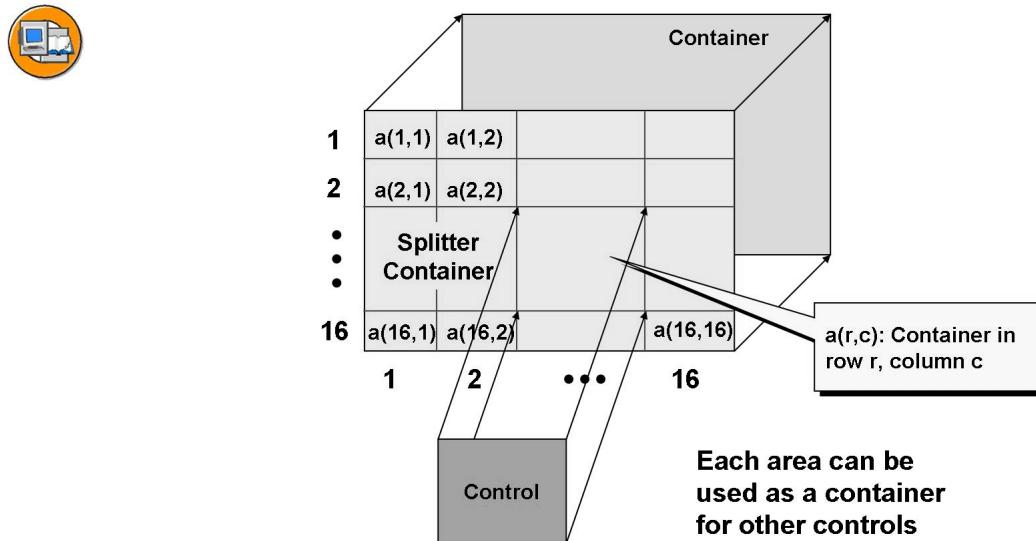


Figure 83: SAP Splitter Container: Use

You can only assign a Splitter Container instance to a screen by using another container that is directly attached to the screen. The Splitter Container can not be attached directly to the screen.

The cells of a splitter container are containers themselves. They are created when the Splitter Control is instantiated. They are used as containers for other controls, for which they are the “parents.”

## SAP Splitter Container: Finding a Cell Reference



```

DATA: split_container TYPE REF TO cl_gui_splitter_container,
      area_1_2          TYPE REF TO cl_gui_container.
      ...
CALL METHOD split_container->get_container
  EXPORTING
    row      = 1
    column   = 2
  RECEIVING
    container = area_1_2.
  ...

```

Returns the object reference to the cell (1,2) - that is, first line, second column of the SAP Splitter Container Control

Parameter	Meaning
rows	Row number in grid
columns	Column number in grid

Figure 84: SAP Splitter Container: Finding a Cell Reference

To assign further control instances to the cells of your splitter container instance, you need object references to the individual cells. To get a reference to a cell, use the instance method GET\_CONTAINER. In the method call, use the row and column parameters to specify the cell. The RETURNING parameter contains a reference to a container instance.

## Reading and Setting Attributes



Attribute	Method name	Function
Cell	add_control	Assigns a control to a cell
	remove_control	Removes a control from a cell
	get_container	Gets a container reference to a cell
Columns	get_columns	Reads the number of columns
	get_column_mode set_column_mode	Reads mode for division: absolute/relative Sets mode for division: absolute/relative
Column	get_column_width set_column_width	Reads column width Sets column width
	get_column_sash set_column_sash	Reads mode for splitter bar: fixed/movable Sets mode for splitter bar: fixed/movable
Rows	get_rows	Reads the number of lines
	get_row_mode set_row_mode	Reads mode for division: absolute/relative Sets mode for division: absolute/relative
	get_row_height set_row_height	Reads line height Sets line height
Frame	get_row_sash set_row_sash	Reads mode for splitter bar: fixed/movable Sets mode for splitter bar: fixed/movable
	set_border	Sets border display: Yes/no

Figure 85: Reading and Setting Attributes

The methods shown above allow you to find out or set the values of attributes of your container control.

For more details on the exact functions and interfaces of the methods, see the online documentation.



# Exercise 7: SAP Splitter Container Control

## Exercise Objectives

After completing this exercise, you will be able to:

- Create instances of the SAP Splitter Container Control and configure them. (That is, split them into cells.)
- Get references to the cells of your Splitter Container instance.
- Use the cell references as containers for other EnjoySAP Controls

## Business Example

Change your program so the two pictures appear next to each other on the container screen. To do this, use a SAP Splitter Control instance consisting of two cells. As in the previous exercise, allow the user to change the picture control instance of the right-hand cell using pushbuttons on the screen. Event handling (PICTURE\_CLICK and CONTROL\_DBLCICK) should be possible for both picture control instances.

**Program:** ZBC412\_##\_CON\_EX2

**Template:** SAPBC412\_CONT\_EXERCISE1

**Model solution:** SAPBC412\_CONS\_EXERCISE2

where ## is the group number.

### Task 1:

Copy the program template SAPBC412\_CONT\_EXERCISE1 with all its subobjects to ZBC412\_##\_CON\_EX2 or use your solution from the last exercise in the previous chapter, “Introduction to the Control Framework”. This copy template has the same features as the model solution to this last exercise in Introduction to the Control Framework.

### Task 2:

You will need several additional data objects, since you are attaching splitter control instances to another container and since the splitter container consists of several cells that can accommodate other control instances. In the declaration part of your program, create the following data objects:

1. A reference variable for a splitter container instance. (We suggest the name `ref_splitter`). Assign it the type `c1_gui_splitter_container`.

*Continued on next page*

2. A reference variable for a second picture control instance. (We suggest the name `ref_pic_left`). Assign it the type `cl_gui_picture`. To make your source code more readable, rename the reference to the first picture control object. (We suggest the name `ref_pic_right`).
3. Two reference variables for the two cells of your splitter container instance. (We suggest the names `cell_1_1` and `cell_1_2`). Assign them the type `cl_gui_container`.
4. A second variable that contains the URL to the second image you want to display (suggested name `l_url2`, to which you assign the ABAP Dictionary type `BABIURI_URI`). To make your source code more readable, rename the variable `l_url` to `l_url1`.

### Task 3:

You need a second image for your second picture control instance (although you could use the same image in both). Again, use the images stored in the Business Document Service for this course.

1. Use the function module `BC412_BDS_GET_PIC_URL` to get a valid URL to an image in the BDS. Assign the value 2 to the `IMPORT` parameter number of the function module. Store the URL to the image in the data object `l_url2`. Again, this URL is valid for the duration of the program.

### Task 4:

Extend the PBO module `init_control_processing` as follows:

1. Create an instance of a Splitter Container that you attach to your Custom Control instance. This Splitter Container instance should have two cells, split horizontally (one row, two columns). If the constructor raises an exception, make your program react by displaying the termination (type A) message 010 (message class `bc412`). **Make sure** that you have already instantiated the Custom Container object.
2. Get the references (pointers) to the cells of your Splitter Container instance. To do this, use the instance method `get_container` of the class `cl_gui_splitter_container`.



**Hint:** You do not need to include any error handling since this method does not raise any exceptions.

3. Generate the two Picture Control instances and attach them to the two cells of your Splitter Container instance using the cell references you have just obtained. If the constructor raises an exception, make your program reacts by displaying the termination (type A) message 011 (message class `bc412`).

*Continued on next page*

4. Load the two images into your Picture Control instances using the URL variables `l_url1` and `l_url2`. Use the dialog messages from the previous exercises to react to any exceptions.
5. Register the two events `PICTURE_CLICK` and `CONTROL_DBCLICK` in the Control Framework for the two Picture Control instances. To do this, use the `set_registered_events` method for both instances. Use message 012 from message class BC412 (message type A) to react to any exceptions.
6. Register the static handler methods `on_picture_click` and `on_control_dbclick` of your local class `lcl_event_handler` for both Picture Control instances.

### Task 5:

In the PAI module `user_command_0100`, change the implementations of the `STRETCH`, `NORMAL`, `NORMAL_CENTER`, `FIT`, `FIT_CENTER`, and `MODE_INFO` functions so that the methods listed there for the Picture Control are performed on the right cell of the Splitter Container. To do this, change the name of the Picture Control reference variable to `ref_pic_right` in all implementations.

### Task 6:

Change the subroutine `free_control_ressources` so that all the control instances created on the presentation server are destroyed correctly.



**Hint:** You must perform the instance method `free` on each instance in reverse order to the order in which they were instantiated

1. –

### Task 7:

Test your program.

## Solution 7: SAP Splitter Container Control

### Task 1:

Copy the program template SAPBC412\_CONT\_EXERCISE1 with all its subobjects to ZBC412\_##\_CON\_EX2 or use your solution from the last exercise in the previous chapter, “Introduction to the Control Framework”. This copy template has the same features as the model solution to this last exercise in Introduction to the Control Framework.

### Task 2:

You will need several additional data objects, since you are attaching splitter control instances to another container and since the splitter container consists of several cells that can accommodate other control instances. In the declaration part of your program, create the following data objects:

1. A reference variable for a splitter container instance. (We suggest the name `ref_splitter`). Assign it the type `cl_gui_splitter_container`.
  - a)

```
DATA:  
...  
    ref_splitter    TYPE REF TO cl_gui_splitter_container.
```

2. A reference variable for a second picture control instance. (We suggest the name `ref_pic_left`). Assign it the type `cl_gui_picture`. To make your source code more readable, rename the reference to the first picture control object. (We suggest the name `ref_pic_right`).

- a)

```
DATA:  
...  
    ref_pic_left    TYPE REF TO cl_gui_picture,  
    ref_pic_right   TYPE REF TO cl_gui_picture.
```

*Continued on next page*

3. Two reference variables for the two cells of your splitter container instance. (We suggest the names `cell_1_1` and `cell_1_2`). Assign them the type `cl_gui_container`.

a)

```
DATA:  
...  
    cell_1_1      TYPE REF TO cl_gui_container,  
    cell_1_2      TYPE REF TO cl_gui_container.  
...
```

4. A second variable that contains the URL to the second image you want to display (suggested name `l_url2`, to which you assign the ABAP Dictionary type `BABIURI_URI`). To make your source code more readable, rename the variable `l_url` to `l_url1`.

a)

```
DATA:  
...  
    l_url1      TYPE t_url,          " URL of picture 1 to be shown  
    l_url2      TYPE t_url.         " URL of picture 2 to be shown  
...
```

*Continued on next page*

### Task 3:

You need a second image for your second picture control instance (although you could use the same image in both). Again, use the images stored in the Business Document Service for this course.

1. Use the function module BC412\_BDS\_GET\_PIC\_URL to get a valid URL to an image in the BDS. Assign the value 2 to the IMPORT parameter number of the function module. Store the URL to the image in the data object l\_url2. Again, this URL is valid for the duration of the program.
  - a)

```
CALL FUNCTION 'BC412_BDS_GET_PIC_URL' fetch URL of second picture
      EXPORTING
          number           = 2
      IMPORTING
          url             = l_url2
      EXCEPTIONS
          OTHERS          = 1.

      IF sy-subrc <> 0.                      " no picture --> end of program
          LEAVE PROGRAM.
      ENDIF.
```

### Task 4:

Extend the PBO module init\_control\_processing as follows:

1. Create an instance of a Splitter Container that you attach to your Custom Control instance. This Splitter Container instance should have two cells, split horizontally (one row, two columns). If the constructor raises an exception,

*Continued on next page*

make your program react by displaying the termination (type A) message 010 (message class bc412). **Make sure** that you have already instantiated the Custom Container object.

a)

```
CREATE OBJECT ref_splitter
EXPORTING
  parent  = ref_container
  rows    = 1
  columns = 2
EXCEPTIONS
  others  = 1.

IF sy-subrc NE 0.
  MESSAGE a010.                      " cancel program processing
ENDIF.
```

*Continued on next page*

2. Get the references (pointers) to the cells of your Splitter Container instance. To do this, use the instance method `get_container` of the class `cl_gui_splitter_container`.



**Hint:** You do not need to include any error handling since this method does not raise any exceptions.

a)

```
*      left cell
CALL METHOD ref_splitter->get_container
EXPORTING
      row      = 1
      column   = 1
RECEIVING
      container = cell_1_1.
*      alternatively possible: (functional method)
*      cell_1_1 = ref_splitter->get_container( row = '1'
*                                         column = '1' ).
*
*      right cell
CALL METHOD ref_splitter->get_container
EXPORTING
      row      = 1
      column   = 2
RECEIVING
      container = cell_1_2.
```

*Continued on next page*

3. Generate the two Picture Control instances and attach them to the two cells of your Splitter Container instance using the cell references you have just obtained. If the constructor raises an exception, make your program reacts by displaying the termination (type A) message 011 (message class bc412).

a)

```
*      create picture control objects and link to cell pointer
*      picture control object 1 (left cell)
CREATE OBJECT ref_pic_left
EXPORTING
  parent = cell_1_1
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
  MESSAGE a011.                               " cancel program processing
ENDIF.

*      picture control object 2 (right cell)
CREATE OBJECT ref_pic_right
EXPORTING
  parent = cell_1_2
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
  MESSAGE a011.                               " cancel program processing
ENDIF.
```

*Continued on next page*

4. Load the two images into your Picture Control instances using the URL variables l\_url1 and l\_url2. Use the dialog messages from the previous exercises to react to any exceptions.

a)

```
*      load pictures into picture control objects
*      left cell
CALL METHOD ref_pic_left->load_picture_from_url
EXPORTING
      url      = l_url1
EXCEPTIONS
      OTHERS   = 1.

IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*      right cell
CALL METHOD ref_pic_right->load_picture_from_url
EXPORTING
      url      = l_url2
EXCEPTIONS
      OTHERS   = 1.

IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.
```

*Continued on next page*

5. Register the two events PICTURE\_CLICK and CONTROL\_DBCLICK in the Control Framework for the two Picture Control instances. To do this, use the set\_registered\_events method for both instances. Use message 012 from message class BC412 (message type A) to react to any exceptions.

a)

```
*      send event table for picture in left cell to cfw
CALL METHOD ref_pic_left->set_registered_events
      EXPORTING
            events = it_events
      EXCEPTIONS
            OTHERS = 1.

      IF sy-subrc NE 0.
      MESSAGE a012.
      ENDIF.

*      send event table for picture in right cell to cfw
CALL METHOD ref_pic_right->set_registered_events
      EXPORTING
            events = it_events
      EXCEPTIONS
            OTHERS = 1.

      IF sy-subrc NE 0.
      MESSAGE a012.
      ENDIF.
```

*Continued on next page*

6. Register the static handler methods `on_picture_click` and `on_control_dblclick` of your local class `lcl_event_handler` for both Picture Control instances.

a)

```
*      set event handler for ABAP object instance:  
*      ref_pic_left (left cell)  
SET HANDLER lcl_event_handler=>on_picture_click  
FOR ref_pic_left.  
SET HANDLER lcl_event_handler=>on_control_dblclick  
FOR ref_pic_left.  
  
*      ref_pic_right (right cell)  
SET HANDLER lcl_event_handler=>on_picture_click  
FOR ref_pic_right.  
SET HANDLER lcl_event_handler=>on_control_dblclick  
FOR ref_pic_right.
```

*Continued on next page*

## Task 5:

In the PAI module `user_command_0100`, change the implementations of the `STRETCH`, `NORMAL`, `NORMAL_CENTER`, `FIT`, `FIT_CENTER`, and `MODE_INFO` functions so that the methods listed there for the Picture Control are performed on the right cell of the Splitter Container. To do this, change the name of the Picture Control reference variable to `ref_pic_right` in all implementations.

## Task 6:

Change the subroutine `free_control_ressources` so that all the control instances created on the presentation server are destroyed correctly.



**Hint:** You must perform the instance method `free` on each instance in reverse order to the order in which they were instantiated

1. –
- a)

```

FORM free_control_ressources.
  CALL METHOD ref_pic_left->free.
  CALL METHOD ref_pic_right->free.
  CALL METHOD ref_splitter->free.
  CALL METHOD ref_container->free.
  FREE: ref_pic_left,
        ref_pic_right,
        cell_1_1,
        cell_1_2,
        ref_splitter,
        ref_container.
ENDFORM.                                     " free_control_ressources

```

## Task 7:

Test your program.

### Result

#### Screen flow logic

#### SCREEN 100

```
PROCESS BEFORE OUTPUT.
```

```
MODULE status_0100.
```

*Continued on next page*

```

MODULE init_control_processing.

PROCESS AFTER INPUT.

MODULE exit_command_0100 AT EXIT-COMMAND.
MODULE user_command_0100.
```

## ABAP Program

### *Local Classes*

```

REPORT sapbc412_cons_exercise2 MESSAGE-ID bc412.
*-----
*      CLASS lcl_event_handler DEFINITION
*-----
*      Definition of a local class containing event handler      *
*      methods for picture control objects                      *
*-----
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      on_picture_click
        FOR EVENT picture_click OF cl_gui_picture
          IMPORTING mouse_pos_x mouse_pos_y,
      *-----
      on_control_dblclick
        FOR EVENT control_dblclick OF cl_gui_picture
          IMPORTING mouse_pos_x mouse_pos_y.
  ENDCLASS.

*-----
*      CLASS lcl_event_handler IMPLEMENTATION
*-----
*      Corresponding class implementation                      *
*-----
CLASS lcl_event_handler IMPLEMENTATION.
  METHOD on_picture_click.
    MESSAGE i016 WITH mouse_pos_x mouse_pos_y.
  ENDMETHOD.

  METHOD on_control_dblclick.
    MESSAGE i017 WITH mouse_pos_x mouse_pos_y.
  ENDMETHOD.
ENDCLASS.
```

*Continued on next page*

**Data Declarations**

```

* global types
TYPE-POOLS: cntl.

* data types
TYPES:
  t_url          TYPE bapiuri-uri.

* data declarations
*   screen specific
DATA:
  ok_code        TYPE sy-ucomm,           " command field
  copy_ok_code  LIKE ok_code,           " copy of ok_code
  l_answer       TYPE c,                " return flag (used in
                                         " standard user dialogs)

*   control specific: object references
  ref_container  TYPE REF TO cl_gui_custom_container,
  ref_splitter   TYPE REF TO cl_gui_splitter_container,
  ref_pic_left   TYPE REF TO cl_gui_picture,
  ref_pic_right  TYPE REF TO cl_gui_picture,

*   object references to splitter control areas
  cell_1_1       TYPE REF TO cl_gui_container,
  cell_1_2       TYPE REF TO cl_gui_container,

*   control specific: auxiliary fields
  l_url1         TYPE t_url,             " URL of picture 1 to be shown
  l_url2         TYPE t_url,             " URL of picture 2 to be shown
  current_mode   LIKE cl_gui_picture=>display_mode,

*   event handling
  it_events      TYPE cntl_simple_events, " internal (event) table
  wa_events      LIKE LINE OF it_events.  " work area

```

**Main ABAP Program: Event Blocks**

```

* start of main program
START-OF-SELECTION.

  CALL FUNCTION 'BC412_BDS_GET_PIC_URL' " fetch URL of first picture
*     EXPORTING
*       NUMBER           = 1

```

*Continued on next page*

```

IMPORTING
    url          = l_url1
EXCEPTIONS
    OTHERS       = 1.

IF sy-subrc <> 0.                      " no picture --> end of program
    LEAVE PROGRAM.
ENDIF.

CALL FUNCTION 'BC412_BDS_GET_PIC_URL' fetch URL of second picture
    EXPORTING                               " from BDS
        number      = 2
    IMPORTING
        url          = l_url2
EXCEPTIONS
    OTHERS       = 1.

IF sy-subrc <> 0.                      " no picture --> end of program
    LEAVE PROGRAM.
ENDIF.

CALL SCREEN 100.                         " container screen for SAP-Enjoy
                                         " controls

* end of main program

```

### **Modules**

```

*&-----*
*&     Module INIT_CONTROL_PROCESSING OUTPUT
*&-----*
*     Implementation of EnjoySAP control processing:
*     1) create custom container object and link to screen area
*     2) create splitter container object and link to the custom
*        control object
*     3) fetch pointer to splitter control areas
*     4) create two picture objects and link to splitter control
*        areas
*     5) load pictures into picture control objects
*     6) event handling (for the picture on the left side only)
*        a1) register event PICTURE_CLICK at CFW
*        a2) register event CONTROL_DBLCCLICK at CFW
*        b1) register event handler method ON_PICTURE_CLICK
*            for both picture control objects

```

*Continued on next page*

```

*           b2) register event handler method ON_CONTROL_DBLCCLICK
*                 for both picture control objects
*-----
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.      " prevent re-processing on ENTER
*     create custom container object and link to screen area
  CREATE OBJECT ref_container
    EXPORTING
      container_name = 'CONTROL_AREA1'
    EXCEPTIONS
      others = 1.

  IF sy-subrc NE 0.
    MESSAGE a010.                      " cancel program processing
  ENDIF.

*     create splitter container object and link to custom control
  CREATE OBJECT ref_splitter
    EXPORTING
      parent   = ref_container
      rows     = 1
      columns  = 2
    EXCEPTIONS
      others = 1.

  IF sy-subrc NE 0.
    MESSAGE a010.                      " cancel program processing
  ENDIF.

*     fetch pointer to splitter control areas
*     left cell
  CALL METHOD ref_splitter->get_container
    EXPORTING
      row       = 1
      column    = 1
    RECEIVING
      container = cell_1_1.
*     alternatively possible: (functional method)
*     cell_1_1 = ref_splitter->get_container( row = '1'
*                                           column = '1' ).
*
*     right cell
  CALL METHOD ref_splitter->get_container
    EXPORTING
      row       = 1

```

*Continued on next page*

```

        column      = 2
RECEIVING
        container = cell_1_2.

*      create picture control objects and link to cell pointer
*      picture control object 1 (left cell)
CREATE OBJECT ref_pic_left
EXPORTING
        parent = cell_1_1
EXCEPTIONS
        others = 1.

IF sy-subrc NE 0.
MESSAGE a011.                               " cancel program processing
ENDIF.

*      picture control object 2 (right cell)
CREATE OBJECT ref_pic_right
EXPORTING
        parent = cell_1_2
EXCEPTIONS
        others = 1.

IF sy-subrc NE 0.
MESSAGE a011.                               " cancel program processing
ENDIF.

*      load pictures into picture control objects
*      left cell
CALL METHOD ref_pic_left->load_picture_from_url
EXPORTING
        url      = l_url1
EXCEPTIONS
        OTHERS = 1.

IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*      right cell
CALL METHOD ref_pic_right->load_picture_from_url
EXPORTING
        url      = l_url2
EXCEPTIONS

```

*Continued on next page*

```

OTHERS = 1.

IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*      event handling
*      1. register events for control framework
wa_events-eventid    = cl_gui_picture->eventid_picture_click.
wa_events-appl_event = ''.
INSERT wa_events INTO TABLE it_events.

wa_events-eventid    = cl_gui_picture->eventid_control_dblclick.
wa_events-appl_event = ''.
INSERT wa_events INTO TABLE it_events.

*      send event table for picture in left cell to cfw
CALL METHOD ref_pic_left->set_registered_events
EXPORTING
events = it_events
EXCEPTIONS
OTHERS = 1.

IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*      send event table for picture in right cell to cfw
CALL METHOD ref_pic_right->set_registered_events
EXPORTING
events = it_events
EXCEPTIONS
OTHERS = 1.

IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*      2. set event handler for ABAP object instance:
*      ref_pic_left (left cell)
SET HANDLER lcl_event_handler=>on_picture_click
FOR ref_pic_left.
SET HANDLER lcl_event_handler=>on_control_dblclick
FOR ref_pic_left.

```

*Continued on next page*

```

*      ref_pic_right (right cell)
SET HANDLER lcl_event_handler=>on_picture_click
      FOR ref_pic_right.
SET HANDLER lcl_event_handler=>on_control_dblclick
      FOR ref_pic_right.

ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING  OUTPUT

*-----*
*&      Module  USER_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type ' ':
*      - push buttons on screen 100
*      - GUI functions
*-----*
MODULE user_command_0100 INPUT.
copy_ok_code = ok_code.
CLEAR ok_code.

CASE copy_ok_code.
  WHEN 'BACK'.                               " back to program, leave screen
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        DEFAULTOPTION  = 'Y'
        textline1      = text-004
        textline2      = text-005
        titel          = text-007
        cancel_display = ' '
      IMPORTING
        answer         = l_answer.

CASE l_answer.
  WHEN 'J'.
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
  WHEN 'N'.
    SET SCREEN sy-dynnr.
ENDCASE.

```

*Continued on next page*

```

WHEN 'STRETCH'.           " picture operation: stretch to fit area
  CALL METHOD ref_pic_right->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture=>display_mode_stretch
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'NORMAL'.            " picture operation: fit to normal size
  CALL METHOD ref_pic_right->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture=>display_mode_normal
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'NORMAL_CENTER'.     " picture operation: center normal size
  CALL METHOD ref_pic_right->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture=>display_mode_normal_center
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'FIT'.                " picture operation: zoom picture
  CALL METHOD ref_pic_right->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture=>display_mode_fit
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'FIT_CENTER'.         " picture operation: zoom and center
  CALL METHOD ref_pic_right->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture=>display_mode_fit_center
    EXCEPTIONS

```

*Continued on next page*

```

        OTHERS = 1.
        IF sy-subrc NE 0.
          MESSAGE s015.
        ENDIF.

        WHEN 'MODE_INFO'.
          current_mode = ref_pic_right->display_mode.
          MESSAGE i025 WITH current_mode.

        ENDCASE.
      ENDMODULE.                                     " USER_COMMAND_0100 INPUT

*-----*
*&     Module STATUS_0100 OUTPUT
*-----*
*     Set GUI for screen 0100
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'STATUS_NORM_0100'.
  SET TITLEBAR 'TITLE_NORM_0100'.
ENDMODULE.                                       " STATUS_0100 OUTPUT

*-----*
*&     Module EXIT_COMMAND_0100 INPUT
*-----*
*     Implementation of user commands of type 'E'.
*-----*
MODULE exit_command_0100 INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.           " cancel current screen processing
      CLEAR l_answer.
      CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
        EXPORTING
          DEFAULTOPTION = 'Y'
          textline1      = text-004
          textline2      = text-005
          titel         = text-006
          cancel_display = ''
        IMPORTING
          answer        = l_answer.
    CASE l_answer.
      WHEN 'J'.
        PERFORM free_control_ressources.
        LEAVE TO SCREEN 0.
  ENDIF.
ENDMODULE.                                     " EXIT_COMMAND_0100 INPUT

```

*Continued on next page*

```

WHEN 'N'.
  CLEAR ok_code.
  SET SCREEN sy-dynnrv.
ENDCASE.

WHEN 'EXIT'.                      " leave program
  CLEAR l_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
    EXPORTING
      *          DEFAULTOPTION = 'Y'
      textline1   = text-001
      textline2   = text-002
      titel       = text-003
      cancel_display = 'X'
    IMPORTING
      answer       = l_answer.
CASE l_answer.
  WHEN 'J' OR 'N'.                 " no data to update
    PERFORM free_control_ressources.
    LEAVE PROGRAM.
  WHEN 'A'.
    CLEAR ok_code.
    SET SCREEN sy-dynnrv.
ENDCASE.
ENDCASE.
ENDMODULE.                         " EXIT_COMMAND_0100  INPUT

```

### ***Subroutines***

```

*&-----*
*&     Form  free_control_ressources
*&-----*
*      Free all control related ressources.
*-----*
*      no interface
*-----*
FORM free_control_ressources.
  CALL METHOD ref_pic_left->free.
  CALL METHOD ref_pic_right->free.
  CALL METHOD ref_splitter->free.
  CALL METHOD ref_container->free.
  FREE: ref_pic_left,
        ref_pic_right,
        cell_1_1,

```

*Continued on next page*

```
    cell_1_2,  
    ref_splitter,  
    ref_container.  
ENDFORM.                                " free_control_ressources
```



## Lesson Summary

You should now be able to:

- Describe the Splitter Container features
- Create an instance of the SAP Splitter Container Control
- Use the SAP Splitter Container
- Find a cell reference
- Read and Set attributes

# Lesson: SAP Easy Splitter Container Control

## Lesson Overview

This lesson presents the Easy Splitter Container Control.



## Lesson Objectives

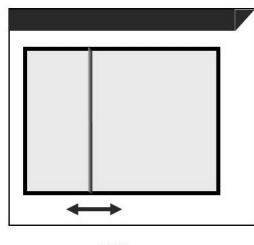
After completing this lesson, you will be able to:

- Describe the Easy Splitter Container features
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container

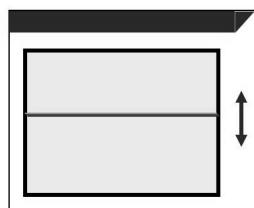
## Business Example

The application to be implemented needs an Easy Splitter Container to present an SAP Control. You need to know what is involved when using this Container type.

## SAP Easy Splitter Container: Features



or



### Properties

- Simplified version of Splitter Container Control
- Is divided into two cells
- Division can be vertical or horizontal
- Can only be placed in another container control
- Split bars can be movable or fixed
- You can control the height and width of the cells from the program
- You can display a frame for the container

**Figure 86: SAP Easy Splitter Container: Features**

The SAP Easy Splitter Container is a simplified version of the SAP Splitter Container with a smaller range of functions. It allows you to display two controls in separate cells of a container area. You can arrange the cells either horizontally or vertically.

You can only use instances of the SAP Easy Splitter Container in other container instances. The SAP Easy Splitter Container can not be attached directly to the screen.

You can nest Easy Splitter Container instances.

## Creating an Instance of the SAP Easy Splitter Container Control



```

DATA: easy_split_container TYPE REF TO
      CL_GUI_easy_splitter_container,
      cust_container TYPE REF TO CL_GUI_custom_container.
...
CREATE OBJECT easy_split_container
  EXPORTING
    parent      = cust_container
  * repid      = " program name
  * dynnr      = " screen number
    orientation = CL_GUI_easy_splitter_container->orientation_vertical
...
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
...

```

Creates easy splitter control instance, split horizontally.  
Attached to Custom Container Control

Parameter	Meaning
parent	Reference variable to relevant container object
orientation	...=>orientation_vertical split vertically ...=>orientation_horizontal split horizontally

**Figure 87: Creating an Instance of the SAP Easy Splitter Container Control**

To create an instance of the SAP Easy Splitter Container Control, you need a data object that you declare using TYPE REF TO cl\_gui\_easy\_splitter\_container.

To create the instance itself, use the statement CREATE OBJECT <object\_reference\_var>. In the statement, you must pass the relevant parameters to specify the attributes of the container. This is illustrated in the graphic. If you do not assign values to the REPID and DYNNR parameters, the system uses the current values at runtime (current dialog box level, current program name, current screen).

Use the ORIENTATION parameter to specify whether the areas should be arranged horizontally or vertically. You can do this using the class constants cl\_gui\_easy\_splitter\_container=>orientation\_vertical and cl\_gui\_easy\_splitter\_container=>orientation\_horizontal.

Use the PARENT parameter to assign your splitter control instance to another container.

To set the position of the splitter bar at runtime, use the instance method SET\_SASH\_POSITION. For further details, refer to the online documentation.

For details of the other interface parameters in the constructor, refer to the online documentation.



## SAP Easy Splitter Container: Use

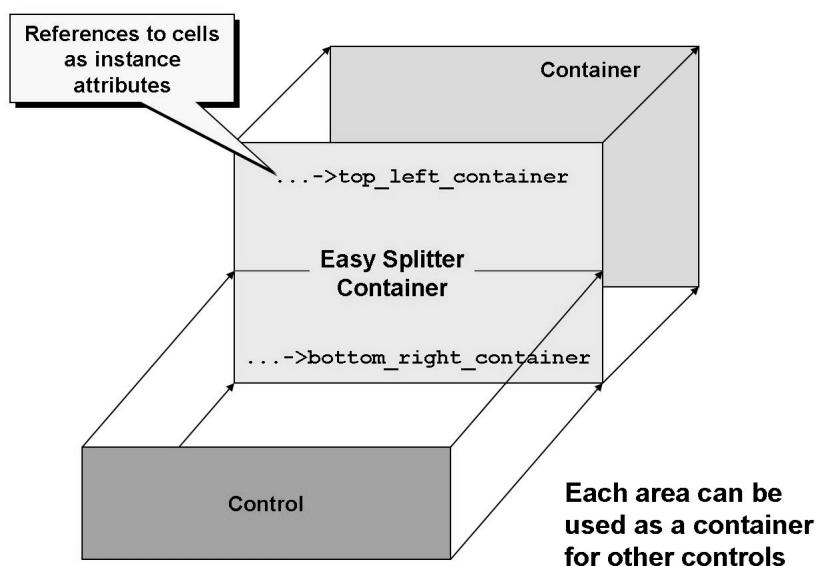


Figure 88: SAP Easy Splitter Container: Use

To assign further control instances to the cells of your splitter container, you need references to the individual cells. These are contained in the instance attributes TOP\_LEFT\_CONTAINER and BOTTOM\_RIGHT\_CONTAINER.



## Lesson Summary

You should now be able to:

- Describe the Easy Splitter Container features
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container



## Unit Summary

You should now be able to:

- Use Container Controls
- Present the inheritance hierarchy of the Container Classes
- Describe the graphical visualization of Containers
- Describe the Custom Container features
- Create an instance of the SAP Custom Container Control
- Relink an SAP Custom Container
- Describe the Dialog Box Container features
- Create an instance of the SAP Dialog Box Container Control
- Close the SAP Dialog Box Container
- Describe the Docking Container features
- Attach the SAP Docking Container to the screen
- Create an instance of the SAP Docking Container Control
- Read and Set the attributes
- Relink the SAP Docking Container
- Describe the Splitter Container features
- Create an instance of the SAP Splitter Container Control
- Use the SAP Splitter Container
- Find a cell reference
- Read and Set attributes
- Describe the Easy Splitter Container features
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container

# Unit 4

## Context Menus

### Unit Overview

This is the last unit with a general basic topic. The unit presents necessary basic information valid for all following controls. Context Menus can be applied on each EnjoySAP Control – and, of course, to screen based dialog elements, too (covered in BC410 Programming User Dialogs).



### Unit Objectives

After completing this unit, you will be able to:

- Describe how context menus work with EnjoySAP Controls
- Create context menus for EnjoySAP Controls
- Process functions selected in context menus

### Unit Contents

Lesson: Context Menus.....	196
Exercise 8: Assigning a context menu to different Picture Controls .	205

# Lesson: Context Menus

## Lesson Overview

The Lesson presents the usage of Context Menus, static and dynamic Context Menus and processing functions from Context Menus.



## Lesson Objectives

After completing this lesson, you will be able to:

- Describe how context menus work with EnjoySAP Controls
- Create context menus for EnjoySAP Controls
- Process functions selected in context menus

## Business Example

You need to implement an application using the EnjoySAP Controls. You can apply Context Menus on every EnjoySAP Control.

## Context Menus

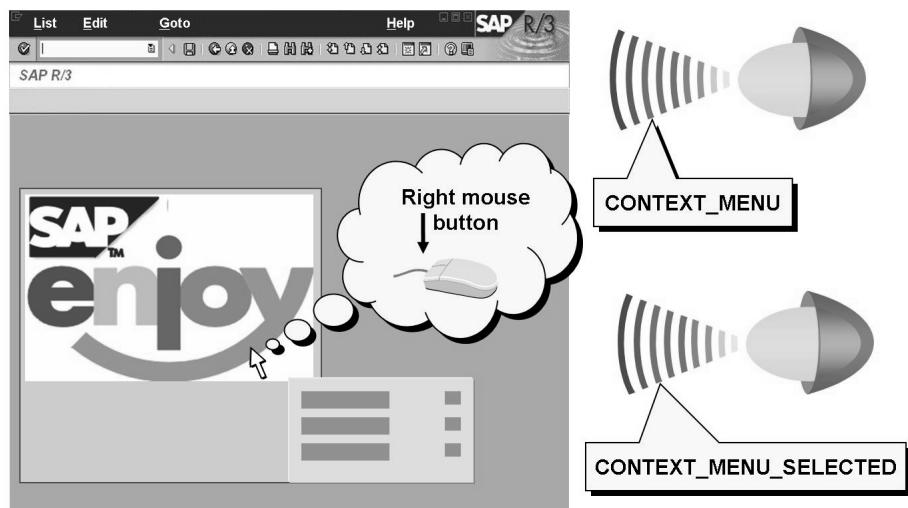


Figure 89: Context Menus

You can use a context menu to offer users functions that are specific to a certain program context.

The system displays a context menu when the user chooses the right mouse button or <Shift + F10> while the mouse cursor is positioned over an EnjoySAP Control.

When the user chooses the right mouse button or <Shift + F10> the control triggers the **CONTEXT\_MENU** event. The system decides what program context applies from the position of the mouse cursor – that is, from the control that triggered the event.

If the user chooses a function from the context menu the relevant control triggers the **CONTEXT\_MENU\_SELECTED** event.

## Using Context Menus

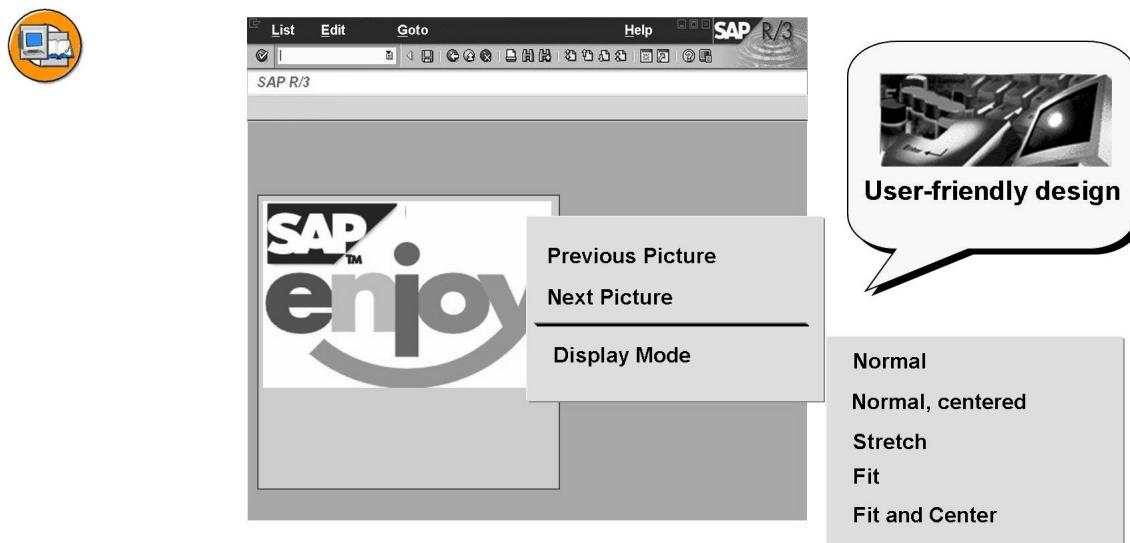


Figure 90: Using Context Menus

Use the context menu whenever the functions should be applied to the control, not the whole screen.

**Example:** To change the display mode of a picture, use a context menu.

Follow these rules on software ergonomics when you create context menus:

- Create no more than two hierarchy levels
- Create a maximum of ten entries, but make sure you include functionality for all pushbuttons.
- Structure your context menu using separators.
- Put object-specific functions at the top.

## EnjoySAP Controls and Context Menus



Context menu events:

Control	Context menu chosen	Function chosen
Picture	CONTEXT_MENU	CONTEXT_MENU_SELECTED
HTML Viewer	CTXMENU_REQUEST	CTXMENU_SELECTED
Text Edit	CONTEXT_MENU	CONTEXT_MENU_SELECTED
SAP Grid	CONTEXT_MENU_REQUEST	USER_COMMAND
Tree	CONTEXT_MENU	NODE/DEFAULT_CONTEXT_...

Comments:

- Picture:  
The reference to the context menu must be passed to the Control
- HTML Viewer:  
The reference to the context menu must be passed to the Control
- Tree:  
The Control registers the *NODE\_CONTEXT\_MENU\_SELECT* and *DEFAULT\_CONTEXT\_MENU\_SELECT* events in the Control Framework.

**Figure 91: EnjoySAP Controls and Context Menus**

In general, the context menus are handled similarly, except that the technical names of events differ, depending on the EnjoySAP Control.

However, in the case of the Picture Control and the HTML Viewer Control, the event does **not** generate a context menu, so these controls **do not** pass a reference. For these controls, you instantiate the context menu in the program itself and pass the reference using the *DISPLAY\_CONTEXT\_MENU* and *TRACK\_CONTEXT\_MENU* methods.

names of events for Tree are:

- **NODE\_CONTEXT\_MENU\_REQUEST** and **DEFAULT\_CONTEXT\_MENU\_REQUEST**
- **NODE\_CONTEXT\_MENU\_SELECT** and **DEFAULT\_CONTEXT\_MENU\_SELECT**

## The CONTEXT\_MENU Event



```

CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS: on_context_menu
      FOR EVENT context_menu OF cl_gui<control>
        IMPORTING ref_menu.
      ...
    ENDCLASS.

    CLASS lcl_event_handler IMPLEMENTATION.
      METHOD on_context_menu.
      ...
    ENDMETHOD.

    ...
  ENDCLASS.

  ...
  SET HANDLER lcl_event_handler=>on_context_menu
    FOR ... .
  
```

The static method `on_context_menu` can react to the event `context_menu` and imports a reference to an object of the class `cl_ctmenu`

Implementing the static method `on_context_menu`

Registering the handler method

**Figure 92: The CONTEXT\_MENU Event**

When the user chooses the right mouse button or <Shift + F10>, the control underneath the mouse cursor triggers the CONTEXT\_MENU event.

To display a special context menu, you must implement a handler method for this event and register it on the application server (SET HANDLER ... FOR EVENT ... OF ...). In general, you also need to set the filter in the Automation Controller. To do this, use the instance method SET\_REGISTERED\_EVENTS of the control for which you are creating a context menu.

## The Picture Control: A Special Case



```
CLASS lcl_event_handler DEFINITION.  
  PUBLIC SECTION.  
    CLASS-METHODS: on_context_menu  
      FOR EVENT context_menu OF cl_gui_picture  
        IMPORTING sender.  
      ...  
  ENDCLASS.  
  
CLASS lcl_event_handler IMPLEMENTATION.  
  METHOD on_context_menu.  
    DATA ref_menu TYPE REF TO cl_ctmenu.  
    CREATE OBJECT ref_menu.  
    ...  
    CALL METHOD sender->display_context_menu  
      EXPORTING context_menu = ref_menu.  
  ENDMETHOD.  
ENDCLASS.
```

Event handler method imports a reference to the Picture Control that raises the event

Define the object reference and create an instance of the context menu

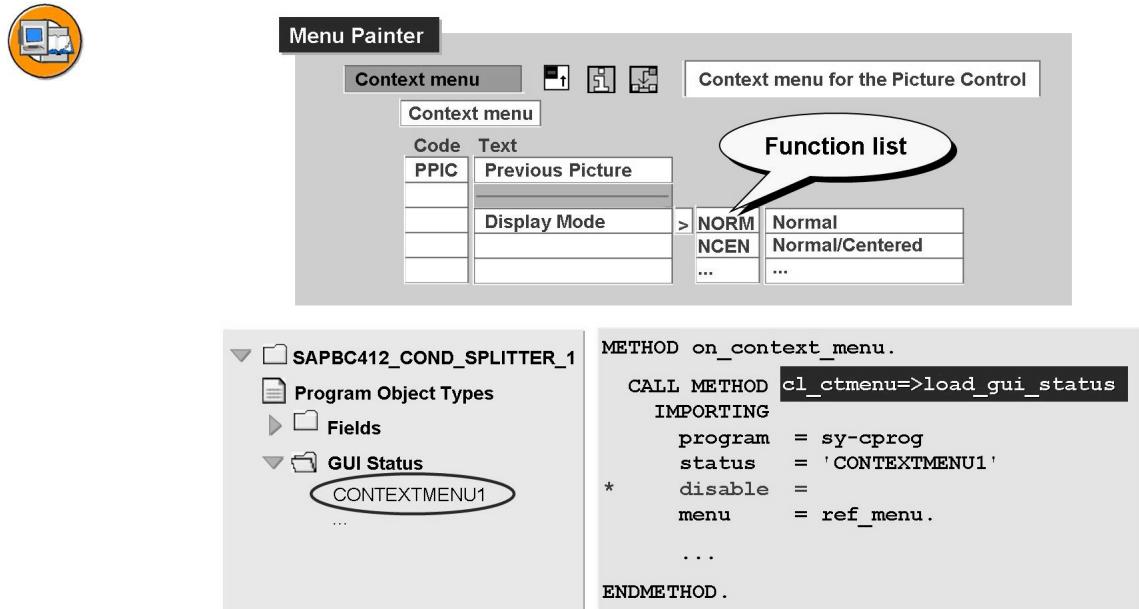
Send the context menu to the Picture Control

**Figure 93: The Picture Control: A Special Case**

The CONTEXT\_MENU event of the Picture Control does **not** send a reference to the context menu.

In this case, you must instantiate the context menu yourself and then send it to the Control using the DISPLAY\_CONTEXT\_MENU method, after you have assigned the relevant functions.

## Creating Static Context Menus with the Menu Painter



**Figure 94: Creating Static Context Menus with the Menu Painter**

The easiest way to create a context menu is to load a GUI status of the type Context menu. You create the status in the Menu Painter and assign it to the program user interface.

To load the GUI status in the context menu, use the static method LOAD\_GUI\_STATUS of the class CL\_CTMENU. The parameters are filled as follows: **PROGRAM**: Name of the ABAP program where the GUI status is defined (= sy-cprog, if the status belongs to the GUI of the current program) The Lesson presents the usage of Context Menus, static and dynamic Context Menus and processing functions from Context Menus

**STATUS**: Name of the predefined GUI status

**DISABLE**: Table of functions that are to be inactive in the context menu (Table type: UI\_FUNCTIONS)

**MENU**: Reference to the object of the class CL\_CTMENU, in which the GUI status is to be loaded

## Dynamic Context Menus



Methods of the class CL\_CTMENU:

<i>Method</i>	<i>Meaning</i>
<b>ADD_FUNCTION</b>	Add a function
<b>ADD_SEPARATOR</b>	Add a separator
<b>HIDE_FUNCTIONS</b>	Hide functions
<b>SHOW_FUNCTIONS</b>	Show functions
<b>DISABLE_FUNCTIONS</b>	Deactivate functions
<b>ENABLE_FUNCTIONS</b>	Activate functions

Figure 95: Dynamic Context Menus

The class CL\_CTMENU provides several other methods apart from the static method LOAD\_GUI\_STATUS. You can use these to create or adapt a context menu **at runtime** (generally using values in specified data objects).

You can then call the appropriate methods within the handler method.

For further information, refer to the keyword documentation for the CL\_CTMENU class.

## Processing Functions from Context Menus

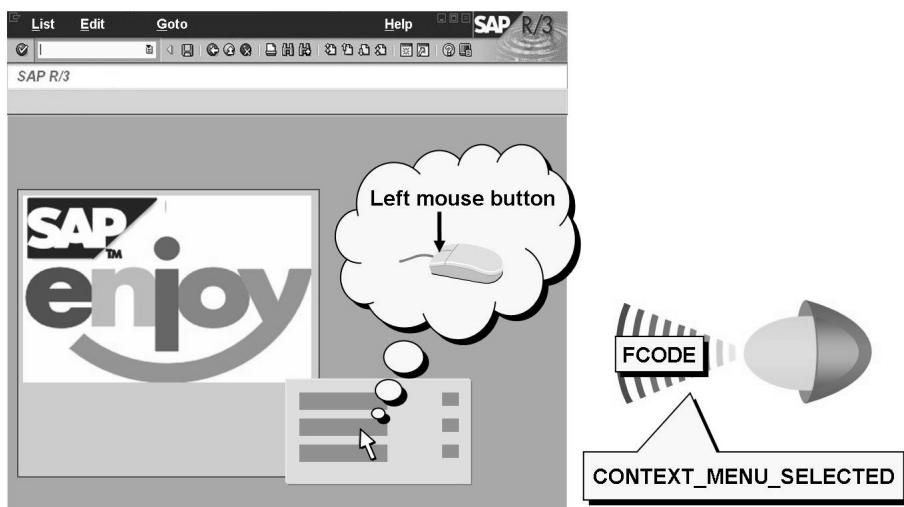


Figure 96: Processing Functions from Context Menus

If the user chooses a function from the context menu, the **CONTEXT\_MENU\_SELECTED** event is raised.

The event exports the code of the function chosen by the user.

## The CONTEXT\_MENU\_SELECTED Event



```
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS: on_ctmenu_selected
      FOR EVENT context_menu_selected
        OF cl_gui<control>
        IMPORTING fcode sender.
  ENDCLASS.

  CLASS lcl_event_handler IMPLEMENTATION.
    METHOD on_ctmenu_selected.
      CASE fcode.
        WHEN 'FC1'.
        ...
        WHEN 'FC2'.
        ...
      ENDCASE.
    ENDMETHOD.
  ENDCLASS.
```

The static method `on_ctmenu_selected` imports the function code `fcode` and the sender of the event

Case distinction for the function chosen by the user

**Figure 97: The CONTEXT\_MENU\_SELECTED Event**

You need to implement an event handler method and register it with the Automation Controller and on the application server. This does not apply to the Tree Control, where the control itself registers the method with the Automation Controller.

Within the handler method, you then decide what function to perform using the function code.



## Exercise 8: Assigning a context menu to different Picture Controls

### Exercise Objectives

After completing this exercise, you will be able to:

- Generate a context menu
- Send a context menu to a Picture Control instance
- Implement the functions of a context menu

### Business Example

You will be improving your program from the exercise you completed in the previous chapter. Your program will now offer the functions for changing the image display mode using a context menu rather than pushbuttons on the screen. This allows the user to change the display of each image individually.

**Program:** ZBC412\_##\_CMU\_EX1

**Copy template:** SAPBC412\_CONS\_EXERCISE2

**Model solution:** SAPBC412\_CMUS\_EXERCISE1

where ## is the group number.

### Task 1:

Copy the Template

1. Copy your solution from the last exercise in chapter “SAP Container” (ZBC412\_##\_CON\_EX2) or the appropriate model solution SAPBC412\_CONS\_EXERCISE2 with all their sub-objects to the name **ZBC412\_##\_CMU\_EX1**. Get to know how your copy of the program works.

### Task 2:

Create a GUI status for the context menu that contains the functions for controlling how the image is displayed as follows:

1. In the *Menu Painter*, create a status of the type *Context menu* (suggested name: CTXMENU) and assign it to the GUI belonging to your program.

*Continued on next page*

2. To create the function entries, copy the five function codes and texts from the existing screen pushbuttons.



**Hint:** The input fields in the *Menu Painter* are too short for the function codes and texts. For best results therefore, you should:

*Choose Goto → Object Lists → Function List.* Create entries using *Create (<F5>)*.

Then add these entries to the context menu using (<F4>) and double-clicking.

3. Activate the GUI Status.

### Task 3:

Define and implement a handler method for the context menu event of the picture control and register it to both picture control instances.

1. In the local class `lcl_event_handler`, implement another handler method for the `context_menu` event of the `cl_gui_picture` class (suggested name: `on_context_menu`).
2. Define a reference for the context menu locally, within the method `on_context_menu` (suggested name: `l_ref_menu`).
3. Create a context menu instance and load the status CTXMENU using the `load_gui_status` method.
4. Pass the context menu `l_ref_menu` to the Picture Control instance that raised the event using the method `display_context_menu`.
5. In the PBO module `init_control_processing`, register the handler method for the event of **both** Picture Control instances with the Automation Controller.
6. Register the method for both picture control instances on the application server (use the SET HANDLER statement).

*Continued on next page*

## Task 4:

Activate and test your program.

## Task 5:

Define and implement a handler method for the event of the picture control that is fired when the user has chosen a function. Register the handler method to both picture control instances.

1. In the local class `lcl_event_handler`, implement another handler method for the `context_menu_selected` event of the `cl_gui_picture` class (suggested name: `on_context_menu_selected`).
2. Set the image display mode the user has chosen for the Picture Control instance that raised the event using the `set_display_mode` method



**Hint:** Use the import parameter `fcode` and the standard import parameter `sender` of the event. You can reuse most of your source code from the module `user_command_0100`.

3. In the PBO module `init_control_processing`, register the event and its handler method for **both** Picture Control instances with the Automation Controller.
4. Register the method for both picture control instances on the application server (use the SET HANDLER statement).

## Task 6:

1. Activate and test your program

## Task 7:

Remove the obsolete pushbuttons and the coding related to them.

1. Finally, remove the obsolete pushbuttons from your screen and delete the associated source code segments from the PAI module `user_command_0100`. After that you can then further enlarge the Custom Container area on the screen.

*Continued on next page*

## Task 8: Optional

In the statically created context menu, dynamically hide the function that corresponds to the current display mode for the Picture Control instance that has made the context menu request:

1. Define an internal table for the codes for the functions that are to be hidden locally within the method. (Suggested name: `l_it_functions`, of the global type `ui_functions`).
2. Again locally within the method, define a suitable work area for this internal table (suggested name: `l_wa_func`).
3. In each case, write the function code for the current display mode to the work area `l_wa_func`.
4. Insert the work area `l_wa_func` in the internal table `l_it_functions` and apply the `disable_functions` method to it.



**Hint:** Get the current display mode from the instance attribute `display_mode`. For the comparison, use the class constants `cl_gui_picture=>display_mode_....`.

## Solution 8: Assigning a context menu to different Picture Controls

### Task 1:

Copy the Template

1. Copy your solution from the last exercise in chapter “SAP Container” (ZBC412\_##\_CON\_EX2) or the appropriate model solution SAPBC412\_CONS\_EXERCISE2) with all their sub-objects to the name **ZBC412\_##\_CMU\_EX1**. Get to know how your copy of the program works.
  - a) -

### Task 2:

Create a GUI status for the context menu that contains the functions for controlling how the image is displayed as follows:

1. In the *Menu Painter*, create a status of the type *Context menu* (suggested name: CTXMENU) and assign it to the GUI belonging to your program.
  - a) -

*Continued on next page*

2. To create the function entries, copy the five function codes and texts from the existing screen pushbuttons.



**Hint:** The input fields in the *Menu Painter* are too short for the function codes and texts. For best results therefore, you should:

*Choose Goto → Object Lists → Function List.* Create entries using *Create (<F5>)*.

Then add these entries to the context menu using (<F4>) and double-clicking.

a) **GUI Status**

CTXMENU

**Function list**

<b>FIT</b>	<b>Fill container area but maintain height/width ratio</b>
<b>FIT_CENTER</b>	<b>Fill container area and center image but maintain height/width ratio</b>
<b>NORMAL</b>	<b>Display at normal size</b>
<b>NOR-MAL_CENTER</b>	<b>Center in original size</b>
<b>STRETCH</b>	<b>Fill container area</b>

3. Activate the GUI Status.

a) –

*Continued on next page*

### Task 3:

Define and implement a handler method for the context menu event of the picture control and register it to both picture control instances.

1. In the local class `lcl_event_handler`, implement another handler method for the `context_menu` event of the `cl_gui_picture` class (suggested name: `on_context_menu`).
  - a)

```
CLASS lcl_event_handler DEFINITION.  
  PUBLIC SECTION.  
    CLASS-METHODS:  
      ...  
      on_context_menu  
        FOR EVENT context_menu OF cl_gui_picture  
        IMPORTING sender.  
      ...  
  ENDCLASS.
```

2. Define a reference for the context menu locally, within the method `on_context_menu` (suggested name: `l_ref_menu`).

- a)

```
CLASS lcl_event_handler IMPLEMENTATION.  
  ...  
  METHOD on_context_menu.  
    DATA:  
      l_ref_menu TYPE REF TO cl_ctmenu.  
    ...  
  ENDMETHOD.  
  ...  
ENDCLASS.
```

*Continued on next page*

3. Create a context menu instance and load the status CTXMENU using the `load_gui_status` method.

a)

```
CLASS lcl_event_handler IMPLEMENTATION.  
...  
METHOD on_context_menu.  
...  
    CREATE OBJECT l_ref_menu.  
  
    * assign functions to context menu:  
    CALL METHOD l_ref_menu->load_gui_status  
        EXPORTING  
            program      = sy-cprog  
            status       = 'CTXMENU'  
        * DISABLE      =  
            menu        = l_ref_menu  
        EXCEPTIONS  
            OTHERS      = 1.  
        IF sy-subrc <> 0.  
            MESSAGE a012.  
        ENDIF.  
  
    ...  
ENDMETHOD.  
...  
ENDCLASS.
```

*Continued on next page*

4. Pass the context menu `l_ref_menu` to the Picture Control instance that raised the event using the method `display_context_menu`.

a)



**Hint:** Use the standard import parameter `sender` of the event.

```
CLASS lcl_event_handler IMPLEMENTATION.  
...  
METHOD on_context_menu.  
...  
    CALL METHOD sender->display_context_menu  
        EXPORTING  
            context_menu = l_ref_menu  
        EXCEPTIONS  
            OTHERS      = 1.  
        IF sy-subrc <> 0.  
            MESSAGE a012.  
        ENDIF.  
  
    ENDMETHOD.  
...  
ENDCLASS.
```

5. In the PBO module `init_control_processing`, register the handler method for the event of **both** Picture Control instances with the Automation Controller.

a)

```
wa_events-eventid = cl_gui_picture=>eventid_context_menu.  
wa_events-appl_event = ' '.  
INSERT wa_events INTO TABLE it_events.
```

*Continued on next page*

6. Register the method for both picture control instances on the application server (use the SET HANDLER statement).

a)

```
SET HANDLER lcl_event_handler=>on_context_menu
FOR ref_pic_left.
```

```
SET HANDLER lcl_event_handler=>on_context_menu
FOR ref_pic_right.
```

### Task 4:

Activate and test your program.

### Task 5:

Define and implement a handler method for the event of the picture control that is fired when the user has chosen a function. Register the handler method to both picture control instances.

1. In the local class lcl\_event\_handler, implement another handler method for the context\_menu\_selected event of the cl\_gui\_picture class (suggested name: on\_context\_menu\_selected).

a)

```
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
    ...
      on_context_menu_selected
        FOR EVENT context_menu_selected OF cl_gui_picture
          IMPORTING fcode sender.
    ...
ENDCLASS.
```

2. Set the image display mode the user has chosen for the Picture Control instance that raised the event using the set\_display\_mode method



**Hint:** Use the import parameter fcode and the standard import parameter sender of the event. You can reuse most of your source code from the module user\_command\_0100.

*Continued on next page*

a)

```

CLASS lcl_event_handler IMPLEMENTATION.

...
METHOD on_context_menu_selected.
CASE fcode.
  WHEN 'STRETCH'.           " picture operation: stretch to fit
    CALL METHOD sender->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_stretch
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'NORMAL'.            " picture operation: fit to normal size
    CALL METHOD sender->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_normal
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'NORMAL_CENTER'.     " picture operation: center in normal size
    CALL METHOD sender->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_normal_center
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'FIT'.                " picture operation: zoom picture
    CALL METHOD sender->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_fit
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

```

*Continued on next page*

```

WHEN 'FIT_CENTER'.           " picture operation: zoom and center
    CALL METHOD sender->set_display_mode
        EXPORTING
            display_mode = cl_gui_picture=>display_mode_fit_center
        EXCEPTIONS
            OTHERS = 1.
        IF sy-subrc NE 0.
            MESSAGE s015.
        ENDIF.
    ENDCASE.
ENDMETHOD.

...
ENDCLASS.

```

3. In the PBO module `init_control_processing`, register the event and its handler method for **both** Picture Control instances with the Automation Controller.
- a)

```

wa_events-eventid = cl_gui_picture=>eventid_context_menu_selected.
wa_events-appl_event = ''.
INSERT wa_events INTO TABLE it_events.

```

4. Register the method for both picture control instances on the application server (use the SET HANDLER statement).

- a)

```

SET HANDLER lcl_event_handler=>on_context_menu_selected
FOR ref_pic_left.

SET HANDLER lcl_event_handler=>on_context_menu_selected
FOR ref_pic_right.

```

## Task 6:

1. Activate and test your program
- a) -

*Continued on next page*

## Task 7:

Remove the obsolete pushbuttons and the coding related to them.

1. Finally, remove the obsolete pushbuttons from your screen and delete the associated source code segments from the PAI module `user_command_0100`. After that you can then further enlarge the Custom Container area on the screen.

a) –

## Task 8: Optional

In the statically created context menu, dynamically hide the function that corresponds to the current display mode for the Picture Control instance that has made the context menu request:

1. Define an internal table for the codes for the functions that are to be hidden locally within the method. (Suggested name: `l_it_functions`, of the global type `ui_functions`).

a) –

2. Again locally within the method, define a suitable work area for this internal table (suggested name: `l_wa_func`).

a) –

3. In each case, write the function code for the current display mode to the work area `l_wa_func`.

a) –

4. Insert the work area `l_wa_func` in the internal table `l_it_functions` and apply the `disable_functions` method to it.



**Hint:** Get the current display mode from the instance attribute `display_mode`. For the comparison, use the class constants `cl_gui_picture->display_mode_...`

a)

```
CLASS lcl_event_handler IMPLEMENTATION.
...
METHOD on_context_menu.
  DATA:
...
    l_it_functions TYPE ui_functions,      " optional part
    l_wa_func LIKE LINE OF l_it_functions. " optional part
...

```

*Continued on next page*

```

*      disable actual display mode (optional part):
CASE sender->display_mode.
  WHEN cl_gui_picture=>display_mode_normal.
    l_wa_func = 'NORMAL'.
  WHEN cl_gui_picture=>display_mode_normal_center.
    l_wa_func = 'NORMAL_CENTER'.
  WHEN cl_gui_picture=>display_mode_stretch.
    l_wa_func = 'STRETCH'.
  WHEN cl_gui_picture=>display_mode_fit.
    l_wa_func = 'FIT'.
  WHEN cl_gui_picture=>display_mode_fit_center.
    l_wa_func = 'FIT_CENTER'.
ENDCASE.
INSERT l_wa_func INTO TABLE l_it_functions.
CALL METHOD l_ref_menu->disable_functions
  EXPORTING
    fcodes = l_it_functions.
...
ENDMETHOD.
...
ENDCLASS.
```

## Result

### Screen flow logic

#### *SCREEN 100*

```

PROCESS BEFORE OUTPUT.
MODULE status_0100.
MODULE init_control_processing.

PROCESS AFTER INPUT.
MODULE exit_command_0100 AT EXIT-COMMAND.
MODULE user_command_0100.
```

### ABAP Program

#### *Data Declarations*

```

REPORT  sapbc412_cmus_exercise1 MESSAGE-ID bc412.

TYPE-POOLS: cntl.
```

*Continued on next page*

```

TYPES:
  t_url          TYPE bapiuri-uri.

DATA:
  ok_code        TYPE sy-ucomm,
  copy_ok_code  LIKE ok_code,
  l_answer       TYPE c,

*   control specific: object references
  ref_container  TYPE REF TO cl_gui_custom_container,
  ref_splitter   TYPE REF TO cl_gui_splitter_container,
  ref_pic_left   TYPE REF TO cl_gui_picture,
  ref_pic_right  TYPE REF TO cl_gui_picture,

*   object references to splitter control areas
  cell_1_1       TYPE REF TO cl_gui_container,
  cell_1_2       TYPE REF TO cl_gui_container,

*   control specific: auxiliary fields
  l_url1         TYPE t_url,
  l_url2         TYPE t_url,

  current_mode   LIKE cl_gui_picture=>display_mode,

*   event registration
  it_events      TYPE cntl_simple_events,
  wa_events      LIKE LINE OF it_events.

```

### ***ABAP Program: Event Blocks***

```

START-OF-SELECTION.

CALL FUNCTION 'BC412_BDS_GET_PIC_URL' fetch URL of first picture
*      EXPORTING                               " from BDS
*          NUMBER      = 1
IMPORTING
  url          = l_url1
EXCEPTIONS
  OTHERS       = 1.

IF sy-subrc <> 0.                                " no picture --> end of program
MESSAGE i035(bc412) WITH 1.

```

*Continued on next page*

```

LEAVE PROGRAM.

ENDIF.

CALL FUNCTION 'BC412_BDS_GET_PIC_URL' fetch URL of second picture
  EXPORTING
    number          = 2
  IMPORTING
    url            = l_url2
  EXCEPTIONS
    OTHERS         = 1.

IF sy-subrc <> 0.                      " no picture --> end of program
  MESSAGE i035(bc412) WITH 2.
  LEAVE PROGRAM.
ENDIF.

CALL SCREEN 100.

```

### *Local Classes*

```

*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
*      Definition of a local class containing event handler methods *
*      for picture control objects *
*-----*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      on_picture_click
        FOR EVENT picture_click OF cl_gui_picture
        IMPORTING mouse_pos_x mouse_pos_y,
      * -----
      on_control_dblclick
        FOR EVENT control_dblclick OF cl_gui_picture
        IMPORTING mouse_pos_x mouse_pos_y,
      * -----
      on_context_menu
        FOR EVENT context_menu OF cl_gui_picture
        IMPORTING sender,
      * -----
      on_context_menu_selected
        FOR EVENT context_menu_selected OF cl_gui_picture
        IMPORTING fcode sender.

```

*Continued on next page*

```

ENDCLASS.

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
*      Corresponding class implementation
*-----*
CLASS lcl_event_handler IMPLEMENTATION.

METHOD on_picture_click.
  MESSAGE i016 WITH mouse_pos_x mouse_pos_y.
ENDMETHOD.

METHOD on_control_dblclick.
  MESSAGE i017 WITH mouse_pos_x mouse_pos_y.
ENDMETHOD.

METHOD on_context_menu.

DATA:
  l_ref_menu TYPE REF TO cl_ctmenu,
  l_it_functions TYPE ui_functions,          " optional part
  l_wa_func LIKE LINE OF l_it_functions. " optional part

CREATE OBJECT l_ref_menu.

* assign functions to context menu:
CALL METHOD l_ref_menu->load_gui_status
  EXPORTING
    program      = sy-cprog
    status       = 'CTXMENU'
*  DISABLE      =
    menu        = l_ref_menu
  EXCEPTIONS
    OTHERS      = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

* disable actual display mode (optional part):
CASE sender->display_mode.
  WHEN cl_gui_picture=>display_mode_normal.
    l_wa_func = 'NORMAL'.
  WHEN cl_gui_picture=>display_mode_normal_center.
    l_wa_func = 'NORMAL_CENTER'.
  WHEN cl_gui_picture=>display_mode_stretch.
    l_wa_func = 'STRETCH'.

```

*Continued on next page*

```

WHEN cl_gui_picture=>display_mode_fit.
  l_wa_func = 'FIT'.
WHEN cl_gui_picture=>display_mode_fit_center.
  l_wa_func = 'FIT_CENTER'.
ENDCASE.
INSERT l_wa_func INTO TABLE l_it_functions.
CALL METHOD l_ref_menu->disable_functions
  EXPORTING
    fcodes = l_it_functions.

* assign context menu to picture control:
CALL METHOD sender->display_context_menu
  EXPORTING
    context_menu = l_ref_menu
  EXCEPTIONS
    OTHERS      = 1.
IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

ENDMETHOD.

METHOD on_context_menu_selected.
CASE fcode.
  WHEN 'STRETCH'.           " picture operation: stretch to fit
    CALL METHOD sender->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_stretch
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'NORMAL'.            " picture operation: fit to normal size
    CALL METHOD sender->set_display_mode
      EXPORTING
        display_mode = cl_gui_picture=>display_mode_normal
      EXCEPTIONS
        OTHERS = 1.
    IF sy-subrc NE 0.
      MESSAGE s015.
    ENDIF.

  WHEN 'NORMAL_CENTER'.     " picture operation: center in normal size

```

*Continued on next page*

```

CALL METHOD sender->set_display_mode
EXPORTING
    display_mode = cl_gui_picture->display_mode_normal_center
EXCEPTIONS
    OTHERS = 1.
IF sy-subrc NE 0.
MESSAGE s015.
ENDIF.

WHEN 'FIT'.                      " picture operation: zoom picture
CALL METHOD sender->set_display_mode
EXPORTING
    display_mode = cl_gui_picture->display_mode_fit
EXCEPTIONS
    OTHERS = 1.
IF sy-subrc NE 0.
MESSAGE s015.
ENDIF.

WHEN 'FIT_CENTER'.           " picture operation: zoom and center
CALL METHOD sender->set_display_mode
EXPORTING
    display_mode = cl_gui_picture->display_mode_fit_center
EXCEPTIONS
    OTHERS = 1.
IF sy-subrc NE 0.
MESSAGE s015.
ENDIF.
ENDCASE.
ENDMETHOD.

ENDCLASS.

```

### **Modules**

```

*&-----*
*&     Module  INIT_CONTROL_PROCESSING  OUTPUT
*&-----*
MODULE init_control_processing OUTPUT.
    IF ref_container IS INITIAL.          " prevent re-processing on ENTER
*   create custom container object and link to screen area
    CREATE OBJECT ref_container
    EXPORTING
        container_name = 'CONTROL_AREA1'

```

*Continued on next page*

```

EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
  MESSAGE a010.                      " cancel program processing
ENDIF.

*   create splitter container object and link to custom control
CREATE OBJECT ref_splitter
  EXPORTING
    parent  = ref_container
    rows     = 1
    columns  = 2
  EXCEPTIONS
    others = 1.

IF sy-subrc NE 0.
  MESSAGE a010.                      " cancel program processing
ENDIF.

*   fetch pointer to splitter control areas
*   left cell
CALL METHOD ref_splitter->get_container
  EXPORTING
    row      = 1
    column   = 1
  RECEIVING
    container = cell_1_1.

*   right cell
CALL METHOD ref_splitter->get_container
  EXPORTING
    row      = 1
    column   = 2
  RECEIVING
    container = cell_1_2.

*   create picture control objects and link to cell pointer
*   picture control object 1 (left cell)
CREATE OBJECT ref_pic_left
  EXPORTING
    parent = cell_1_1
  EXCEPTIONS
    others = 1.

```

*Continued on next page*

Internal Use SAP Partner Only

```

IF sy-subrc NE 0.
MESSAGE a011.                                     " cancel program processing
ENDIF.

*   picture control object 2 (right cell)
CREATE OBJECT ref_pic_right
EXPORTING
parent = cell_1_2
EXCEPTIONS
others = 1.

IF sy-subrc NE 0.
MESSAGE a011.                                     " cancel program processing
ENDIF.

*   load pictures into picture control objects
*   left cell
CALL METHOD ref_pic_left->load_picture_from_url
EXPORTING
url      = l_url1
EXCEPTIONS
OTHERS = 1.
IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*   right cell
CALL METHOD ref_pic_right->load_picture_from_url
EXPORTING
url      = l_url2
EXCEPTIONS
OTHERS = 1.
IF sy-subrc NE 0.
MESSAGE a012.
ENDIF.

*   event handling
*   1. register events for control framework
wa_events-eventid      = cl_gui_picture->eventid_picture_click.
wa_events-appl_event = ' '.
INSERT wa_events INTO TABLE it_events.

wa_events-eventid      = cl_gui_picture->eventid_control_dblclick.
wa_events-appl_event = ' '.

```

*Continued on next page*

```

        INSERT wa_events INTO TABLE it_events.

        wa_events-eventid = cl_gui_picture=>eventid_context_menu.
        wa_events-appl_event = ' '.
        INSERT wa_events INTO TABLE it_events.

        wa_events-eventid = cl_gui_picture=>eventid_context_menu_selected.
        wa_events-appl_event = ' '.
        INSERT wa_events INTO TABLE it_events.

*   send event table for picture in left cell to cfw
        CALL METHOD ref_pic_left->set_registered_events
            EXPORTING
                events = it_events
            EXCEPTIONS
                OTHERS = 1.

        IF sy-subrc NE 0.
            MESSAGE a012.
        ENDIF.

*   send event table for picture in right cell to cfw
        CALL METHOD ref_pic_right->set_registered_events
            EXPORTING
                events = it_events
            EXCEPTIONS
                OTHERS = 1.

        IF sy-subrc NE 0.
            MESSAGE a012.
        ENDIF.

*   2. set event handler for ABAP object instance:
*   ref_pic_left (left cell)
        SET HANDLER lcl_event_handler=>on_picture_click
            FOR ref_pic_left.
        SET HANDLER lcl_event_handler=>on_control_dblclick
            FOR ref_pic_left.
        SET HANDLER lcl_event_handler=>on_context_menu
            FOR ref_pic_left.
        SET HANDLER lcl_event_handler=>on_context_menu_selected
            FOR ref_pic_left.

*   ref_pic_right (right cell)
        SET HANDLER lcl_event_handler=>on_picture_click

```

*Continued on next page*

```

        FOR ref_pic_right.
        SET HANDLER lcl_event_handler->on_control_dblclick
            FOR ref_pic_right.
        SET HANDLER lcl_event_handler->on_context_menu
            FOR ref_pic_right.
        SET HANDLER lcl_event_handler->on_context_menu_selected
            FOR ref_pic_right.

        ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING OUTPUT

*&-----*
*&     Module   USER_COMMAND_0100  INPUT
*&-----*
*     Implementation of user commands of type ' ':
*     - push buttons on screen 100
*     - GUI functions
*-----*
MODULE user_command_0100 INPUT.
    copy_ok_code = ok_code.
    CLEAR ok_code.

CASE copy_ok_code.
    WHEN 'BACK'.                               " back to program, leave screen
        CLEAR l_answer.
        CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
            EXPORTING
*             DEFAULTOPTION = 'Y'
*             textline1      = text-004
*             textline2      = text-005
*             titel         = text-007
*             cancel_display = ' '
            IMPORTING
                answer          = l_answer.

CASE l_answer.
    WHEN 'J'.
        PERFORM free_control_ressources.
        LEAVE TO SCREEN 0.
    WHEN 'N'.
        SET SCREEN sy-dynnr.
ENDCASE.

ENDCASE.
ENDMODULE.                                     " USER_COMMAND_0100  INPUT

```

*Continued on next page*

```

*-----*
*&     Module STATUS_0100  OUTPUT
*-----*
*      Set GUI for screen 0100
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'STATUS_NORM_0100'.
  SET TITLEBAR  'TITLE_NORM_0100'.
ENDMODULE.                                     " STATUS_0100  OUTPUT

*-----*
*&     Module EXIT_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type 'E'.
*-----*
MODULE exit_command_0100 INPUT.
CASE ok_code.
  WHEN 'CANCEL'.           " cancel current screen processing
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *          DEFAULTOPTION = 'Y'
        textline1   = text-004
        textline2   = text-005
        titel       = text-006
        cancel_display = ' '
      IMPORTING
        answer      = l_answer.
CASE l_answer.
  WHEN 'J'.
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
  WHEN 'N'.
    CLEAR ok_code.
    SET SCREEN sy-dynnr.
ENDCASE.

WHEN 'EXIT'.                                " leave program
  CLEAR l_answer.
  CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
    EXPORTING
      *          DEFAULTOPTION = 'Y'

```

*Continued on next page*

```

textline1      = text-001
textline2      = text-002
titel         = text-003
cancel_display = 'X'
IMPORTING
    answer          = l_answer.
CASE l_answer.
    WHEN 'J' OR 'N'.           " no data to update
        PERFORM free_control_ressources.
    LEAVE PROGRAM.
    WHEN 'A'.
        CLEAR ok_code.
        SET SCREEN sy-dynnr.
    ENDCASE.
ENDCASE.
ENDMODULE.           " EXIT_COMMAND_0100  INPUT

```

### *Subroutines*

```

*&-----*
*&      Form  free_control_ressources
*&-----*
*      Free all control related ressources.
*-----*
FORM free_control_ressources.
    CALL METHOD ref_pic_left->free.
    CALL METHOD ref_pic_right->free.
    CALL METHOD ref_splitter->free.
    CALL METHOD ref_container->free.
    FREE: ref_pic_left,
          ref_pic_right,
          cell_1_1,
          cell_1_2,
          ref_splitter,
          ref_container.
ENDFORM.           " free_control_ressources

```



## Lesson Summary

You should now be able to:

- Describe how context menus work with EnjoySAP Controls
- Create context menus for EnjoySAP Controls
- Process functions selected in context menus



## Unit Summary

You should now be able to:

- Describe how context menus work with EnjoySAP Controls
- Create context menus for EnjoySAP Controls
- Process functions selected in context menus



# Unit 5

## Text Edit Control

### Unit Overview

This is the first unit which covers only one single EnjoySAP Control. Everything learned so far is valid for all following controls.



### Unit Objectives

After completing this unit, you will be able to:

- Exchange data with the Text Edit Control
- Adapt the local context menu of the Text Edit Control to your own application
- Process functions selected in context menus

### Unit Contents

Lesson: Text Edit Control .....	234
Exercise 9: Text Edit Control.....	247

# Lesson: Text Edit Control

## Lesson Overview

The Lesson presents how to exchange data with the Text Edit Control, adapt the local Context Menu of the Text Edit Control to your own application and processing functions selected in the Context Menus.



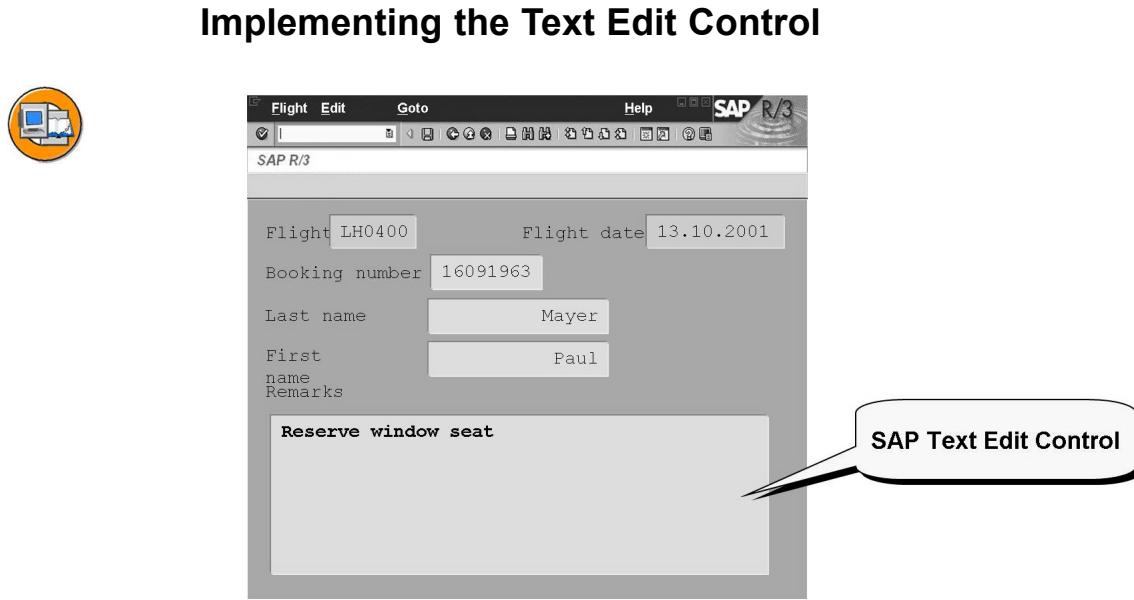
## Lesson Objectives

After completing this lesson, you will be able to:

- Exchange data with the Text Edit Control
- Adapt the local context menu of the Text Edit Control to your own application
- Process functions selected in context menus

## Business Example

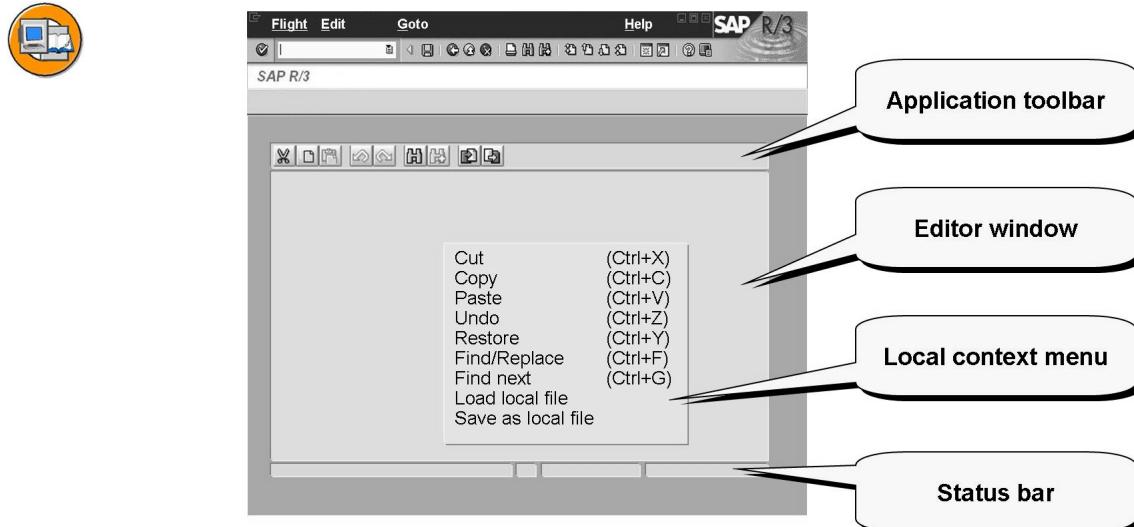
You need to implement an application using the Text Edit Control. You want to exchange data with the Text Edit Control and process functions selected in the context menu.



**Figure 98: Implementing the Text Edit Control**

- The SAP Text Edit Control allows you implement an Editor, used to enter and edit pure text. It does not allow you to assign formatting information or include graphics.
- You can use **temporary** additions to apply additional display options, such as highlighted text. Temporary in this sense means that the additions are set and managed on the presentation server, but not passed back to the controller.

## Characteristics of the Text Edit Control



**Figure 99: Characteristics of the Text Edit Control**

The SAP Text Edit Control has three parts:

- An application toolbar containing predefined pushbuttons
- The editor window for displaying text
- The status bar

The application toolbar offers several functions for processing text, such as Find/Replace, Undo, Restore, and Save as local file

These functions are provided in the form of a local context menu.

The user can also edit the text using keyboard shortcuts.

The status bar can be used to display text messages, provide information on the selected text area, read or set the current cursor position, or toggle between the Overwrite and Insert modes.

The user can hide the application tool bar and status bar.

## Creating a Text Edit Control

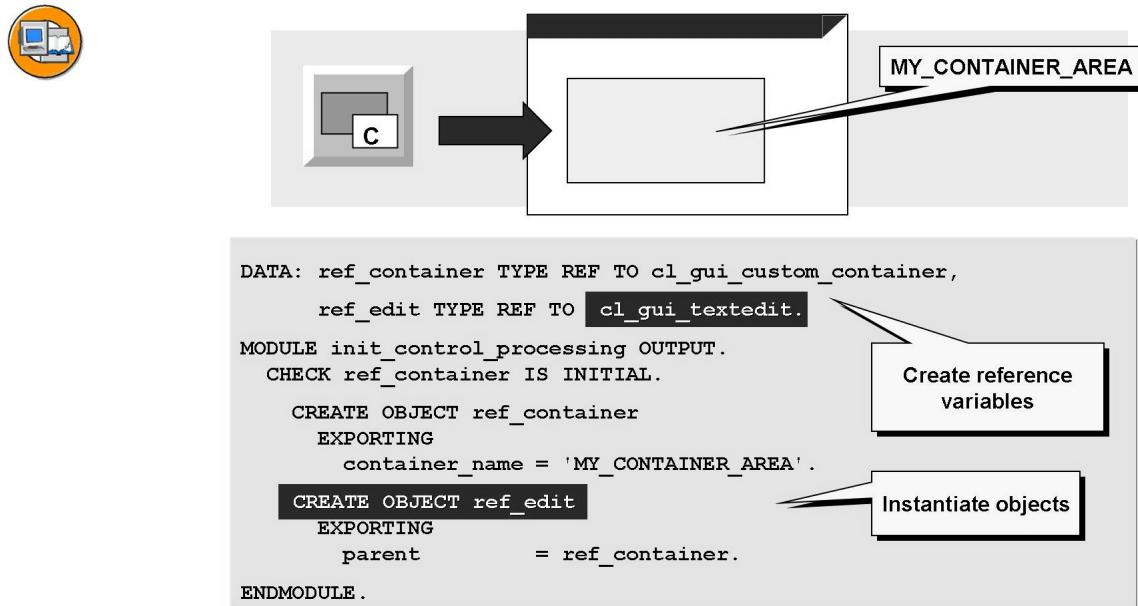


Figure 100: Creating a Text Edit Control

To create a Text Edit Control in a Custom Container:

**Reference variables:** Define two reference variables, one for the container (... TYPE REF TO cl\_gui\_custom\_container) and one for the Editor (... TYPE REF TO cl\_gui\_textedit).

**Screen area:** Reserve and name an appropriate area for your Custom Container on the screen.

**Control instances:** Create instances of the Container and Text Edit Controls using their associated constructors.

## Data for the Text Edit Control

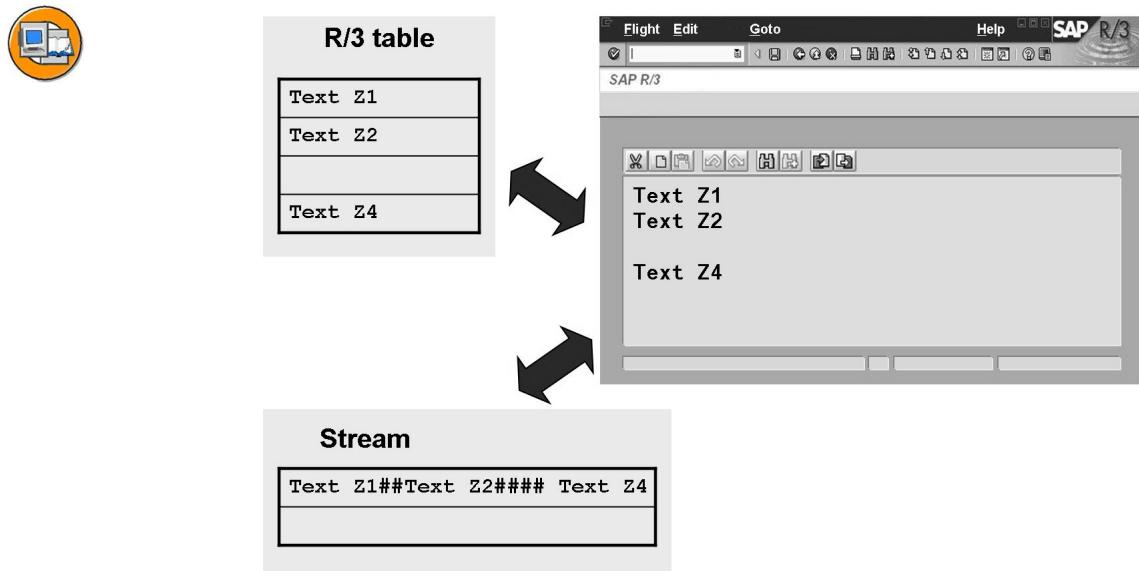
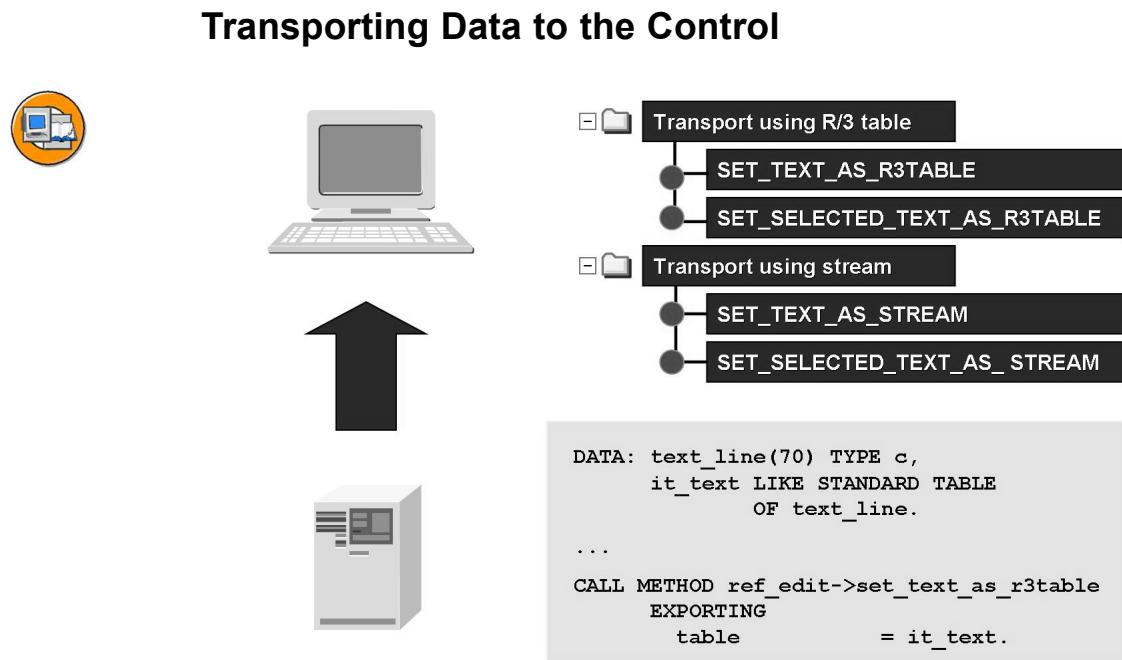


Figure 101: Data for the Text Edit Control

You can exchange data between the ABAP program and the Text Edit Control in one of two ways:

**R/3 table:** The data is held in the ABAP program as an **internal table**. Line breaks in the Text Edit Control cause a new line to be created in the internal table and vice versa.

**Stream:** Data is again held as an **internal table** in the ABAP program, but transported as a **data stream**. Line breaks in the Text Edit Control cause control characters to be inserted in the internal table and vice versa.



**Figure 102: Transporting Data to the Control**

You can set text from an internal table using the `SET_TEXT_AS_R3TABLE` method. This method overwrites any existing text. The text is transported **without** the line break information.

You can set text from an internal table in the form of a data stream using the `SET_TEXT_AS_STREAM` method. This method overwrites any existing text. The text is transported **with** line break information, which is stored in the data stream.

The `SET_SELECTED_TEXT_AS_R3TABLE` and `SET_SELECTED_TEXT_AS_STREAM` methods are similar to those outlined above, except that the text is inserted wherever the cursor is positioned and is not completely overwritten. Any selected text is overwritten.

## Transporting Data from the Control

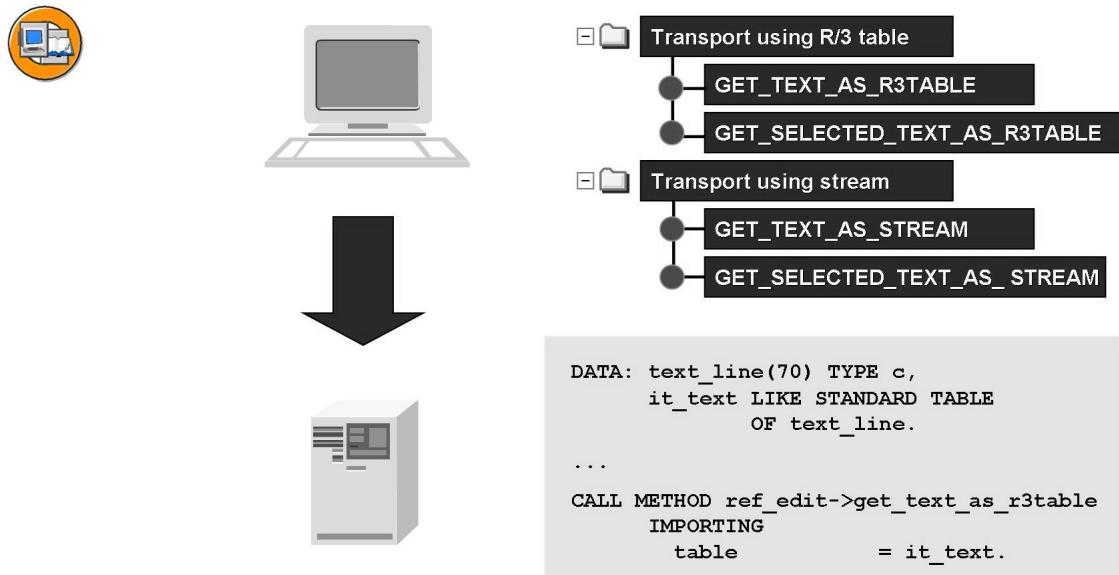


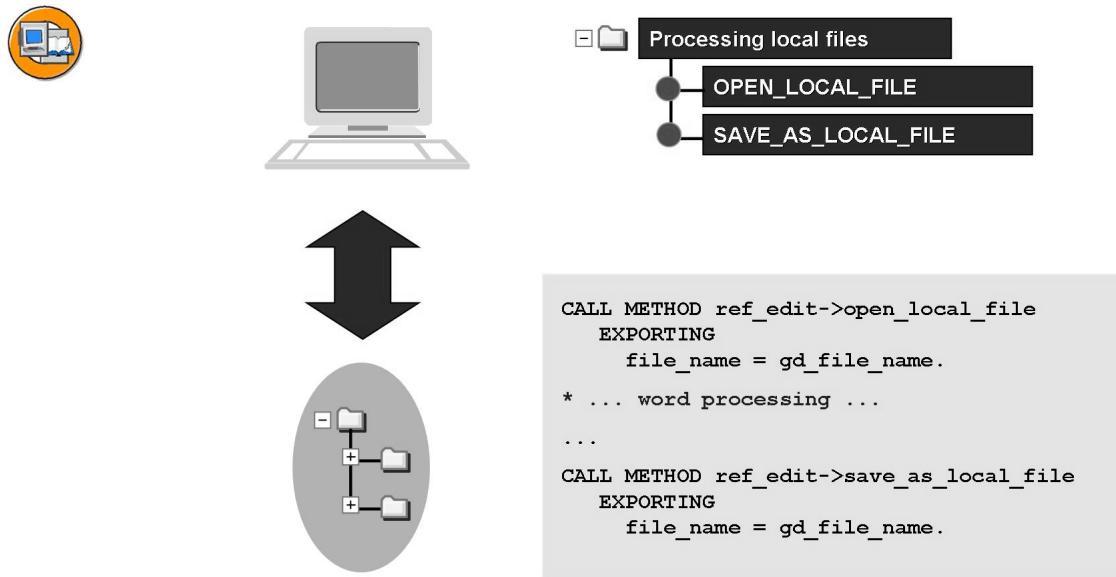
Figure 103: Transporting Data from the Control

To process the edited texts in the ABAP program, you must first transport the data to this program, since all entries are initially made on the presentation server.

Use one of the following methods to do this:

- **GET\_TEXT\_AS\_R3TABLE:** This method transports the **entire** text to the Text Editor without line break information. Note the behavior for handling line breaks defined in the constructor, particularly for soft line breaks. If the exception **POTENTIAL\_DATA\_LOSS** is raised, use **GET\_TEXT\_AS\_STREAM** instead.
- **GET\_SELECTED\_TEXT\_AS\_R3TABLE:** This method transports the **selected** text without line break information. If you use an internal table whose line length is shorter than that defined in the control, the text is truncated and lost from that point on. If the line length of the table is longer than that of the control, the line is padded with space characters.
- **GET\_TEXT\_AS\_STREAM:** This method transports the **entire** text to the Text Editor as a data stream with line break information.
- **GET\_SELECTED\_TEXT\_AS\_STREAM:** This method transports the **selected** text to the Text Editor as a data stream with line break information.

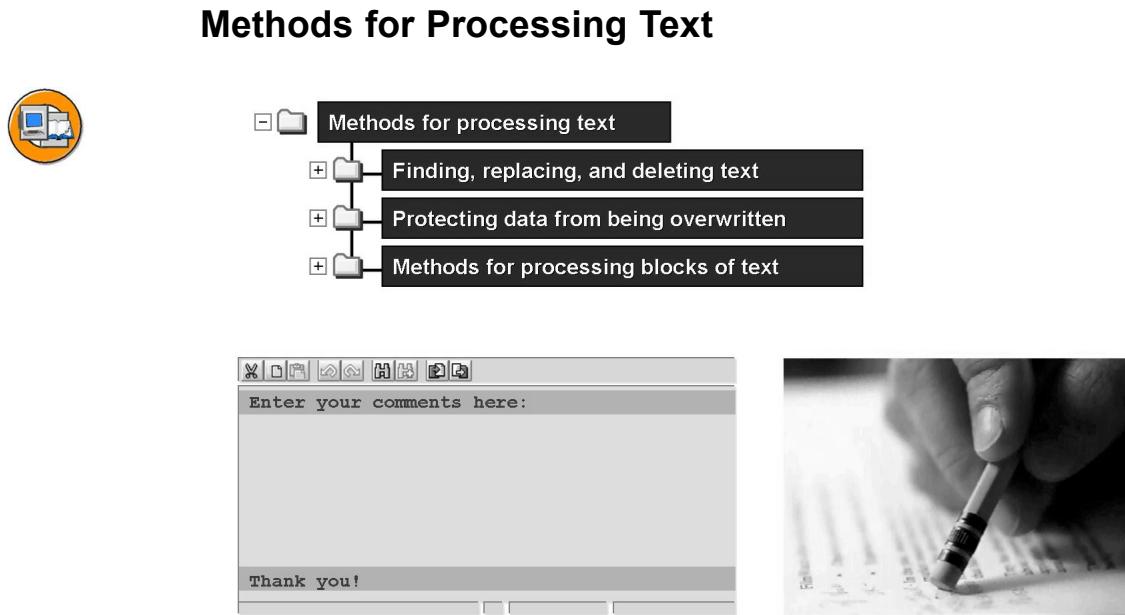
## Processing Local Text Files



**Figure 104: Processing Local Text Files**

You can open a local file using the OPEN\_LOCAL\_FILE method. If you do not fill the FILE\_NAME parameter, a dialog box appears, where the user enters the file name and path.

Similarly, you can use the SAVE\_AS\_LOCAL\_FILE method to save text in a local file. If you do not fill the FILE\_NAME parameter, a dialog box appears, where the user enters the file name and path.



**Figure 105: Methods for Processing Text**

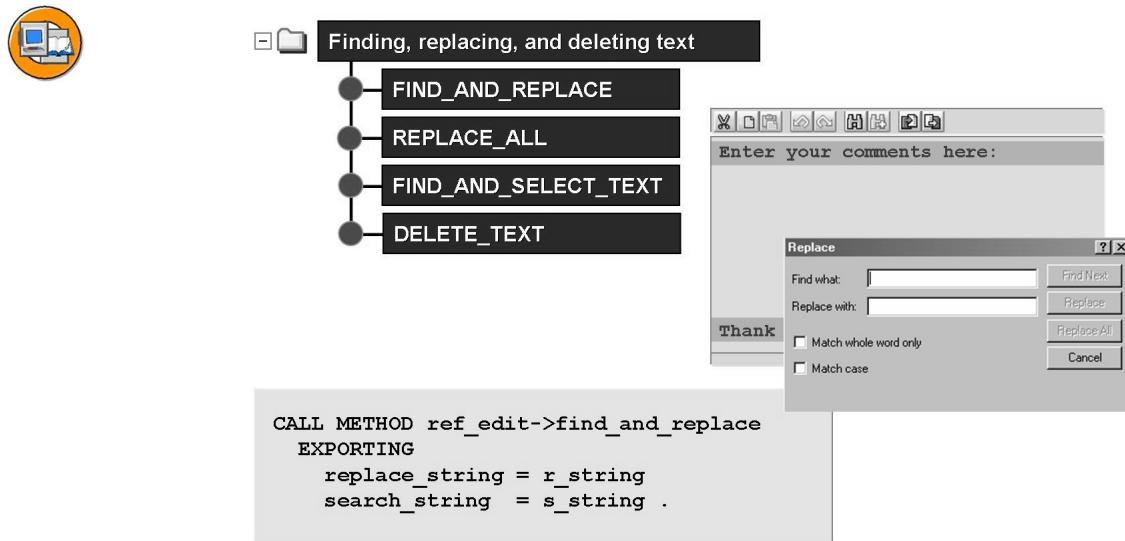
You can generally provide methods in the local context menu for processing text. The appropriate function is then carried out locally on the presentation server.

However, to process the results further in your ABAP program, the user must transport the data or save them as a local file.

You can, however, also use these methods to trigger your own program-controlled text processing functions.

Static constants have been defined for the Text Edit Control to allow you to activate or deactivate functions. Use the class constants `cl_gui_textedit=>true` and `cl_gui_textedit=>false`.

## Finding, Replacing, and Deleting Text



**Figure 106: Finding, Replacing, and Deleting Text**

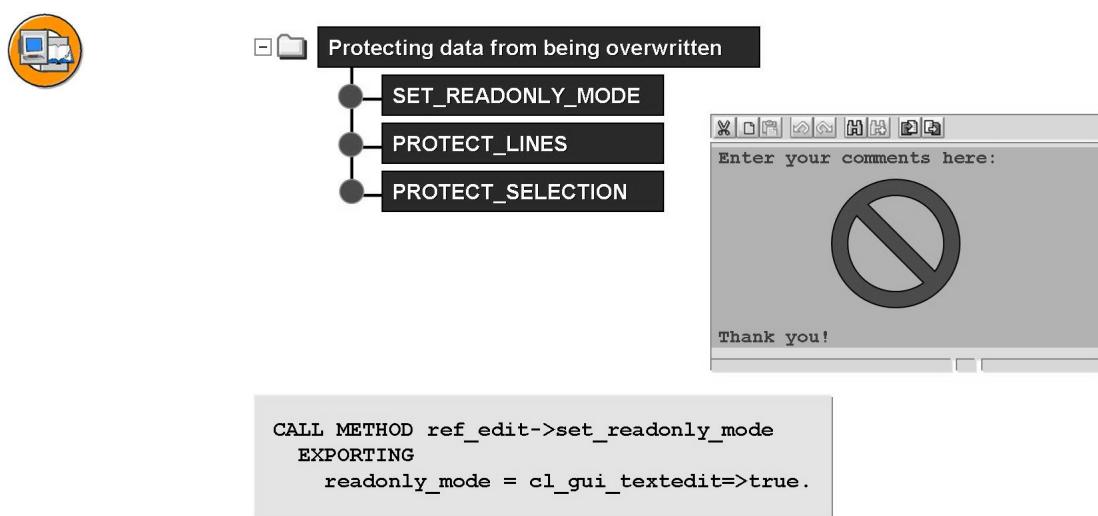
The FIND\_AND\_REPLACE method allows users to find and replace text.

You can replace all occurrences of a text at once using the REPLACE\_ALL method.

To find and select text, use the FIND\_AND\_SELECT\_TEXT method.

You can delete all the text in the Editor using the DELETE\_TEXT method.

## Protecting Data from Being Overwritten



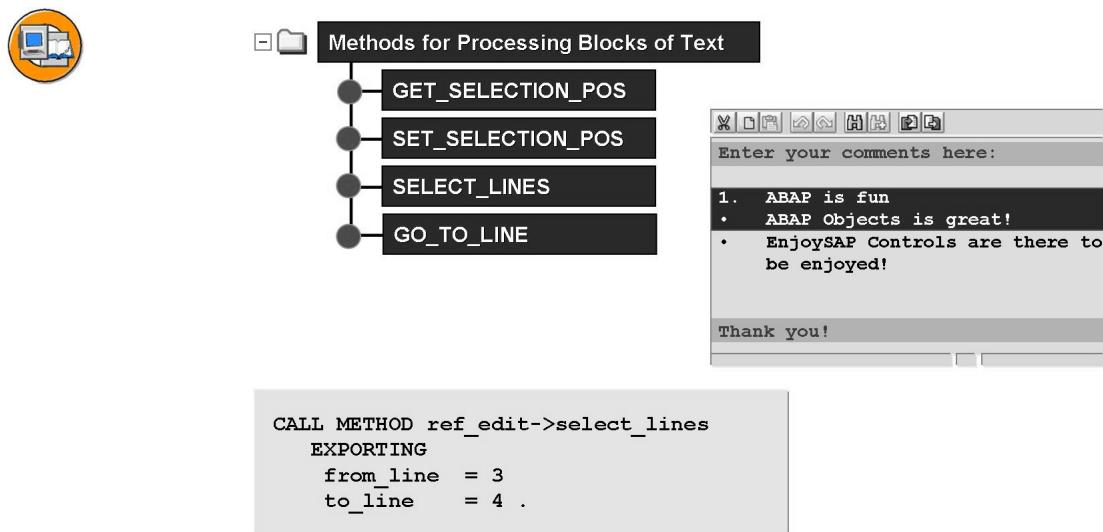
**Figure 107: Protecting Data from Being Overwritten**

In the Text Edit Control, you can protect text from being overwritten by users—either single lines, or the entire text.

If a line is protected against user entries, the background color is changed. If the user tries to enter a text, the system makes a warning sound.

You can use this function, for example to lock parts of a form against user input, or display longer texts on screen.

## Methods for Processing Blocks of Text



**Figure 108: Methods for Processing Blocks of Text**

Using the text block processing methods, you can select blocks in the text (controlled by the program), find out what text the user has selected, or navigate within the text. These functions are available either from the local context menu or the application toolbar.

The selected texts can then be overwritten; commented out or “de-commented” (in an ABAP syntax sense), or transported to the ABAP program.

## Keyboard Commands



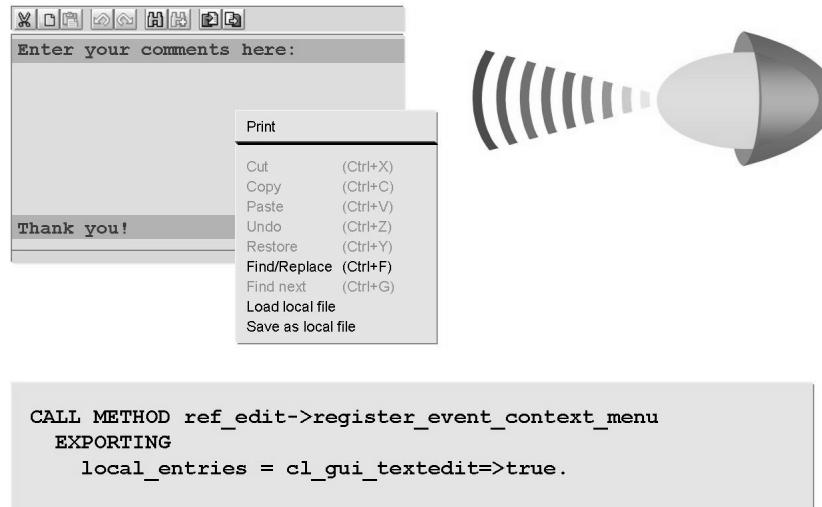
<b>Ins</b>	Toggle between insert and overwrite mode
<b>PgUp/PgDn</b>	Scroll up or down page by page
<b>Ctrl + Z</b>	Undo
<b>Ctrl + Y</b>	Restore
<b>Ctrl + F</b>	Find and replace
<b>Ctrl + G</b>	Find next
<b>Ctrl + A</b>	Select all
<b>Ctrl + End</b>	Select to end of line

**Figure 109: Keyboard Commands**

The above graphic shows the commands most commonly needed for processing texts.

For a complete list of all keyboard combinations and mouse operations, see the online documentation for the SAP Text Edit control.

## Local Context Menu



**Figure 110: Local Context Menu**

The SAP Text Edit Control automatically generates a local context menu.

You do not need to implement handler methods for the CONTEXT\_MENU and CONTEXT\_MENU\_SELECTED events, **unless** you want to offer the user **additional** functions.

To register additional events with the Automation Controller, use the REGISTER\_EVENT\_CONTEXT\_MENU method provided by the Text Edit Control. When calling methods you can use the LOCAL\_ENTRIES parameter to control whether the program displays the default menu (cl\_gui\_textedit=>true) or not (cl\_gui\_textedit=>false).

## Editing the Context Menu



```
CLASS lcl_event_handler DEFINITION.  
  PUBLIC SECTION.  
    CLASS-METHODS: on_ctmenu_selected  
      FOR EVENT context_menu_selected  
      OF cl_gui_textedit  
      IMPORTING fcode.  
  ENDCLASS.  
  
CLASS lcl_event_handler IMPLEMENTATION.  
  METHOD on_ctmenu_selected.  
    CASE fcode.  
      WHEN 'FC1'.  
      ...  
      WHEN 'FC2'.  
      ...  
    ENDCASE.  
  ENDMETHOD.  
ENDCLASS.
```

Static method  
on\_ctmenu\_selected  
imports the function  
code fcode

Case distinction for  
the function chosen  
by the user

Figure 111: Editing the Context Menu

To handle your own functions in the local context menu, write a handler method for the CONTEXT\_MENU\_SELECTED event. The method registers the REGISTER\_EVENT\_CONTEXT\_MENU event with the Automation Controller.

The function code for the function chosen by the user is passed to the handler method using the FCODE parameter.



# Exercise 9: Text Edit Control

## Exercise Objectives

After completing this exercise, you will be able to:

- Use the SAP Text Edit Control

## Business Example

Copy a template consisting of an ABAP program that calls a screen. The program displays a picture on the screen. Allow the user to store a description of the picture

**Program:** ZBC412\_##\_TEC\_EX1

**Template:** SAPBC412\_TECT\_EXERCISE1

**Model solution:** SAPBC412\_TECS\_EXERCISE1

where ## is the group number.

### Task 1:

Copy the template.

1. Copy the template SAPBC412\_TECT\_EXERCISE1 (including all its sub-objects) to the program **ZBC412\_##\_TEC\_EX1** and get to know the features of this copy. This copy template has the same features as the model solution for the last exercise in Chapter “Introduction to the Control Framework”.

### Task 2:

Create a Custom Container Control and a Text Edit Control

1. On screen 100, create a custom control area with the following attributes

Attribute	Value	Suggested value
Element type	Custom control	
Last name	CONTROL_AREA2	
Row	Any	15
columns	Any	2
defLength/ visLength	Any	85
Height	Any	9

*Continued on next page*

Attribute	Value	Suggested value
Resizing	yes	
Rows (minimum)	Any	4
Columns (minimum)	Any	42

Activate the screen.

2. Create a reference variable for the container instance (We suggest you use the name `ref_container_tec`). Use the global class type `cl_gui_custom_container`
3. Screen 100 already has a PBO module `init_control_processing`. Extend this module so that the system generates a container control instance when it executes the program for the first time. Pass the name of the custom control area you created to the interface parameter `container_name`. Catch the generic exception, `OTHERS`, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing, using the termination **message 010**.
4. Create a reference variable for the Text Edit Control instance (We suggest you use the name `ref_edit`). Use the global class `cl_gui_textedit`
5. Extend the `init_control_processing` module so that the system generates a Text Edit Control instance when it executes the program for the first time. Pass your container control reference to the interface parameter `parent`. Catch the generic exception, `OTHERS`, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing, using the **termination message 040**
6. Test your program.

### Task 3:

Hide the application toolbar and status bar of the Text Edit Control

1. Use the instance methods of the `set_toolbar_mode` and `set_statusbar_mode` of the class `cl_gui_textedit`. Extend the `init_control_processing` module
2. When an exception occurs, make sure your program stops processing, using the **termination message 012**
3. Test your program.

*Continued on next page*

## Task 4:

Transport an internal table with data to the Text Edit Control.

1. Create an internal standard table (suggested name: *it\_text*) and a corresponding work area (we suggest the name *text\_line*, of type *c* and length 85).
2. Fill the internal table with a single line (we suggest the text “Description of Picture”). Use a text symbol, to ensure that your program is multi-lingual
3. To transport the data, use the instance method *set\_text\_as\_r3table* of the class *cl\_gui\_textedit*
4. When an exception occurs, make sure your program stops processing, using the **termination message 012**
5. Test your program. You can use the local context menu of the Text Edit Control to save the description locally to the hard disk

## Solution 9: Text Edit Control

### Task 1:

Copy the template.

1. Copy the template SAPBC412\_TECT\_EXERCISE1 (including all its sub-objects) to the program **ZBC412 ## TEC EX1** and get to know the features of this copy. This copy template has the same features as the model solution for the last exercise in Chapter “Introduction to the Control Framework”.
  - a) –

### Task 2:

Create a Custom Container Control and a Text Edit Control

1. On screen 100, create a custom control area with the following attributes

Attribute	Value	Suggested value
Element type	Custom control	
Last name	CONTROL_AREA2	
Row	Any	15
columns	Any	2
defLength/ visLength	Any	85
Height	Any	9
Resizing	yes	
Rows (minimum)	Any	4
Columns (minimum)	Any	42

Activate the screen.

- a) –

*Continued on next page*

2. Create a reference variable for the container instance (We suggest you use the name `ref_container_tec`). Use the global class type `cl_gui_custom_container`

a)

```
DATA:  
...  
    ref_container_tec TYPE REF TO cl_gui_custom_container.
```

3. Screen 100 already has a PBO module `init_control_processing`. Extend this module so that the system generates a container control instance when it executes the program for the first time. Pass the name of the custom control area you created to the interface parameter `container_name`. Catch the generic exception, `OTHERS`, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing, using the termination message **010**.

a)

```
*      create container object and link to screen area  
CREATE OBJECT ref_container_tec  
    EXPORTING  
        container_name          = 'CONTROL_AREA2'  
    EXCEPTIONS  
        others                  = 6.  
    IF sy-subrc NE 0.  
        MESSAGE a010.           " cancel program processing  
    ENDIF.
```

4. Create a reference variable for the Text Edit Control instance (We suggest you use the name `ref_edit`). Use the global class `cl_gui_textedit`

a)

```
DATA:  
...  
    ref_edit      TYPE REF TO cl_gui_textedit.
```

5. Extend the `init_control_processing` module so that the system generates a Text Edit Control instance when it executes the program for the first time. Pass your container control reference to the interface parameter `parent`.

*Continued on next page*

Catch the generic exception, *OTHERS*, instead of evaluating each exception individually. When an exception occurs, make sure your program stops processing, using the **termination message 040**

a)

```
IF ref_container IS INITIAL.          " prevent re-processing on ENTER
...
*      create Text Edit object and link to container
CREATE OBJECT ref_edit
  EXPORTING
    parent           = ref_container_tec
  EXCEPTIONS
    others          = 6      .
IF sy-subrc NE 0.
  MESSAGE a040.          " cancel program processing
ENDIF.
...
ENDIF.
```

6. Test your program.

a) –

### Task 3:

Hide the application toolbar and status bar of the Text Edit Control

1. Use the instance methods of the *set\_toolbar\_mode* and *set\_statusbar\_mode* of the class *cl\_gui\_textedit*. Extend the *init\_control\_processing* module

a) –

*Continued on next page*

2. When an exception occurs, make sure your program stops processing, using the **termination message 012**

a)

```

*      suppress display of the Text Edit Control toolbar
*      and the statusline
CALL METHOD ref_edit->set_toolbar_mode
EXCEPTIONS
  OTHERS          = 3 .
IF sy-subrc NE 0.
  MESSAGE a012.
ENDIF.
CALL METHOD ref_edit->set_statusbar_mode
EXCEPTIONS
  OTHERS          = 3 .
IF sy-subrc NE 0.
  MESSAGE a012.
ENDIF.

```

3. Test your program.

a) -

### Task 4:

Transport an internal table with data to the Text Edit Control.

1. Create an internal standard table (suggested name: *it\_text*) and a corresponding work area (we suggest the name *text\_line*, of type *c* and length 85).

a)

```

DATA:
...
text_line(85),                      "work area
it_text LIKE STANDARD TABLE OF text_line. "text table

```

*Continued on next page*

2. Fill the internal table with a single line (we suggest the text “Description of Picture”). Use a text symbol, to ensure that your program is multi-lingual

a)

```
*      fill text into a table and load the table into
*      Text Edit Control
text_line = 'Description of Picture:'(t11).
APPEND text_line TO it_text.
```

3. To transport the data, use the instance method *set\_text\_as\_r3table* of the class *cl\_gui\_textedit*

a) –

4. When an exception occurs, make sure your program stops processing, using the **termination message 012**

a)

```
CALL METHOD ref_edit->set_text_as_r3table
      EXPORTING
        table                  = it_text
      EXCEPTIONS
        OTHERS                 = 3.
      IF sy-subrc NE 0.
      MESSAGE a012.
    ENDIF.
```

5. Test your program. You can use the local context menu of the Text Edit Control to save the description locally to the hard disk

a) –

## Result

### Screen flow logic SCREEN 100

```
PROCESS BEFORE OUTPUT.
MODULE status_0100.
MODULE init_control_processing.

PROCESS AFTER INPUT.
MODULE exit_command_0100 AT EXIT-COMMAND.
MODULE user_command_0100.
```

### ABAP Program Data Declarations

*Continued on next page*

```

REPORT  sapbc412_tecs_exercise1 MESSAGE-ID bc412.

*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
*      Definition of a local class containing event handler methods  *
*      for picture control objects                                     *
*-----*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS: on_picture_click
      FOR EVENT picture_click OF cl_gui_picture
      IMPORTING mouse_pos_x mouse_pos_y.
  ENDCLASS.

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
*      Corresponding class implementation                         *
*-----*
CLASS lcl_event_handler IMPLEMENTATION.
  METHOD on_picture_click.
    MESSAGE i016 WITH mouse_pos_x mouse_pos_y.
  ENDMETHOD.
ENDCLASS.

* type pools
TYPE-POOLS: cntl.

* data types
TYPES:
  t_url          TYPE bapiuri-uri.

* data declarations
*   screen specific
DATA:
  ok_code        TYPE sy-ucomm,           " command field
  copy_ok_code  LIKE ok_code,            " copy of ok_code
  l_answer       TYPE c,                 " return flag (used in
                                         " standard user dialogs)

*   control specific: object references
  ref_container  TYPE REF TO cl_gui_custom_container,
  ref_picture    TYPE REF TO cl_gui_picture,
  ref_container_tec TYPE REF TO cl_gui_custom_container,

```

*Continued on next page*

```

ref_edit          TYPE REF TO cl_gui_textedit,
* control specific: auxiliary fields
l_url           TYPE t_url,           " URL of picture to be shown
current_mode    LIKE cl_gui_picture=>display_mode,
* event handling
it_events        TYPE cntl_simple_events, " internal (event) table
wa_events        LIKE LINE OF it_events,   " work area

* Text Table for Text Edit Control
text_line(85),           "work area
it_text LIKE STANDARD TABLE OF text_line. "text table

```

### ABAP Program: Event Blocks

```

* start of main program
START-OF-SELECTION.

CALL FUNCTION 'BC412_BDS_GET_PIC_URL' festch URL of first picture
*      EXPORTING                                " from BDS
*          NUMBER      = 1
          IMPORTING
          url        = l_url
          EXCEPTIONS
          OTHERS      = 1.

IF sy-subrc <> 0.                      " no picture --> end of program
MESSAGE i035(bc412) WITH 1.
LEAVE PROGRAM.
ENDIF.

CALL SCREEN 100.                         " container screen for SAP-Enjoy
                                         " controls

```

### Modules

```

MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.          " prevent re-processing on ENTER
*      create container object and link to screen area
  CREATE OBJECT ref_container

```

*Continued on next page*

```

EXPORTING
  container_name = 'CONTROL_AREA1'
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
  MESSAGE a010.                      " cancel program processing
ENDIF.

*      create picture control and link to container object
CREATE OBJECT ref_picture
EXPORTING
  parent = ref_container
EXCEPTIONS
  others = 1.

IF sy-subrc NE 0.
  MESSAGE a011.                      " cancel program processing
ENDIF.

*      load picture into picture control
CALL METHOD ref_picture->load_picture_from_url
EXPORTING
  url    = l_url
EXCEPTIONS
  OTHERS = 1.

IF sy-subrc NE 0.
  MESSAGE a012.
ENDIF.

*      event handling
*      1. register events for control framework
wa_events-eventid    = cl_gui_picture->eventid_picture_click.
wa_events-appl_event = ''.
INSERT wa_events INTO TABLE it_events.

CALL METHOD ref_picture->set_registered_events
EXPORTING
  events = it_events
EXCEPTIONS
  OTHERS = 1.

IF sy-subrc NE 0.

```

*Continued on next page*

```

MESSAGE a012.
ENDIF.
*      2. set event handler for ABAP object instance: ref_picture
SET HANDLER lcl_event_handler=>on_picture_click FOR ref_picture.

*      create container object and link to screen area
CREATE OBJECT ref_container_tec
EXPORTING
  container_name          = 'CONTROL_AREA2'
EXCEPTIONS
  others                  = 6.

IF sy-subrc NE 0.
  MESSAGE a010.           " cancel program processing
ENDIF.

*      create Text Edit object and link to container
CREATE OBJECT ref_edit
EXPORTING
  parent                  = ref_container_tec
EXCEPTIONS
  others                  = 6
IF sy-subrc NE 0.
  MESSAGE a040.           " cancel program processing
ENDIF.

*      suppress display of the Text Edit Control toolbar
*      and the statusline
CALL METHOD ref_edit->set_toolbar_mode
EXCEPTIONS
  OTHERS                  = 3 .

IF sy-subrc NE 0.
  MESSAGE a012.
ENDIF.
CALL METHOD ref_edit->set_statusbar_mode
EXCEPTIONS
  OTHERS                  = 3 .

IF sy-subrc NE 0.
  MESSAGE a012.
ENDIF.
*      fill text into a table and load the table into
*      Text Edit Control

```

*Continued on next page*

```

text_line = 'Bildbeschreibung:'(t11).
APPEND text_line TO it_text.

CALL METHOD ref_edit->set_text_as_r3table
  EXPORTING
    table                  = it_text
  EXCEPTIONS
    OTHERS                 = 3.

IF sy-subrc NE 0.
  MESSAGE a012.
ENDIF.

ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING_OUTPUT

*&-----*
*&     Module  USER_COMMAND_0100  INPUT
*&-----*
*     Implementation of user commands of type ' ':
*     - push buttons of screen 100
*     - GUI functions
*-----*
MODULE user_command_0100 INPUT.
copy_ok_code = ok_code.
CLEAR ok_code.

CASE copy_ok_code.
  WHEN 'BACK'.                               " back to program, leave screen
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
      EXPORTING
        *          DEFAULTOPTION  = 'Y'
        textline1   = text-004
        textline2   = text-005
        titel       = text-007
        cancel_display = ' '
      IMPORTING
        answer      = l_answer.

CASE l_answer.
  WHEN 'J'.
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.

```

*Continued on next page*

```

WHEN 'N'.
  SET SCREEN sy-dynnr.
ENDCASE.

WHEN 'STRETCH'.          " picture operation: stretch to fit are
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_stretch
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'NORMAL'.           " picture operation: fit to normal size
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_normal
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'NORMAL_CENTER'.    " picture operation: center in normal size
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_normal_center
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'FIT'.               " picture operation: zoom picture
  CALL METHOD ref_picture->set_display_mode
    EXPORTING
      display_mode = cl_gui_picture->display_mode_fit
    EXCEPTIONS
      OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'FIT_CENTER'.        " picture operation: zoom and center
  CALL METHOD ref_picture->set_display_mode

```

*Continued on next page*

```

EXPORTING
  display_mode = cl_gui_picture->display_mode_fit_center
EXCEPTIONS
  OTHERS = 1.
  IF sy-subrc NE 0.
    MESSAGE s015.
  ENDIF.

WHEN 'MODE_INFO'.
  current_mode = ref_picture->display_mode.
  MESSAGE i025 WITH current_mode.

ENDCASE.

ENDMODULE.                                     " USER_COMMAND_0100  INPUT

*&-----*
*&     Module STATUS_0100  OUTPUT
*&-----*
*      Set GUI for screen 0100
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'STATUS_NORM_0100'.
  SET TITLEBAR 'TITLE_NORM_0100'.
ENDMODULE.                                     " STATUS_0100  OUTPUT

*&-----*
*&     Module EXIT_COMMAND_0100  INPUT
*&-----*
*      Implementation of user commands of type 'E'.
*-----*
MODULE exit_command_0100 INPUT.
  CASE ok_code.
    WHEN 'CANCEL'.           " cancel current screen processing
      CLEAR l_answer.
      CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
        EXPORTING
          *        DEFAULTOPTION = 'Y'
          textline1   = text-004
          textline2   = text-005
          titel       = text-006
          cancel_display = ' '
        IMPORTING
          answer      = l_answer.
  CASE l_answer.

```

*Continued on next page*

```

WHEN 'J'.
  PERFORM free_control_ressources.
  LEAVE TO SCREEN 0.

WHEN 'N'.
  CLEAR ok_code.
  SET SCREEN sy-dynnrr.
ENDCASE.

WHEN 'EXIT'.                      " leave program
CLEAR l_answer.
CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
  EXPORTING
    *           DEFAULTOPTION = 'Y'
    textline1   = text-001
    textline2   = text-002
    titel       = text-003
    cancel_display = 'X'
  IMPORTING
    answer      = l_answer.
CASE l_answer.
  WHEN 'J' OR 'N'.                  " no data to update
    PERFORM free_control_ressources.
    LEAVE PROGRAM.

  WHEN 'A'.
    CLEAR ok_code.
    SET SCREEN sy-dynnrr.
ENDCASE.

ENDCASE.

ENDMODULE.                         " EXIT_COMMAND_0100  INPUT

FORM free_control_ressources.
  CALL METHOD ref_picture->free.
  CALL METHOD ref_container->free.
  FREE: ref_picture, ref_container.
ENDFORM.                            " free_control_ressources

```



## Lesson Summary

You should now be able to:

- Exchange data with the Text Edit Control
- Adapt the local context menu of the Text Edit Control to your own application
- Process functions selected in context menus



## Unit Summary

You should now be able to:

- Exchange data with the Text Edit Control
- Adapt the local context menu of the Text Edit Control to your own application
- Process functions selected in context menus

# *Unit 6*

## SAP Grid Control

### **Unit Overview**

- Layout and features
- Selected methods
- Data description
- Selected events



### **Unit Objectives**

After completing this unit, you will be able to:

- Name the default functions of the SAP Grid Control
- Describe the screen layout of the SAP Grid Control
- Describe the data area layout of the SAP Grid Control
- Describe the print list layout of the SAP Grid Control
- Understand the technical view of the SAP Grid Control
- Pass list data, the field catalog and additional information to the SAP Grid Control instance
- Work with layout variants in a SAP Grid Control
- Fill and pass the layout structure of the SAP Grid Control
- Explain the purpose of the field catalogue
- Add columns to or change columns of the SAP Grid Control
- Handle colors of rows and cells in the SAP Grid Control
- Suppress standard functionality of the toolbar
- Explain how events are triggered and how they can be handled
- Identify existing events
- Create and register an event handler method
- Add new functionality to the standard toolbar
- Enhance the print list by means of event handling

### **Unit Contents**

Lesson: SAP Grid Control: Introduction .....	267
--	-----

Exercise 10: Displaying Data in a SAP Grid Control .....	279
Lesson: Transporting Data and Additional Information .....	286
Lesson: Adapting the Grid Layout .....	293
Exercise 11: Field Catalog .....	307
Lesson: Events.....	317
Exercise 12: Events of the SAP Grid Control.....	323
Exercise 13: Extending the Application Toolbar.....	335

# Lesson: SAP Grid Control: Introduction

## Lesson Overview

This lesson presents the field catalog and layout variants for the SAP Grid Control.



## Lesson Objectives

After completing this lesson, you will be able to:

- Name the default functions of the SAP Grid Control
- Describe the screen layout of the SAP Grid Control
- Describe the data area layout of the SAP Grid Control
- Describe the print list layout of the SAP Grid Control
- Understand the technical view of the SAP Grid Control

## Business Example

The users want to view flight information displayed with an SAP Grid Control. You will have to understand the different layouts available and have the technical background necessary to implement the application.

## SAP Grid Control



The screenshot shows a SAP Grid Control window. At the top is a toolbar with various icons for operations like search, sort, and export. Below the toolbar is a header row with columns labeled 'Air...', 'Flight', 'Date', 'Price', and 'Curr.'. Underneath is a data grid containing 10 rows of flight information. Each row has five columns: Airline (AA), Flight Number (17), Date (e.g., 12/27/2006), Price (422.94), and Currency (USD). The last column of each row contains small up and down arrows, likely for sorting. At the bottom of the grid are navigation buttons for navigating through the data.

Air...	Flight	Date	Price	Curr.
AA	17	12/27/2006	422.94	USD
AA	17	01/24/2007	422.94	USD
AA	17	02/21/2007	422.94	USD
AA	17	03/21/2007	422.94	USD
AA	17	04/18/2007	422.94	USD
AA	17	05/16/2007	422.94	USD
AA	17	06/13/2007	422.94	USD
AA	17	07/11/2007	422.94	USD
AA	17	08/08/2007	422.94	USD

Standard tool for displaying non-hierarchical lists

Figure 112: SAP Grid Control

The SAP Grid Control tool allows you to display and print non-hierarchical lists in a standardized format. The list data is displayed as a table on the screen.

The SAP Grid Control allows users to adapt the list layout to their own needs.

The developer can specify in the program whether users can save user-specific or global Layouts.

The SAP Grid Control has several default interactive functions that users often need when working with lists (such as Print).

As a developer, you can hide these default functions. If necessary, you can also make your implementations application-specific.

You can also add your own functions to the application toolbar.

In the default setting, the ALV Grid Control is in display mode. Users can display and evaluate data of the output table. If set to this mode, the ALV Grid Control is normally used as a reporting tool. Besides, you can also use the ALV Grid Control to add, change or delete table data. Using appropriate techniques, you can set the entire ALV Grid Control or parts of it (columns or cells) to editable. Depending on the input functions allowed, users can then add new rows, or modify and delete existing rows. With its editability feature, the ALV Grid Control presents an alternative to the Table Control. However, all methods that are related to editability of the ALV grid are not released for customer software developments (see SAP Notes 551605 and 695910).

## Default Functions

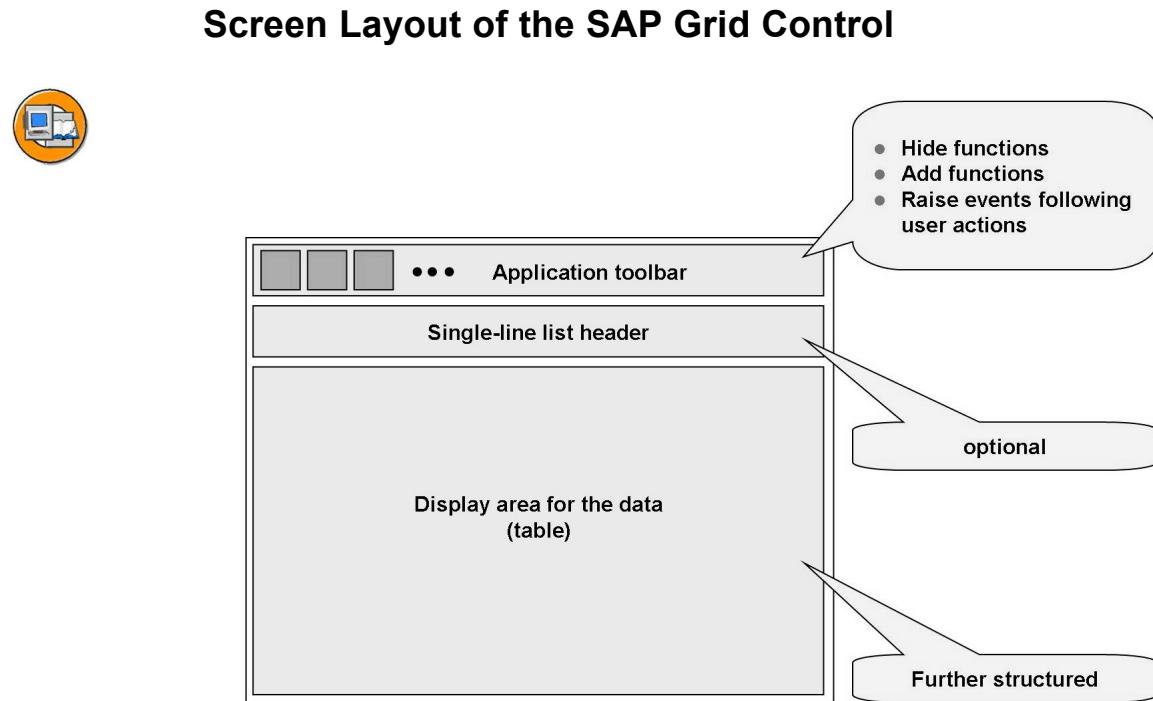


- Display details for this row in the list in a separate window
- Sort ascending (by one or more columns chosen by the user)
- Sort descending (by one or more columns chosen by the user)
- Find in the list
- Set filter (restrict data displayed using conditions chosen by the user)
- Total: Aggregation functions for the selected (numeric) columns
- Subtotal for control levels
- Print
- Views (including *Print Preview*)
- Export (HTML, spreadsheet, word processing, local file, send, and many more)
- Change layout
- Display graphic
- End User documentation

Figure 113: Default Functions

The following functions are offered as standard in the application toolbar, so that you do **not** need to implement them yourself:

- Details: Displays the data for the selected row in a modal dialog box
- Sort ascending/descending: Displays the list content sorted by the selected column or columns. Users can choose the columns they want and the order in which they will be used for sorting.
- Find: Users can specify search terms and search order (columns or rows).
- Set filter: Display only the rows that meet certain criteria chosen by the user.
- Total: User can choose to calculate the total, arithmetic mean, maximum value, or minimum value for a numeric column.
- Subtotal: Calculated over the totalized numeric column for a group of records (the user decides the grouping criteria by selecting a non-numeric column as the subtotal criterion value).
- Print: Opens the standard Print dialog box.
- Views: List Output displays a print preview.
- Export: Allows users to export the list, save it in various formats, send it as a document.
- Layout: Allows users to tailor the outputted list to their individual requirements.
- Graphic: Opens a graphic editor with the data from the list. Users can choose other processing options by double-clicking.
- End User documentation: Starts the online documentation for SAP Grid Control.



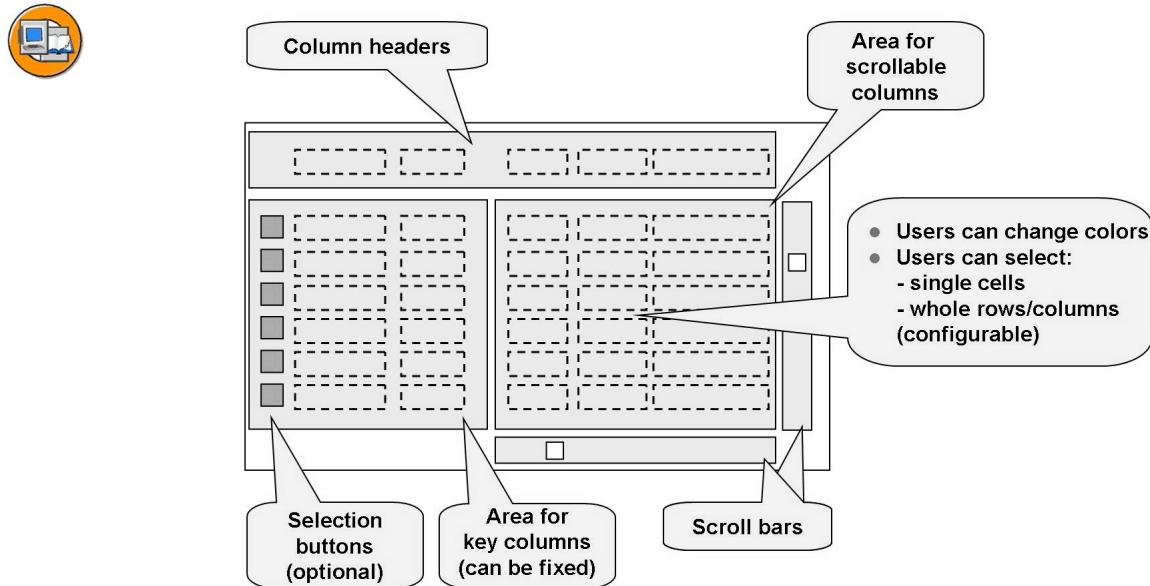
**Figure 114: Screen Layout of the SAP Grid Control**

The screen layout of the SAP Grid Control is made up of three areas:

- The application toolbar is displayed at the top of the control. The calling program can hide individual functions, or even the entire application toolbar.
- The calling program can display a single-line list header under the application toolbar.
- Under the header is the area where the data is displayed.

The data area has a structure of its own.

## Data Area Layout



**Figure 115: Data Area Layout**

The data is passed from the calling program as a table.

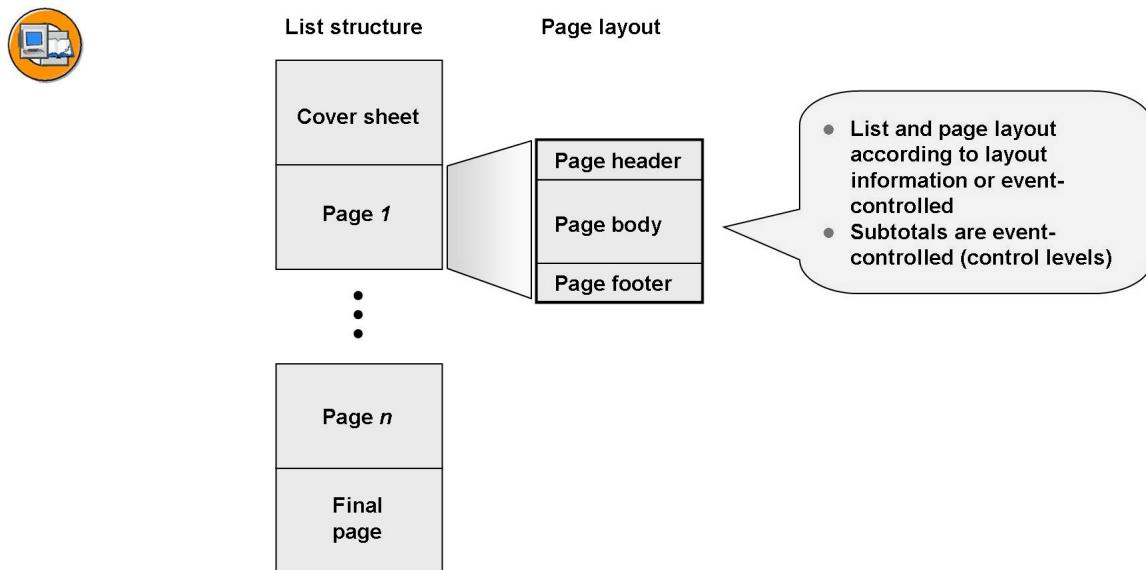
If the data table contains more rows and columns than can be displayed in the control area, horizontal and vertical scroll bars appear automatically.

You can logically group columns into key columns and non-key columns. Key columns have another color than non-key columns. They also have a fixed position and therefore remain unchanged when the user scrolls horizontally through the list. You can make any column in your data table a key column.

Users can select: Cells, groups of cells, whole rows, or whole columns. If Cell Selection or Column/Row selection is configured in the calling program, the user selects the rows through pushbuttons at the left border of the grid control.

For further information, refer to the SAP Grid Control documentation.

## Print List Layout



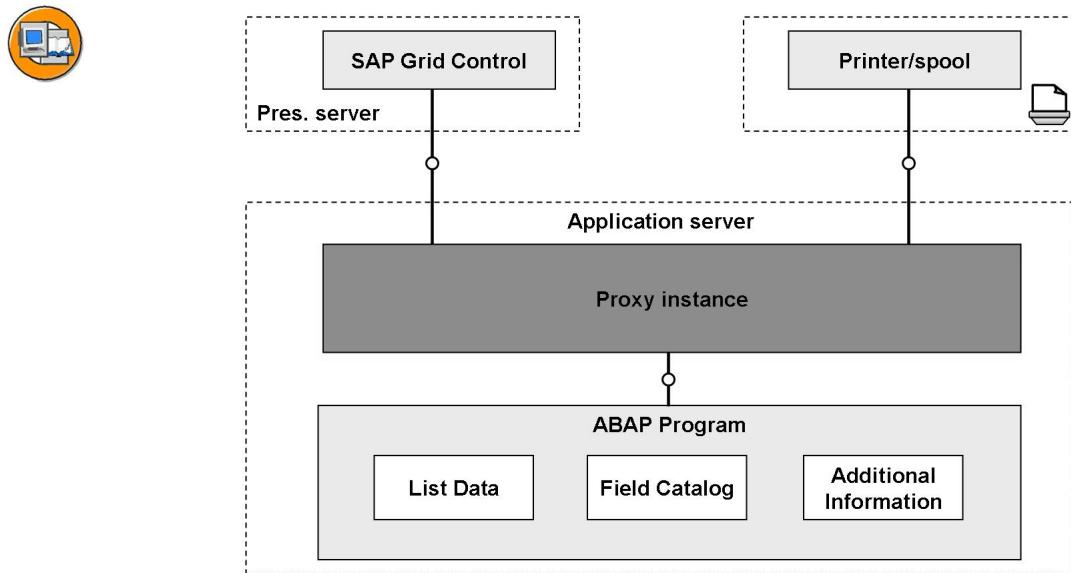
**Figure 116: Print List Layout**

The print list layout is based on the list layout of “classical” ABAP lists:

- The list can consist of a logical cover page, a logical final page and any number of other pages in between that contain the actual list data.
- In addition to the page body, each page can have a page header and a page footer.

The logical organization of the complete list into cover page, pages, final page, and the organization of the individual pages into page header, page body and page footer is made possible by events.

## The Proxy Instance of the SAP Grid Control



**Figure 117: The Proxy Instance of the SAP Grid Control**

A proxy instance of the CL\_GUI\_ALV\_GRID class encapsulates the technical details of communication with the SAP Grid Control instance on the presentation server.

The proxy object also communicates with two other partners:

- The ABAP program, as its user
- The spool system / printer: Output of the list data as an ABAP print list.

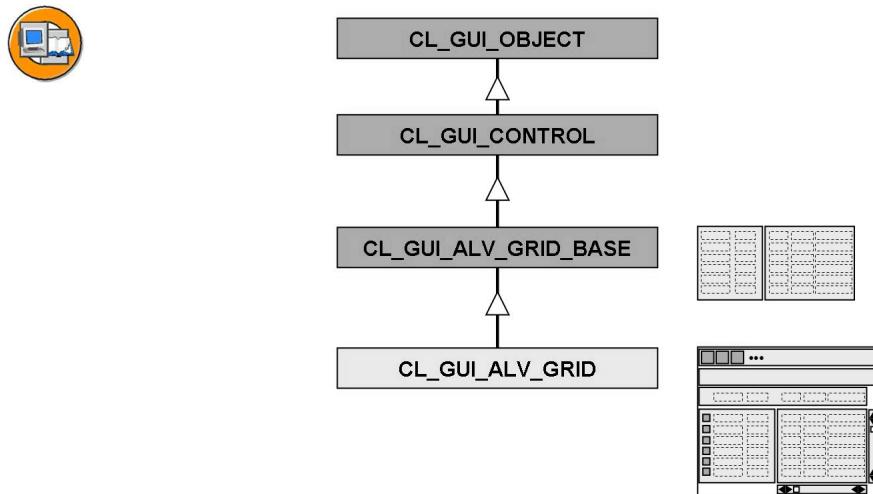
Communication through the ABAP program – Proxy Object – SAP Grid Control on the presentation server chain takes place interactively at runtime.

The standard (interactive) functions provided for displaying lists on screens and creating the ABAP print list are also implemented in the proxy class CL\_GUI\_ALV\_GRID.

Conversely, the calling program has to provide all information that cannot meaningfully be encapsulated in the proxy instance. This information includes: the data to be displayed; the rules specifying how it is displayed (field catalog); additional information, if certain user settings are to be set by the calling program. To do this, you must add special data objects to the program.

The next few slides discuss these three types of information briefly. The first two, list data and the field catalog, will then be covered in more detail later in the appropriate sections.

## The Inheritance Hierarchy of the SAP Grid Control

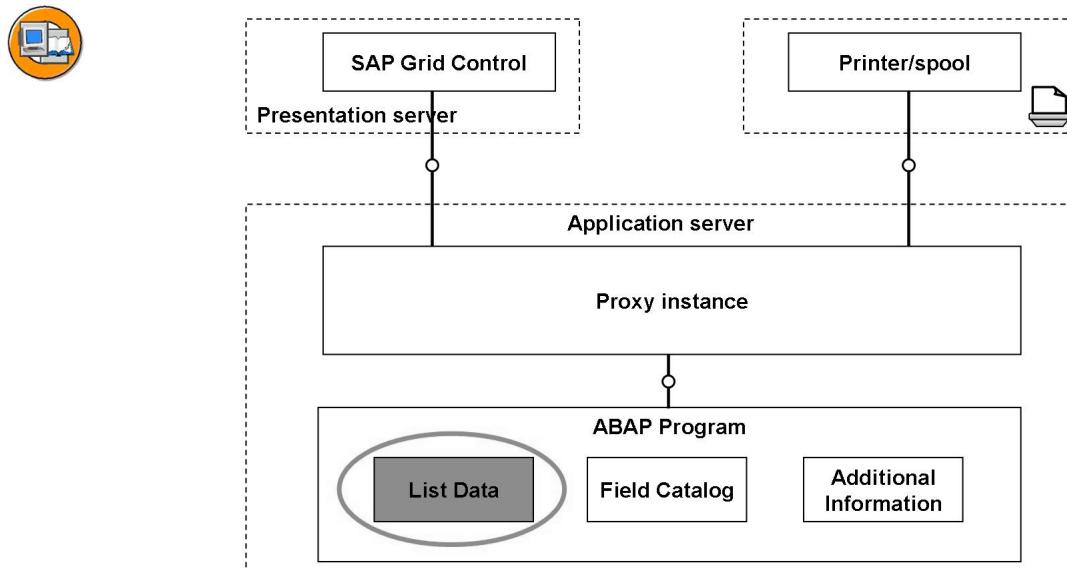


**Figure 118: The Inheritance Hierarchy of the SAP Grid Control**

The inheritance hierarchy for the class CL\_GUI\_ALV\_GRID is shown above.

As discussed already, this class encapsulates communication with the instance on the presentation server, along with many other functions. For this reason, you should instantiate this class, not its superclass.

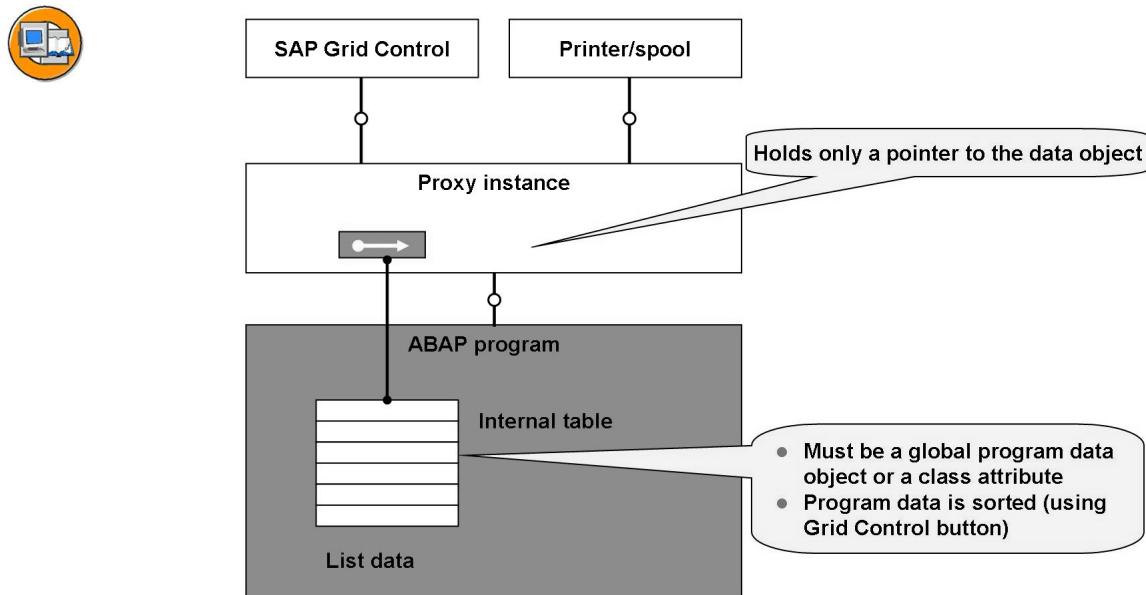
## SAP Grid Control: List Data



**Figure 119: SAP Grid Control: List Data**

List data refers to the data that is to be output on the screen or as an ABAP print list.

## Holding Data for the List



**Figure 120: Holding Data for the List**

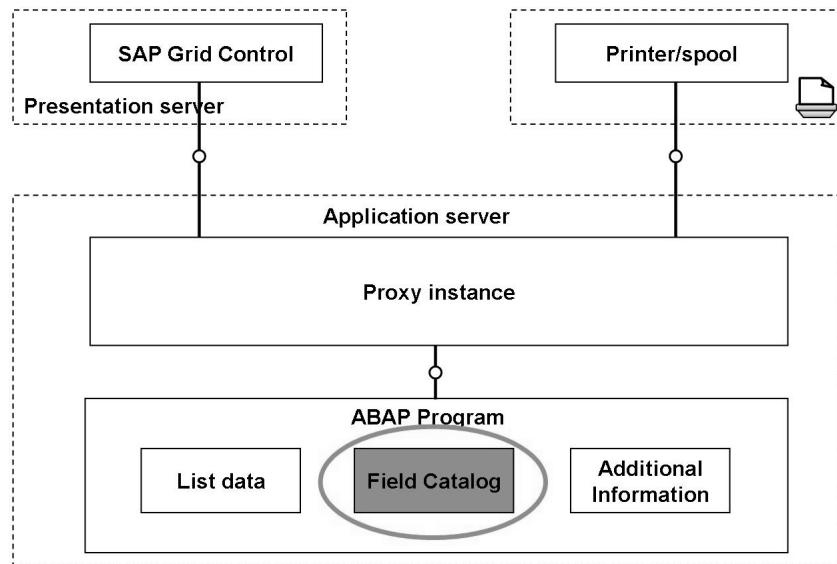
You must pass the data you want to display in an internal table (standard table) to the proxy object.

The proxy object does **not** have a copy of the data, but maintains a **reference** to the internal table passed.

To improve runtime performance, the proxy object performs all interactive actions from the presentation server control (sort, filter, and so on) on the internal table in the calling program. This means that the lifetime of the internal table containing the data must equal that of the proxy instance.

Thus, sort processes in the SAP Grid Control change the state of the internal table in the calling program. All other actions perform only read access to the data.

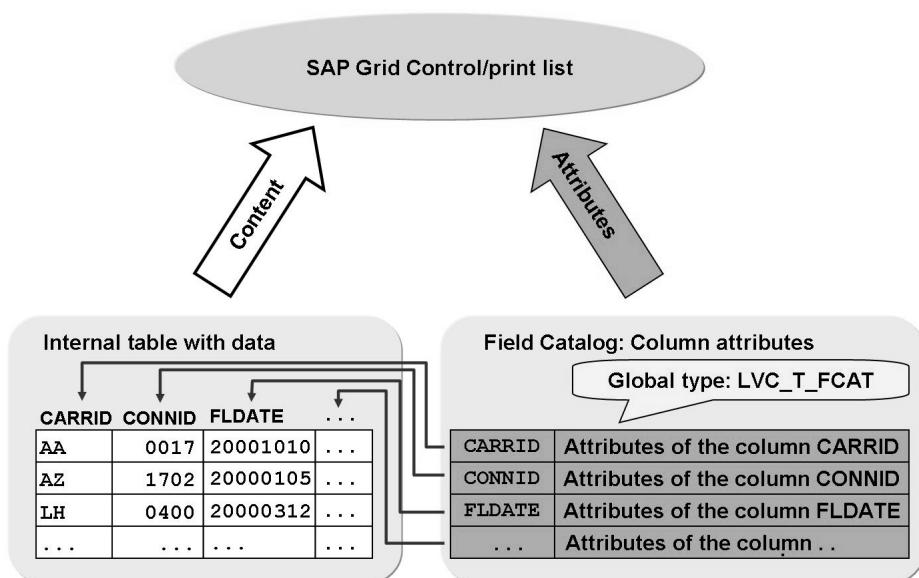
## SAP Grid Control: Field Catalog



**Figure 121: SAP Grid Control: Field Catalog**

The **field catalog** is a description of the formatting for the data display area.

## Function of the Field Catalog



**Figure 122: Function of the Field Catalog**

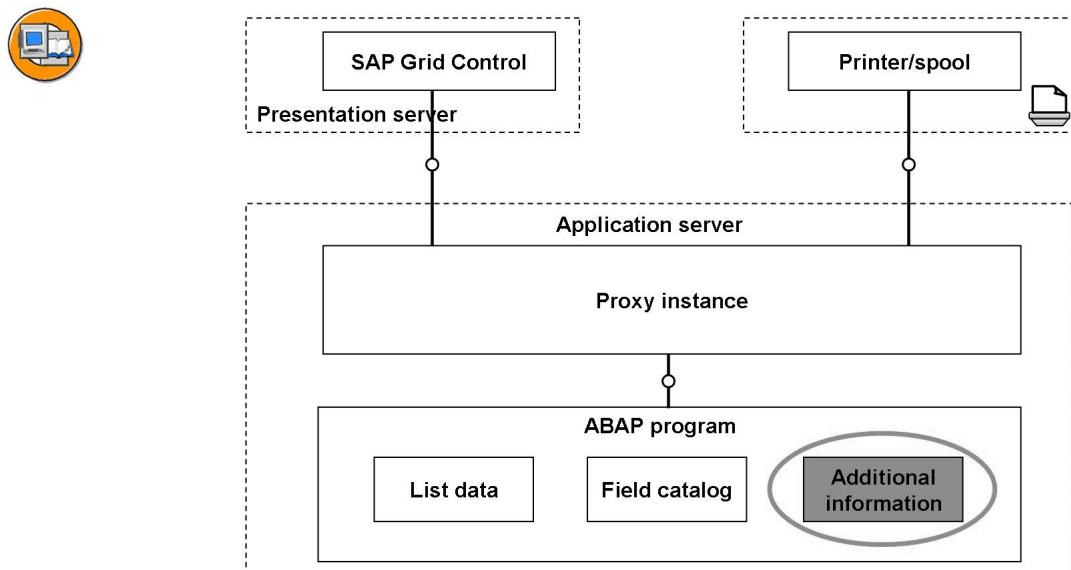
The internal table with the data to be displayed has any line type you want. So that the proxy instance can format the data passed to it for screen display or to create the print list, you need to provide the appropriate information, in the **field catalog**.

You can have the proxy instance generate the field catalog automatically. For line types defined in the ABAP Dictionary, you need only pass the name of the Dictionary structure to the proxy instance.

Alternatively however, you can pass this display information to the proxy instance using an internal table. For simplicity's sake, this **additional table** will be known from now on as the field catalog. The global data type of this internal table is LVC\_T\_FCAT. Its line type is LVC\_S\_FCAT.

For each column in the data table, the field catalog must contain a line that specifies the technical attributes and other formatting for the column.

## SAP Grid Control: Additional information

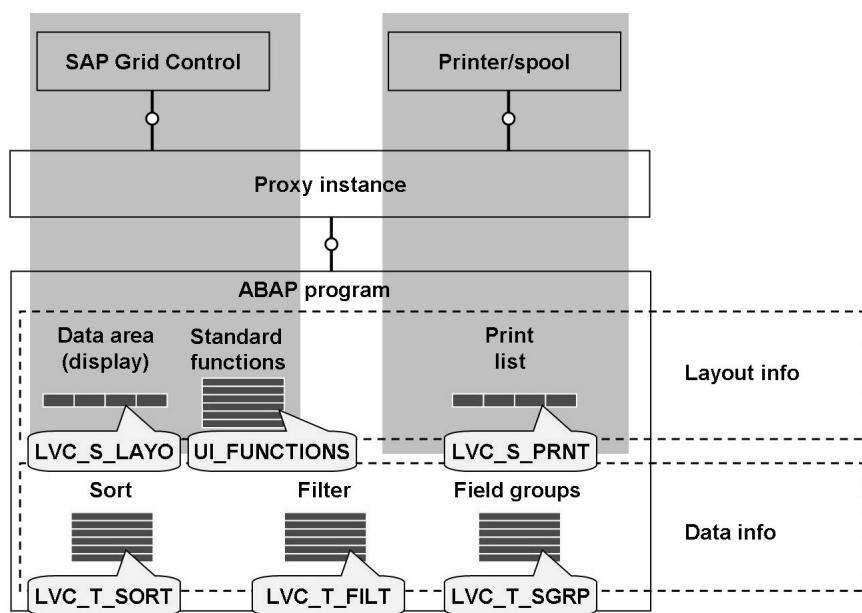


**Figure 123: SAP Grid Control: Additional information**

As well as the list data and field catalog you have the option of passing additional information to the proxy instance.



## Additional Information in Detail



**Figure 124: Additional Information in Detail**

You can define data objects in the calling program using the above global data types and use them to pass additional information to the proxy instance.

Display settings on the presentation server:

- You pass settings for the data area using a structure
- You can hide functions on the application toolbar using an internal table

You can specify output attributes for the print list using a structure.

You can pass sort criteria and filter options using an internal table

Generally, you do not create the filter and sort criteria manually, since the user can change them. The proxy instance automatically changes the associated internal tables based on the user's input. Sort and filter criteria are usually stored and saved as part of a Layout (formerly known as a display variant).

For detailed information on the structure of the internal tables used for the sort and filter criteria, refer to the online documentation.

If the table you want to display has a large number of fields, the dialog boxes where users can specify the columns they want to display will become unclear. For this reason, you can combine fields in logical groups (known as **field groups**) using the **field catalog**. You can pass texts for these field groups to the proxy instance using an internal table.

# Exercise 10: Displaying Data in a SAP Grid Control

## Exercise Objectives

After completing this exercise, you will be able to:

- Display a basic list in a SAP Grid Control instance

## Business Example

In a SAP Grid Control, display the flight connection data stored in the database table SPFLI

**Program:** ZBC412\_##\_GRD\_EX1

**Template:** SAPBC412\_GRDT\_EXERCISE\_1

**Model solution:** SAPBC412\_GRDS\_EXERCISE\_1

where ## is the group number.

### Task 1:

Copy the template.

1. Copy the model solution SAPBC412\_GRDT\_EXERCISE\_1, assigning it the new name **ZBC412\_##\_GRD\_EX1** and get to know how it works

### Task 2:

Create an instance of an SAP Grid Control, which you will display in an existing Docking Container

1. Create a reference variable for SAP Grid Control instance (We suggest you use the name *ref\_alv*).
2. In the *init\_control\_processing* module, create an SAP Grid control instance and link it with the Docking Container. If an error occurs, terminate processing and display the message 045 from the message class *BC412*

### Task 3:

Transfer the data to be displayed to the SAP Grid Control instance.

1. In your SAP Grid Control, display the connection data stored in the internal table *it\_spfli*. To do this, use the instance method *set\_table\_for\_first\_display*. If an error occurs, terminate processing and display the message 012 from the message class *BC412*

## Solution 10: Displaying Data in a SAP Grid Control

### Task 1:

Copy the template.

1. Copy the model solution SAPBC412\_GRDT\_EXERCISE\_1, assigning it the new name **ZBC412\_##\_GRD\_EX1** and get to know how it works
  - a) –

### Task 2:

Create an instance of an SAP Grid Control, which you will display in an existing Docking Container

1. Create a reference variable for SAP Grid Control instance (We suggest you use the name *ref\_alv*).
  - a)

```
DATA:  
...  
    ref_alv      TYPE REF TO cl_gui_alv_grid.
```

2. In the *init\_control\_processing* module, create an SAP Grid control instance and link it with the Docking Container. If an error occurs, terminate processing and display the message 045 from the message class *BC412*
  - a)

```
*   create alv grid object and link to container  
CREATE OBJECT ref_alv  
    EXPORTING  
        i_parent          = ref_container  
    *       I_APPL_EVENTS     = space  
    EXCEPTIONS  
        others           = 1.  
    IF sy-subrc <> 0.  
        MESSAGE a045(bc412).  
    ENDIF.
```

*Continued on next page*

### Task 3:

Transfer the data to be displayed to the SAP Grid Control instance.

1. In your SAP Grid Control, display the connection data stored in the internal table *it\_spfli*. To do this, use the instance method *set\_table\_for\_first\_display*. If an error occurs, terminate processing and display the message *012* from the message class *BC412*

a)

```

*   send basic list to alv grid control
CALL METHOD ref_alv->set_table_for_first_display
      EXPORTING
          i_structure_name          = 'SPFLI'
          *           IS_VARIANT        =
          *           I_SAVE             =
          *           I_DEFAULT          = 'X'
          *           IS_LAYOUT          =
          *           IS_PRINT            =
          *           IT_SPECIAL_GROUPS =
          *           IT_TOOLBAR_EXCLUDING =
      CHANGING
          it_outtab                = it_spfli
          *           IT_FIELDCATALOG =
          *           IT_SORT             =
          *           IT_FILTER            =
      EXCEPTIONS
          OTHERS                   = 1.

IF sy-subrc <> 0.
  MESSAGE a012(bc412).
ENDIF.

```

### Result

#### Screen flow logic-SCREEN 100

```

PROCESS BEFORE OUTPUT.
  MODULE status_0100.
  MODULE init_control_processing.

PROCESS AFTER INPUT.
  MODULE user_command_0100.

```

*Continued on next page*

## ABAP program Data Declarations

```

REPORT  sapbc412_grds_exercise_1  MESSAGE-ID bc412.

DATA:
ok_code      TYPE sy-ucomm,
copy_ok_code  LIKE ok_code,

ref_container TYPE REF TO cl_gui_docking_container,
ref_alv       TYPE REF TO cl_gui_alv_grid,

it_spfli      TYPE TABLE OF spfli,
it_sflight    TYPE TABLE OF sflight,

wa_spfli      LIKE LINE OF it_spfli,
wa_sflight    LIKE LINE OF it_sflight.

```

## Selection Screen

```

SELECT-OPTIONS:
so_carr FOR wa_spfli-carrid.

```

## ABAP Program: Event Blocks

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

```

SELECT * FROM spfli INTO TABLE it_spfli WHERE carrid IN so_carr
      ORDER BY carrid connid.
IF sy-subrc NE 0.
  MESSAGE a061(bc412).
ENDIF.

*  SELECT * FROM sflight INTO TABLE it_sflight WHERE carrid IN so_carr
*      ORDER BY carrid fldate.
*  IF sy-subrc NE 0.
*    MESSAGE a062(bc412).
*  ENDIF.

```

```
CALL SCREEN 100.
```

## Modules

*Continued on next page*

```

*&-----*
*&      Module  STATUS_0100  OUTPUT
*&-----*
*      GUI settings
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'SCREEN_0100'.
  SET TITLEBAR 'SCREEN_0100'.
ENDMODULE.                                     " STATUS_0100  OUTPUT

*&-----*
*&      Module  USER_COMMAND_0100  INPUT
*&-----*
*      implementation of GUI functions
*-----*
MODULE user_command_0100 INPUT.
  copy_ok_code = ok_code.
  CLEAR ok_code.

CASE copy_ok_code.
  WHEN 'BACK' OR 'CANCEL' OR 'EXIT'.
    LEAVE PROGRAM.
ENDCASE.
ENDMODULE.                                     " USER_COMMAND_0100  INPUT

*&-----*
*&      Module  INIT_CONTROL_PROCESSING  OUTPUT
*&-----*
*      control related processing
*-----*
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.
    CREATE OBJECT ref_container
      EXPORTING
    *      SIDE                  = DOCK_AT_LEFT
          extension            = 2000
      EXCEPTIONS
    *      others                = 1.

    IF sy-subrc <> 0.
      MESSAGE a010(bc412).
    ENDIF.

*      create alv grid object and link to container

```

*Continued on next page*

```

CREATE OBJECT ref_alv
  EXPORTING
    i_parent          = ref_container
  *     I_APPL_EVENTS   = space
  EXCEPTIONS
    others           = 1.

  IF sy-subrc <> 0.
    MESSAGE a045(bc412).
  ENDIF.

*   send basic list to alv grid control
CALL METHOD ref_alv->set_table_for_first_display
  EXPORTING
    i_structure_name      = 'SPFLI'
  *     IS_VARIANT          =
  *     I_SAVE               =
  *     I_DEFAULT             = 'X'
  *     IS_LAYOUT              =
  *     IS_PRINT              =
  *     IT_SPECIAL_GROUPS     =
  *     IT_TOOLBAR_EXCLUDING   =
  CHANGING
    it_outtab            = it_spfli
  *     IT_FIELDCATALOG       =
  *     IT_SORT               =
  *     IT_FILTER              =
  EXCEPTIONS
    OTHERS                = 1.

  IF sy-subrc <> 0.
    MESSAGE a012(bc412).
  ENDIF.

ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING  OUTPUT

```



## Lesson Summary

You should now be able to:

- Name the default functions of the SAP Grid Control
- Describe the screen layout of the SAP Grid Control
- Describe the data area layout of the SAP Grid Control
- Describe the print list layout of the SAP Grid Control
- Understand the technical view of the SAP Grid Control

# Lesson: Transporting Data and Additional Information

## Lesson Overview



### Lesson Objectives

After completing this lesson, you will be able to:

- Pass list data, the field catalog and additional information to the SAP Grid Control instance
- Work with layout variants in a SAP Grid Control

### Business Example

The users want to view flight information displayed with a SAP Grid Control. Every user should be enabled to save his/her own layout of the SAP Grid Control.

### Generating a Proxy Instance



```
DATA my_alv TYPE REF TO cl_gui_alv_grid.
```

```
...
```

```
CREATE OBJECT my_alv
  EXPORTING
    i_parent      =      Reference to a container instance
    *  i_appl_events =      All registered events as
                           application events?
                           • 'X': Yes
                           • '' (No (default value))
  EXCEPTIONS
    ...

```

Figure 125: Generating a Proxy Instance

The constructor for instances of the SAP Grid Control includes the following interface parameters:

- **I\_PARENT** Used to identify the instance of a Container Control as the Parent of the ALV Grid Control instance, linking them together.
- **I\_APPL\_EVENTS** Used to specify whether events of your ALV Grid Control instance are to be registered as system events (default) or application events. The proxy object registers the events with the Control Framework. For this reason, you need only register the event handler methods in the calling program.

## Passing List Data and Additional Information



```

CALL METHOD my_alv->set_table_for_first_display
* EXPORTING
*   i_structure_name      = Global structure for field catalog
*   is_variant            = } Managing display variants
*   i_save                = }
*   i_default             = }
*   is_layout              = Structure for displaying control
*   is_print               = Structure for displaying print list
*   it_special_groups     = Internal table with field group texts
*   it_toolbar_excluding  = Internal table with inactive functions
CHANGING
*     it_outtab            = Internal table with list data
*     it_fieldcatalog       = Internal table for field catalog
*     it_sort               = Internal table with initial sort criteria
*     it_filter              = Internal table with initial filter criteria
EXCEPTIONS
  ...

```

**Figure 126: Passing List Data and Additional Information**

You can use method SET\_TABLE\_FOR\_FIRST\_DISPLAY to pass the list data, the field catalog, and any additional information to the ALV Grid Control instance.

To pass the list data to be displayed, you use parameter IT\_OUTTAB.

If you pass the name of a global structure type to the parameter I\_STRUCTURE\_NAME, the proxy object then generates the field catalog automatically. The program then displays only the identically-named fields from the data table. This means that there must be a column in the data table with exactly the same name as each of the components in the structure type.

You use parameters IS\_VARIANT and I\_SAVE to control the configuration options of Layouts (display variants) available to users. You can make an existing Layout (display variant) the default using the I\_DEFAULT parameter.

The IS\_LAYOUT and IS\_PRINT parameters are used to pass settings for displaying the control and printing the print list.

The IT\_SPECIAL\_GROUPS parameter is used to pass the names of field groups defined in the field catalog.

The IT\_TOOLBAR\_EXCLUDING parameter is used to pass the name of the standard functions on the application toolbar that you want to hide.

If you want to change the automatically-generated field catalog or create a new one, you must pass an appropriate internal table to the IT\_FIELDCATALOG parameter.

The IT\_FILTER and IT\_SORT parameters allow you to pass initial filter and sort criteria for the data to be displayed.

The following slides deal with these parameters in more detail. For information about the other parameters, refer to the online documentation.

## Specifying the Administration Mode for Display Variants



Administration mode	Value for parameters IS_VARIANT and I_SAVE in the SET_TABLE_FOR_FIRST_DISPLAY method	
	IS_VARIANT	I_SAVE
Change (default setting)	space	space
Change and select existing variants	<st_name>	space
Change, save, and select existing variants: • User-specific saving only • Global saving only • Global and user saving	<st_name>	'U' 'X' 'A'

```
DATA <st_name> TYPE disvariant.  
...  
<st_name>-report = Program name (usually sy-cprog)  
<st_name>-handle = Logical name (only if program has  
several instances)
```

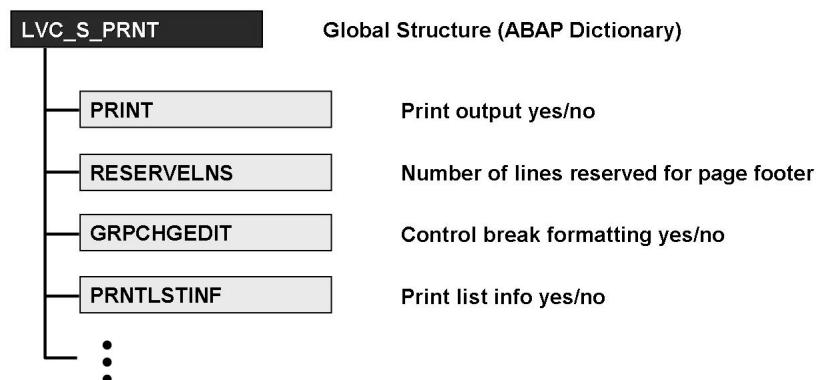
**Figure 127: Specifying the Administration Mode for Display Variants**

Use the IS\_VARIANT and I\_SAVE parameters to specify the variant administration options available to the user.

There are three possible modes:

- The user can change the current display variant: Assign the value **space** to both parameters (default setting).
- The user can change the current display variant and choose other existing variants: Pass a structure with the global type DISVARIANT to the parameter IS\_VARIANT. Assign the program name ((**sy-cprog**) to the field REPORT and **space** to the parameter I\_SAVE. If your calling program contains several control instances, you must be able to tell each variant mode apart by choosing a name and filling the HANDLE field with it.
- The user can change and save the current display variant, as well as choose other existing variants: Assign the same values to the parameter IS\_VARIANT as for the above mode. The value you assign to I\_SAVE specify the options available to the user for saving variants: “U” The user can save only user-specific display variants. “I008668X” The user can save only global display variants. “A” The user can save both user-specific and global display variants.

## Layout Information for the Print List



**Figure 128: Layout Information for the Print List**

You can make settings for the print list using the PRINT, RESERVELNS, GRPCHGEDIT, and PRNTLSTINF fields of a structure with the global type LVC\_S\_PRNT. Fill the IS\_PRINT parameter of the SET\_TABLE\_FOR\_FIRST\_DISPLAY method with a structure of this type.

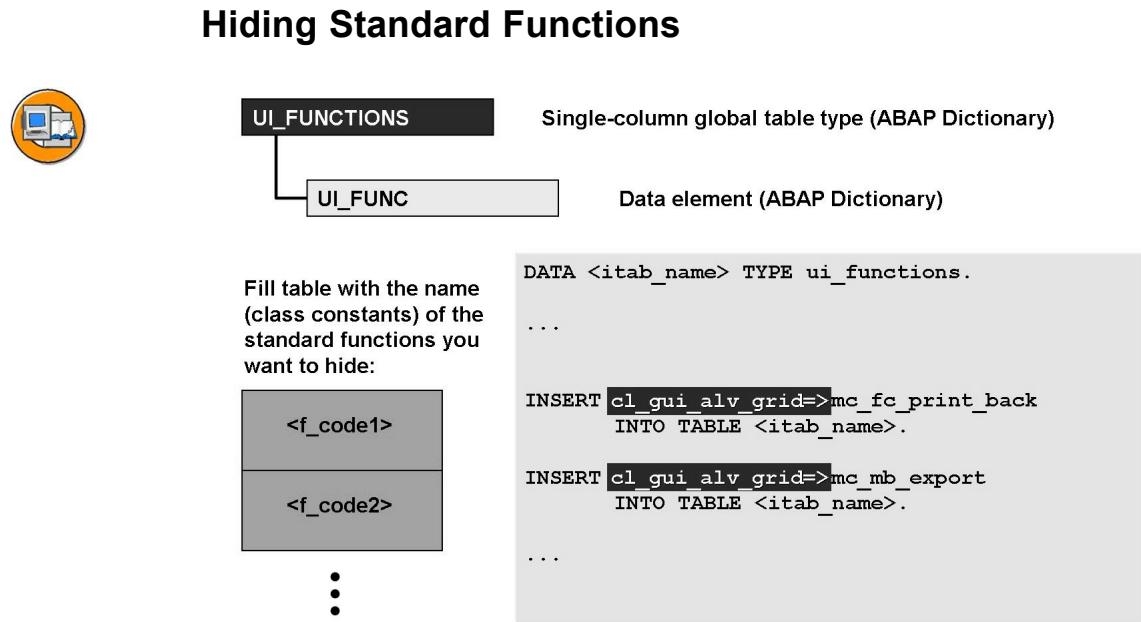
If you assign the value “X” to the PRINT field, you can use the SAP Grid Control to print **without first displaying the list** on screen. The passed data table is formatted at once as an ABAP print list and sent to the print spool. The program does not process the interactive screen display.

RESERVELNS allows you to specify the number of lines needed for the page footer.

GRPCHGEDIT enables user-definable group change editing for the print preview mode. If this field is set, the jump to the SAP List Viewer is configured accordingly. On the sort dialog box, the user can then determine how a sorting criterion value change is indicated graphically: as a page break or as an underline.

If you assign the value “X” to the PRNTLSTINF field, the interactive settings (such as filter criteria) and total number of lines are printed with the table.

The other fields of the structure are **not** used for the SAP Grid Control.



**Figure 129: Hiding Standard Functions**

You can limit the set of functions displayed in the application toolbar using an internal table with the global type **UI\_FUNCTIONS**. To do this, fill the **IT\_TOOLBAR\_EXCLUDING** parameter of the **SET\_TABLE\_FOR\_FIRST\_DISPLAY** method with a structure of this type.

Fill the internal table with the names of the functions you want to deactivate. Inactive functions on the application toolbar are hidden; inactive functions in menu paths are grayed out.

Use the constants of the class **CL\_GUI\_ALV\_GRID** to fill the field **UI\_FUNC**. The class constants for function codes begin with **MC\_FC\_**.

You can also deactivate an entire standard **menu**. The class constants for menus begin with **MC\_MB\_**.



#### Hint:

- To deactivate **all the functions** of the application toolbar, use the class constant **MC\_FC\_EXCL\_ALL**.
- To hide an **instance** of the application toolbar, use the **NO\_TOOLBAR** field of the **LVC\_S\_LAYO** structure.
- **Extending** the application toolbar by adding other functions requires an event-controlled implementation and will be dealt with later in this unit.

## Refreshing the Display



```
CALL METHOD my_alv->refresh_table_display
  EXPORTING
    *  is_stable      =  Retain scroll position of columns/rows
    *  i_soft_refresh =  Update data only, not sort criteria
  .
  DATA <st_name> TYPE lvc_s_stbl.
  ...
  <st_name>-row = Line-specific yes/no
  <st_name>-col = Column-specific yes/no
```

Causes the proxy instance to resend:

- Data table contents
- Additional information if necessary

→ Refreshes the presentation server display

**Figure 130: Refreshing the Display**

The method causes the proxy instance to **send again** the list data and (if necessary) the additional information to the presentation server instance.

The parameter I\_SOFT\_REFRESH is used only in exceptional cases. If you set this parameter, any totals created, any sort order defined and any filters set for the data displayed remain unchanged when the grid control is refreshed. This makes sense, for example, if you have not modified the data of the data table and want to refresh the grid control only with regard to layout or field catalog changes.

If the row or col field of the structure IS\_STABLE is set, the position of the scroll bar for the rows or columns remains stable when the data is refreshed.

→ **Note:** If you change the line **structure** of the data table, you must call the method SET\_TABLE\_FOR\_FIRST\_DISPLAY again, since the system must rebuild the **field catalog**.



## Lesson Summary

You should now be able to:

- Pass list data, the field catalog and additional information to the SAP Grid Control instance
- Work with layout variants in a SAP Grid Control

# Lesson: Adapting the Grid Layout

## Lesson Overview



### Lesson Objectives

After completing this lesson, you will be able to:

- Fill and pass the layout structure of the SAP Grid Control
- Explain the purpose of the field catalogue
- Add columns to or change columns of the SAP Grid Control
- Handle colors of rows and cells in the SAP Grid Control
- Suppress standard functionality of the toolbar

### Business Example

The users want to view flight information sorted by the current date displayed with an SAP Grid Control. Some cells will be shown in green, and some cells will be shown in blue. No downloading should be possible. To meet this request you need to define the layout structure.

### Adapting the Grid Layout Without a Field Catalog

You can often fulfill a need without using the field catalog:



- Include other columns → Use a different global structure type
- Hide or change order of columns → Use standard display variants
- Display only the first part of a column → Use the INCLUDE TYPE statement for the local line type of the data table
- Generate a data table dynamically (from Basis Release 6.10) → Pass the structure type as a field symbol

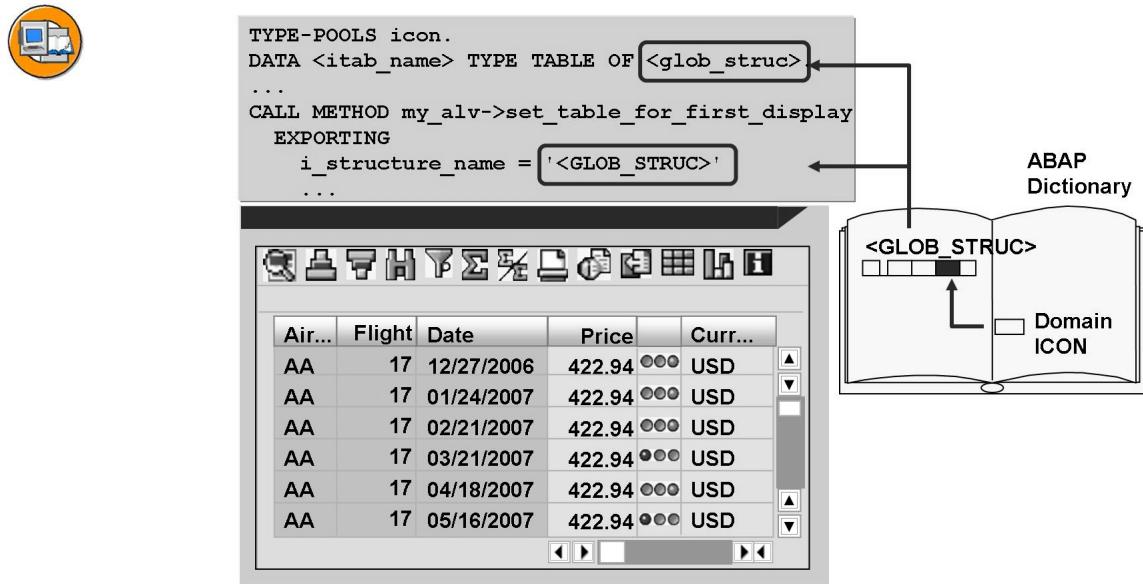
Basically, you need a field catalog for each list that is displayed using the ALV Grid Control. You have several options of generating a field catalog:

- Automatically through a Data Dictionary structure
- Manually in your ABAP program
- Semi-automatically by combining the above two procedures.

Before you define a field catalog, fill it, and pass it to the proxy instance, you may wish to consider fulfilling your requirements differently. In many cases, the above alternatives provide an easier, more flexible solution.

Many of these alternative solutions also offer other advantages – they are reusable, they can be maintained centrally, and so on.

## Example: Outputting Icons



**Figure 131: Example: Outputting Icons**

To manage icons in a context-specific way in the data table, there must be an appropriate column available. Thus it often makes sense to define this column as a component in the global structure type. This type can provide a basis for the table type and for the automatic generation of the field catalog.

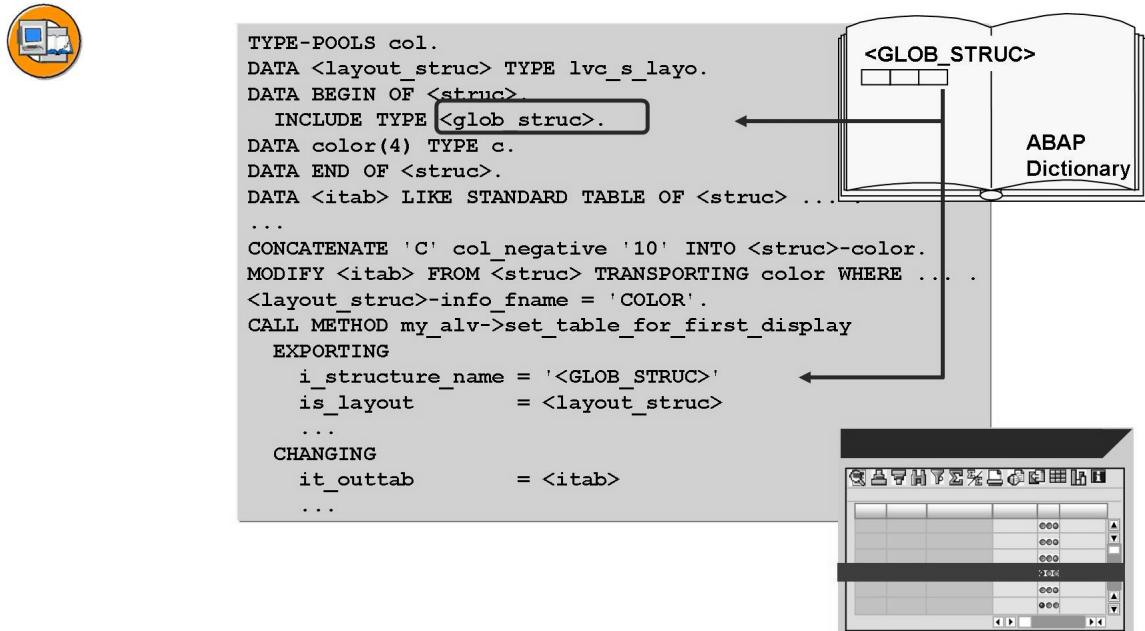
When creating the global structure type in the ABAP Dictionary, note that you can reference **existing** structure fields in different ways.

Insert a component with the data type ICON (domain) where you would like it to be in the table.

In the program, define an internal table with this type and fill the icon column with the icon names from the type group ICON.

Have the proxy instance generate the field catalog based on the global structure type.

## Example: Changing the Color of a Record



**Figure 132: Example: Changing the Color of a Record**

If you want to generate the field catalog automatically using a global structure type, you need only add a new field for the color value in the data table. This field should be the **last** component of the line type of the data table. If you then simply assign the original global structure type as the line type for the SAP Grid Control, the color values will **not** be displayed as a column on the screen. Instead they will be used to change the color of the rows in the table, which is usually what is required.

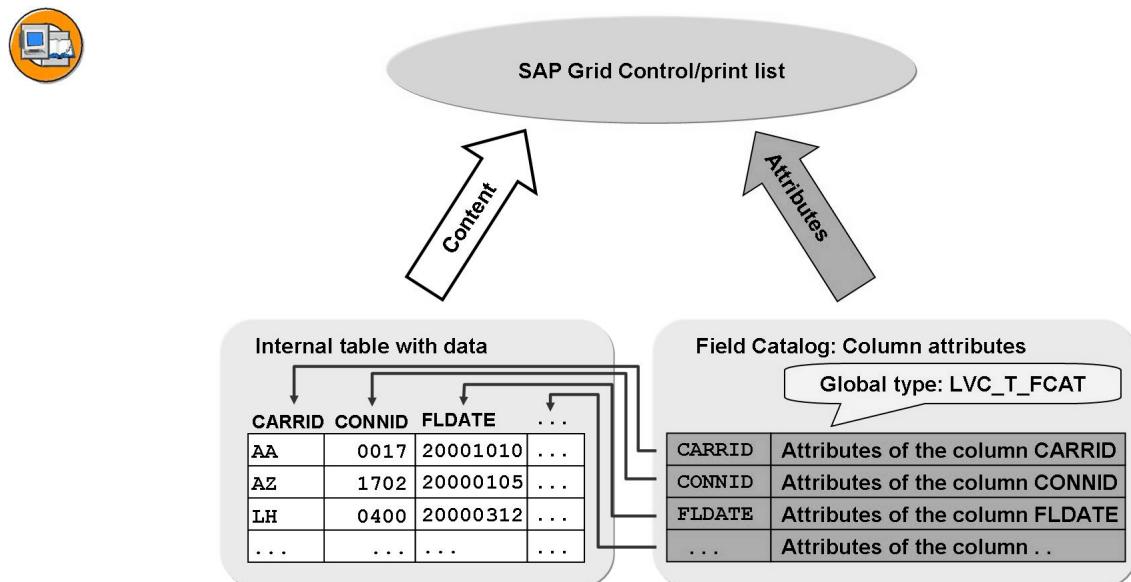
The **INCLUDE STRUCTURE** statement provides a simple way of defining the necessary line type for the data table locally in the program.

Fill the color column of your data table. As a rule, comply with the following convention: “C<color\_constant>10”, where color\_constant is one of the eight color constants of the type group COL.

In the INFO\_FNAME field of a structure with the global type LVC\_S\_LAYO, enter the component name of your color column and pass the structure to the IS\_LAYOUT parameter of the SET\_TABLE\_FOR\_FIRST\_DISPLAY method.

For more details on color formatting applied to rows or columns, refer to the online documentation.

## Recapitulation: Function of the Field Catalog



**Figure 133: Recapitulation: Function of the Field Catalog**

**Remember** – The field catalog is an internal table containing the column formats for the data in the SAP Grid Control. If a global structure type contains only those components that have an identically-named column in the database table, you can make the proxy instance generate the field catalog itself. You need only specify the name of the global structure type. Generally, this is also the line type of the data table.

Thus you need only generate the field catalog in the calling program if you do not use the global type definitions from the ABAP Dictionary for displaying the table in the SAP Grid Control, but conversely do not want to search for or create new global types in the ABAP Dictionary.

Typical situations where the above might apply could be:

- You want to change the column width. (If the field catalog is generated automatically, the width is taken from the data element)
- You want to change the column header. (Again, if the field catalog is generated automatically, the header is taken from the data element)
- You want to change the fixed columns. (If the field catalog is generated automatically, the width is taken from the Key field attribute)

## Grid Layout Options



Which parameters of the SET\_TABLE\_FOR\_FRIST\_DISPLAY method must be filled, and in which situations?

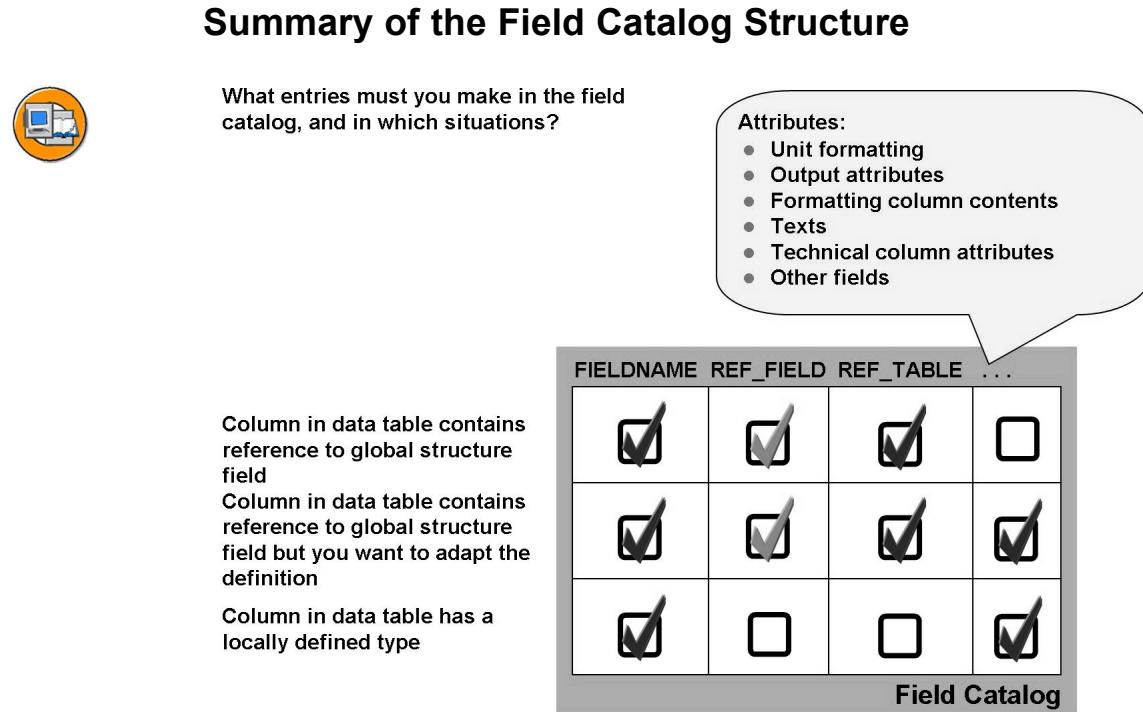
Structure types in the ABAP Dictionary		
Data table in the calling program		
Display in the SAP Grid Control		
Parameter I_STRUCTURE_NAME	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Parameter IT_FIELDCATALOG	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 134: Grid Layout Options

Field catalogs are created in one of the following three ways depending on the line type used by the data table:

- **Case 1:** All the fields of the global structure type have an identically-named column in the data table. You want to display precisely these columns. In this case, you can have the proxy instance generate the field catalog. You need only pass the name of the global structure type. Data table columns are not displayed on the screen unless they are defined in the global structure type.
- **Case 2:** All fields of the global structure type have an identically-named column in the data table, but you want to adapt the attributes derived from the ABAP Dictionary or you want to add columns or both. In this case, you need to add lines in the field catalog matching the columns you want to add or change.
- **Case 3:** Either there are no Dictionary references in the line type of the data table or there are references only for some of the global structure fields. In this case, the calling program must generate the field catalog from scratch.

The following slides deal with **cases 2 and 3** in more detail.



**Figure 135: Summary of the Field Catalog Structure**

The information presented here applies only to the **second** and **third** situations outlined on the previous slide.

You create the field catalog in the calling program as follows:

- Define an internal table for the field catalog based on the global table type LVC\_T\_FCAT and a structure with a compatible line type as a work area.
- Fill the fields of the work area for each relevant column in your data table and add it to the internal table for the field catalog.

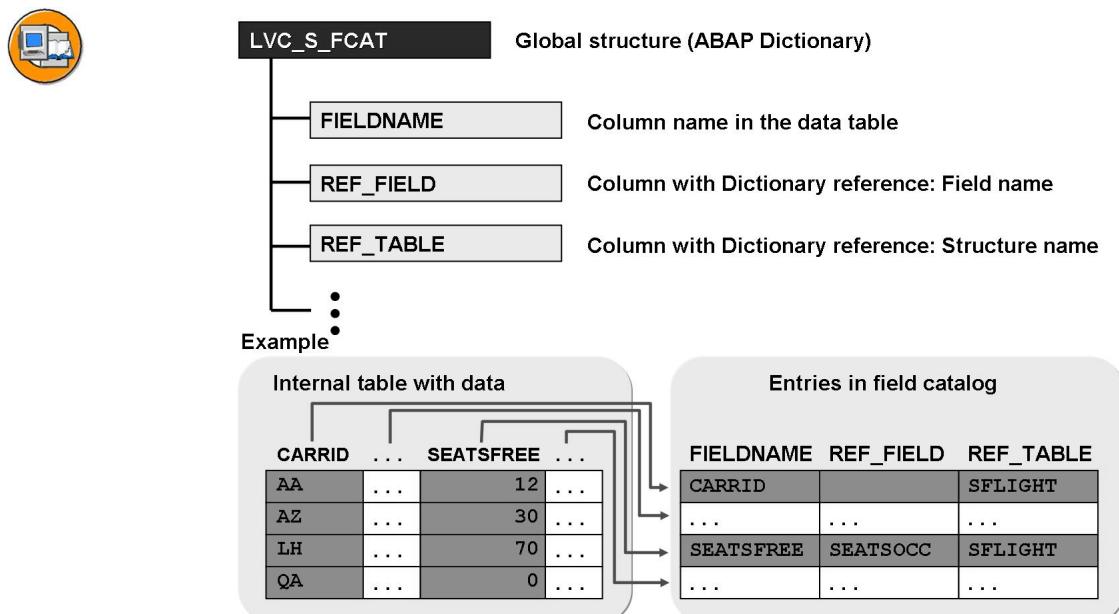
Fill the FIELDNAME field with a column name from the data table. Use this field to assign a column definition in the field catalog to a data table column.

As far as using them to create a field catalog, you can split the other fields in the field catalog into two groups:

- You can create a reference to global types (that is, structure fields from the ABAP Dictionary) using the REF\_FIELD and REF\_TABLE fields. These fields belong to the first group.
- All other fields in the field catalog contain values for column attributes. These fields belong to the second group.

If you fill the REF\_TABLE and where necessary REF\_FIELD fields, the program takes all type definitions from the specified structure fields. You can overwrite **single** fields in the second group by assigning values to them. If you do not fill either REF\_FIELD or REF\_TABLE, you **must, if necessary**. fill **all** the fields in the second group.

## Column Labels



**Figure 136: Column Labels**

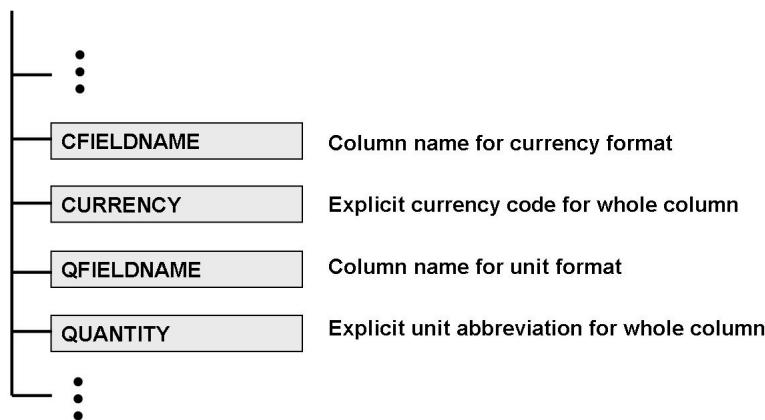
The column FIELDNAME in the field catalog contains the names of the data table columns. Each column of the data table whose formatting you want to change must have a line in the field catalog.

If you want to reference an **identically-named** field of a global structure type, you need only fill the REF\_TABLE field with the name of this structure. You only need to fill the REF\_FIELD field with the name of the field in the structure if the column name of the data table differs from the name of this field.

In the above example, we want to take as many column attributes from the ABAP Dictionary as possible and only overwrite single entries: The column displaying the number of free seats is linked to the field associated with the data element SEATSOCC in the transparent table SFLIGHT. This column header for this data element is Occupied seats. However, we want to display the column header Free seats, so we need to fill the COLTEXT field in the field catalog with this text.



## Appropriate Formatting for Units



**Figure 137: Appropriate Formatting for Units**

Currency formatting:

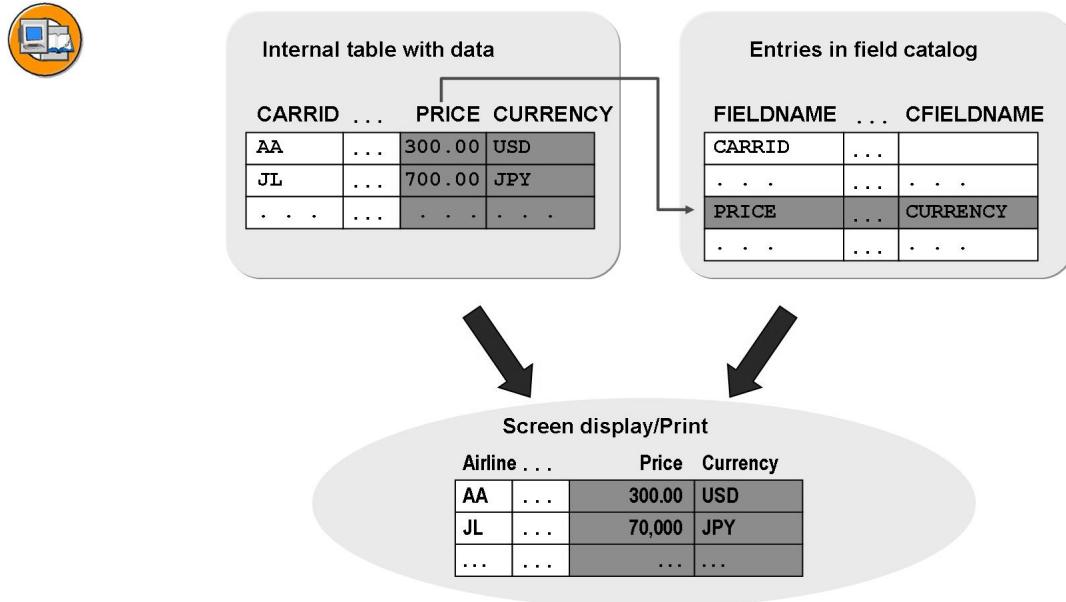
- If the content of a column is to be formatted based on the formatting options of a currency unit contained in another column of the data table, then you must specify the name of the column with the currency units in the field CFIELDNAME.
- If all the values in a column are to be formatted according to a fixed currency, you must enter the currency code for that currency in the CURRENCY column.
- If a value is assigned to the CURRENCY field, entries in the CFIELDNAME field are ignored.

Formatting for units of measure:

- If the content of a column is to be formatted based on the formatting options of a unit of measure contained in another column of the data table, then you must specify the name of the column with the units of measure in the field QFIELDNAME.
- If all the values in a column are to be formatted according to a fixed unit, you must enter the abbreviation for that unit in the QUANTITY column.
- If a value is assigned to the QUANTITY field, entries in the QFIELDNAME field are ignored.

For further information, refer to the online documentation.

## Example: Appropriate Formatting for Currency Columns



**Figure 138: Example: Appropriate Formatting for Currency Columns**

In the above example, the data table contains the two columns PRICE (amount to be paid for the flight) and CURRENCY (currency unit for the amount in PRICE).

The “CURRENCY” entry in the CFIELDNAME field of the field catalog for the PRICE column specifies that the content of this column is to be formatted according to the currency entered in the CURRENCY column. (For example, “USD” ® two decimal places, “JPY” ® no decimal places)



## Output Options for Columns

	<b>CHECKBOX</b>	Display checkbox (not enabled for input)
	<b>COL_POS</b>	Column position
	<b>DO_SUM</b>	Total function for column: Yes/no
	<b>HOTSPOT</b>	Make column hotspot-sensitive: Yes/no
	<b>NO_SUM</b>	Suppress totals function for column: Yes/no
	<b>EMPHASIZE</b>	Highlight column content (color, light, inverse display)
	<b>KEY</b>	Key column: Yes/no
	<b>LOWERCASE</b>	Uppercase and lowercase: Yes/no
	<b>NO_OUT</b>	Do not display column: Yes/no
	<b>OUTPUTLEN</b>	Column width
	<b>TECH</b>	Technical column: Yes/no

**Figure 139: Output Options for Columns**

→ **Note:** COL\_POS: The default setting ("0") means that the column is displayed, left-aligned, after the key columns (those for which the KEY field is filled with "X").

### EMPHASIZE:

The possible values for this four characters field are: SPACE, "X" or "Cxyz" (x: "1"- "9"; y,z: "0"=off "1"=on)

If the field is set to "X", the ALV uses a pre-defined color for highlighting the column.

If the character field begins with "C" (color code), the remaining numbers have the following meaning:

- x: color number
- y: intensified display on/off
- y: inverse display on/off
- NO\_OUT: Hides the column (both for screen display and printing). However, it appears as a possible choice in dialog boxes for standard functions (such as sort or filter).
- TECH: The column is not displayed, either in the list or in the dialog boxes for selecting fields.

For further information, refer to the online documentation.

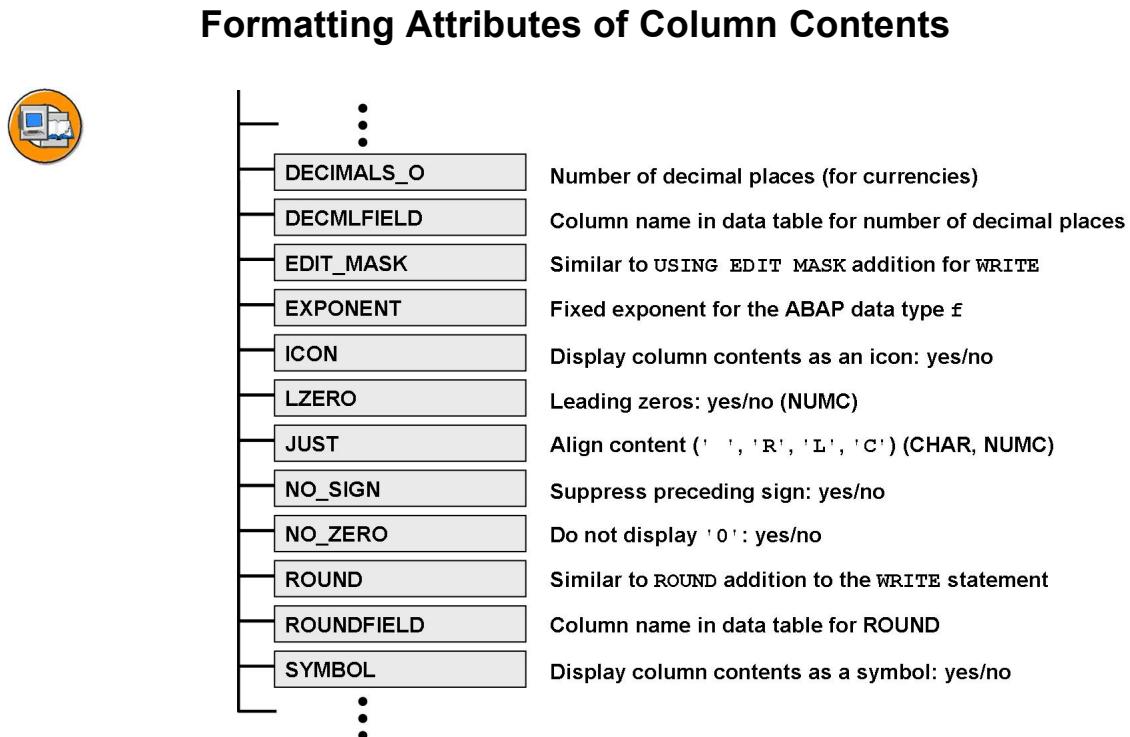


Figure 140: Formatting Attributes of Column Contents

You can use the DECMFIELD and DECIMALS\_O fields to specify the number of decimal places (similar to the formatting options used for currency fields).

These entries affect numeric content.

You can apply this formatting to single cells or whole columns.

For further information, refer to the online documentation.



## Texts

COLDDICTXT	Text to use for column header ('L', 'M', 'S')
COLTEXT	Freely-defined column header
REPTEXT	Header text from ABAP Dictionary
SCRTEXT_L	Long text
SCRTEXT_M	Medium text
SCRTEXT_S	Short text
SELDDICTXT	Used as the text for column selection ('L', 'M', 'S', 'R')
SELTEXT	Freely defined text for selection screen
TIPDDICTXT	Used as the tool tip ('L', 'M', 'S', 'R')
TOOLTIP	Freely definable text as the tool tip

**Figure 141: Texts**

The three fields, SCRTXT\_L, SCRTXT\_M, and SCRTXT\_S can contain three texts of different lengths, similar to the texts attached to data elements in the ABAP Dictionary. You can then use these texts as labels for columns, selection screens, and detail lists.

The value you assign to the COLDDICTXT field specifies which text is used as a column header ("S" = short, "M" = middle, "L" = long). Similarly, SELDDICTXT and TIPDDICTXT specify which text is used on selection screens and in detail lists (for standard functions).

The COLTEXT, SELTEXT, and TOOLTIP fields allow you to create freely-defined texts for column headers, selection screens, and detail lists. Although this values would be ignored if the fields COLDDICTXT, SELDDICTXT, and TIPDDICTXT are filled.

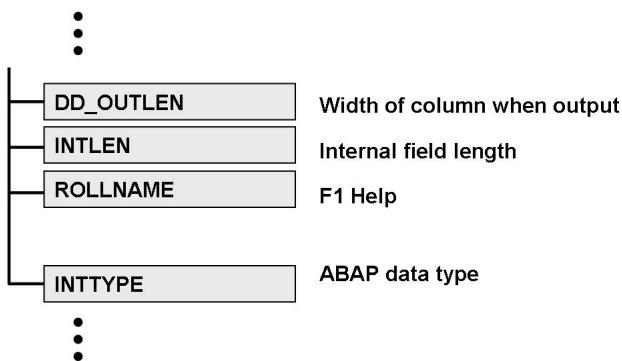
The REPTEXT is relevant only to fields with reference to the Data Dictionary. For such fields, the ALV Grid Control copies the field label for the header of the corresponding data element into this field.

For further information, refer to the online documentation.

## Technical Attributes of Columns



Used only with data table columns that do not make a reference to the ABAP Dictionary:



**Figure 142: Technical Attributes of Columns**

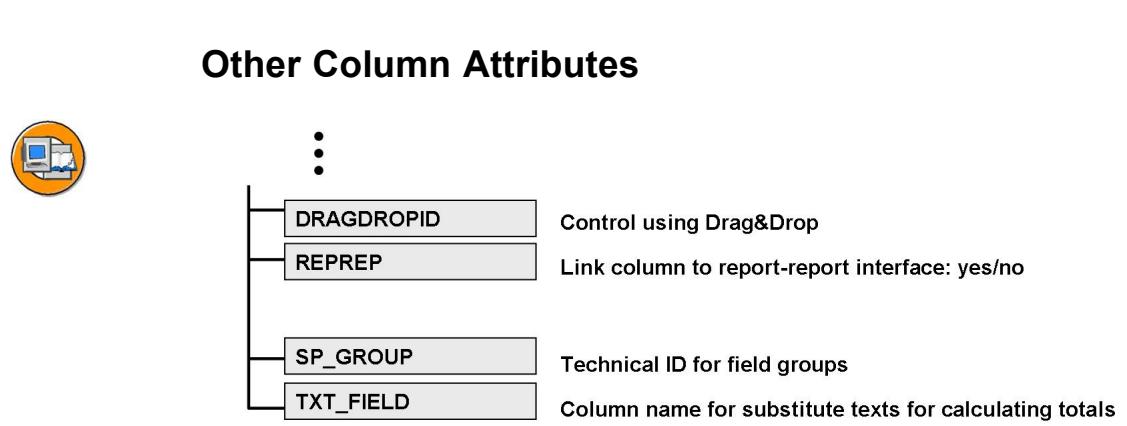
If you do not fill either the REF\_FIELD or the REF\_TABLE field, there is no ABAP Dictionary reference for your fields. In such cases, the fields shown above are used to format these columns technically for output.

If you want to provide F1 help for an output field without Data Dictionary reference or if you want to define a different F1 help than that stored in the DDIC for a field with DDIC reference, you can use the ROLLNAME field. If F1 help is called for this field, the documentation for the data element assigned is displayed. If ROLLNAME is initial for fields with Data Dictionary reference, the documentation for the data element of the referenced field of the Data Dictionary is displayed.

You use the INTLEN field to specify the field output length for internal display. This is only relevant to fields without reference to the Data Dictionary, for which you want to modify the output using a conversion exit.

INTTYPE is only required for field without reference to the Data Dictionary.

For further information, refer to the online documentation.



**Figure 143: Other Column Attributes**

Note on the TXT\_FIELD: You can use this field to specify that **numeric** field contents in the list are **replaced** by a text.

Example: Column A contains plant numbers, column B their associated long text. You want to display the long texts, not the plant numbers in the list. In the TXT\_FIELD field for column A, enter the name of column B. (That is: <struc\_name>-txt\_field = "B".)

For further information, refer to the online documentation.

## Exercise 11: Field Catalog

### Exercise Objectives

After completing this exercise, you will be able to:

- Change the color of rows in the SAP Grid Control
- Create and use a field catalog for the SAP Grid Control

### Business Example

Extend the program from the last exercise. In the SAP Grid Control, highlight all flight connections that last one day or longer in orange. In addition, use the different texts for the flight duration column header than those taken from the data element

**Program:**ZBC412\_##\_GRD\_EX2

**Template:**SAPBC412\_GRDS\_EXERCISE\_1

**Model solution:**SAPBC412\_GRDS\_EXERCISE\_2

where ## is the group number.

### Task 1:

Copy the Template.

1. Copy your solution from the previous exercise (*ZBC412\_##\_GRD\_EX1*) or the appropriate model solution (*SAPBC412\_GRDS\_EXERCISE\_1*) to the name **ZBC412\_##\_GRD\_EX2**. Get to know how your copy of the program works

*Continued on next page*

## Task 2:

Make sure that flights that last longer than one day are colored orange



**Hint:** To fulfill this requirement, make use of the possibility you have to retain additional information in the data table that the program will not display in the SAP Grid Control (provided your data table is suitably typed). (This means that you can still have the proxy instance generate the field catalog for this task).

Proceed as follows:

1. Define a local line type (suggested name *st\_spfli\_col*), which contains all the fields of the structure *SPFLI* and an additional field (suggested name: *color*) for the color values. Use the *INCLUDE STRUCTURE* statement to copy the fields of the *SPFLI* table. Assign the type **c**, length **4** to the color values field and make it the **last** component in your line type
2. Assign the new line type to the internal table for the flight data, *it\_spfli*.
3. Fill the additional data table column with the color values. You only need to modify those rows of the table that fulfill the condition “Arrival in number of days later” >0. Color these lines orange. Include the type group *col* and use the color constant *col\_group*.
4. Define a work area for layout settings of the SAP Grid Control (suggested name: *wa\_layout*). In the module *init\_control\_processing*, fill the *info\_fname* field with the name of the color column in the data table. Pass the filled work area to the *set\_table\_for\_first\_display* method. Remember in addition to have the proxy instance generate the field catalog. How must you fill the *i\_structure\_name* parameter so that the content of the color column in the data table is not displayed?

## Task 3:

Make sure that the three headers of different lengths for the flight duration column header are not taken from the data element. Replace them with the following texts:

“FT”

“Flight time”

“Flight time in Hours:Minutes”

Proceed as follows:

1. Define an internal table for a field catalog (suggested name: *it\_fcat*) and a suitable work area (suggested name: *wa\_fcat*).

*Continued on next page*

2. Fill the field catalog with the appropriate entries for the *fltime* column of the data table. Use text symbols so that you can maintain the texts in a multi-lingual way.
3. Fill the parameters *i\_structure\_name*, *it\_outtab*, and *it\_fieldcatalog* of the method *set\_table\_for\_first\_display* with appropriate values

## Solution 11: Field Catalog

### Task 1:

Copy the Template.

1. Copy your solution from the previous exercise (*ZBC412\_##\_GRD\_EX1*) or the appropriate model solution (*SAPBC412\_GRDS\_EXERCISE\_1*) to the name **ZBC412\_##\_GRD\_EX2**. Get to know how your copy of the program works
  - a) -

### Task 2:

Make sure that flights that last longer than one day are colored orange



**Hint:** To fulfill this requirement, make use of the possibility you have to retain additional information in the data table that the program will not display in the SAP Grid Control (provided your data table is suitably typed). (This means that you can still have the proxy instance generate the field catalog for this task).

Proceed as follows:

1. Define a local line type (suggested name *st\_spfli\_col*), which contains all the fields of the structure *SPFLI* and an additional field (suggested name: *color*) for the color values. Use the *INCLUDE STRUCTURE* statement to copy the fields of the *SPFLI* table. Assign the type **c**, length **4** to the color values field and make it the **last** component in your line type
  - a)

```
TYPES BEGIN OF st_spfli_col.
  INCLUDE STRUCTURE spfli.
  TYPES: color(4) TYPE c,
  END OF st_spfli_col.
```

2. Assign the new line type to the internal table for the flight data, *it\_spfli*.
  - a)

```
DATA:
...
  it_spfli      TYPE TABLE OF st_spfli_col.
...
```

*Continued on next page*

3. Fill the additional data table column with the color values. You only need to modify those rows of the table that fulfill the condition “Arrival in number of days later” >0. Color these lines orange. Include the type group `col` and use the color constant `col_group`.

a)

```
TYPE-POOLS col.  
...  
CONCATENATE 'C' col_group '10' INTO wa_spfli-color.  
MODIFY it_spfli FROM wa_spfli TRANSPORTING color WHERE period > 0.
```

4. Define a work area for layout settings of the SAP Grid Control (suggested name: `wa_layout`). In the module `init_control_processing`, fill the `info_fname` field with the name of the color column in the data table. Pass the filled work area to the `set_table_for_first_display` method. Remember in addition to have the proxy instance generate the field catalog. How must you fill the `i_structure_name` parameter so that the content of the color column in the data table is not displayed?

a)

```
DATA:  
...  
    wa_layout TYPE lvc_s_layo.  
...  
    * set color column for SAP Grid Control:  
    wa_layout-info_fname = 'COLOR'.  
...
```

*Continued on next page*

### Task 3:

Make sure that the three headers of different lengths for the flight duration column header are not taken from the data element. Replace them with the following texts:

“FT”

“Flight time”

“Flight time in Hours:Minutes”

Proceed as follows:

1. Define an internal table for a field catalog (suggested name: *it\_fcat*) and a suitable work area (suggested name: *wa\_fcat*).

a)

```
DATA:  
...  
    it_fcat TYPE lvc_t_fcat,  
    wa_fcat LIKE LINE OF it_fcat.
```

2. Fill the field catalog with the appropriate entries for the *ftime* column of the data table. Use text symbols so that you can maintain the texts in a multi-lingual way.

a)

```
*   adjust header for column 'FLTIME':  
CLEAR:  
    it_fcat,  
    wa_fcat.  
    wa_fcat-fieldname = 'FLTIME'.  
    wa_fcat-scrtext_s = text-fts.  
    wa_fcat-scrtext_m = text-ftm.  
    wa_fcat-scrtext_l = text-ftl.  
    INSERT wa_fcat INTO TABLE it_fcat.
```

3. Fill the parameters *i\_structure\_name*, *it\_outtab*, and *it\_fieldcatalog* of the method *set\_table\_for\_first\_display* with appropriate values

a)

```
CALL METHOD ref_alv->set_table_for_first_display  
    EXPORTING  
        *           I_BUFFER_ACTIVE      =
```

*Continued on next page*

```

        i_structure_name      = 'SPFLI'
*      IS_VARIANT          =
*      I_SAVE               =
*      I_DEFAULT            = 'X'
        is_layout             = wa_layout
*      IS_PRINT              =
*      IT_SPECIAL_GROUPS    =
*      IT_TOOLBAR_EXCLUDING =
*      IT_HYPERLINK         =
*      IT_ALV_GRAPHICS      =
CHANGING
        it_outtab             = it_spfli
        it_fieldcatalog       = it_fcat
*      IT_SORT               =
*      IT_FILTER              =
EXCEPTIONS
        OTHERS                = 4 .

```

## Result

The complete source code for the model solution is shown below.

### ABAP program - Data declarations

```

REPORT  sapbc412_grds_exercise_2 MESSAGE-ID bc412.

TYPE-POOLS col.

TYPES BEGIN OF st_spfli_col.
  INCLUDE STRUCTURE spfli.
TYPES: color(4) TYPE c,
END OF st_spfli_col.

DATA:
  ok_code      TYPE sy-ucomm,
  copy_ok_code LIKE ok_code,

  ref_container TYPE REF TO cl_gui_docking_container,
  ref_alv       TYPE REF TO cl_gui_alv_grid,

  it_spfli     TYPE TABLE OF st_spfli_col,
  it_sflight   TYPE TABLE OF sflight,

  wa_spfli     LIKE LINE OF it_spfli,

```

*Continued on next page*

```

wa_sflight      LIKE LINE OF it_sflight,
wa_layout TYPE lvc_s_layo,
it_fcat TYPE lvc_t_fcat,
wa_fcat LIKE LINE OF it_fcat.
```

## Event blocks

```

START-OF-SELECTION.

SELECT * FROM spfli INTO TABLE it_spfli WHERE carrid IN so_carr
      ORDER BY CARRID connid.
IF sy-subrc NE 0.
  MESSAGE a061(bc412).
ENDIF.

CONCATENATE 'C' col_group '10' INTO wa_spfli-color.
MODIFY it_spfli FROM wa_spfli TRANSPORTING color WHERE period > 0.

...
```

## Modules

```

*&-----*
*&     Module INIT_CONTROL_PROCESSING OUTPUT
*&-----*
*     control related processing
*-----*
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.

  ...

*   set color column for SAP Grid Control:
  wa_layout-info_fname = 'COLOR'.
*   adjust header for column 'FLTIME':
  CLEAR:
    it_fcat,
    wa_fcat.
  wa_fcat-fieldname = 'FLTIME'.
  wa_fcat-scrtext_s = text-fts.
  wa_fcat-scrtext_m = text-ftm.
```

*Continued on next page*

```
wa_fcat-scrtext_1 = text-ft1.  
INSERT wa_fcat INTO TABLE it_fcat.  
  
CALL METHOD ref_alv->set_table_for_first_display  
      EXPORTING  
*       I_BUFFER_ACTIVE          =  
*       i_structure_name         = 'SPFLI'  
*       IS_VARIANT               =  
*       I_SAVE                   =  
*       I_DEFAULT                = 'X'  
*       is_layout                = wa_layout  
*       IS_PRINT                 =  
*       IT_SPECIAL_GROUPS        =  
*       IT_TOOLBAR_EXCLUDING     =  
*       IT_HYPERLINK             =  
*       IT_ALV_GRAPHICS          =  
      CHANGING  
*       it_outtab                = it_spfli  
*       it_fieldcatalog          = it_fcat  
*       IT_SORT                  =  
*       IT_FILTER                =  
      EXCEPTIONS  
      OTHERS                   = 4  
  
....
```



## Lesson Summary

You should now be able to:

- Fill and pass the layout structure of the SAP Grid Control
- Explain the purpose of the field catalogue
- Add columns to or change columns of the SAP Grid Control
- Handle colors of rows and cells in the SAP Grid Control
- Suppress standard functionality of the toolbar

# Lesson: Events

## Lesson Overview

This lesson presents the events raised by the SAP Grid Control and how to react to them.



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain how events are triggered and how they can be handled
- Identify existing events
- Create and register an event handler method
- Add new functionality to the standard toolbar
- Enhance the print list by means of event handling

## Business Example

The users want to view flight information displayed with an SAP Grid Control. You are asked to implement this: When a user double-clicks a row, another SAP Grid Control will be displayed, giving information about the bookings of the flight of the chosen row.

## Mouse Operations

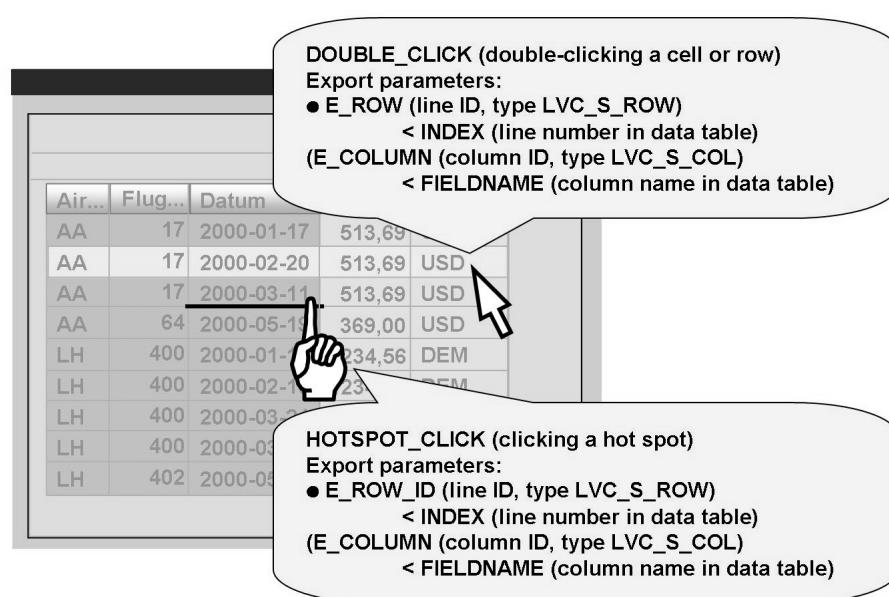


Figure 144: Mouse Operations

If the user double-clicks a data area, the **DOUBLE\_CLICK** event is triggered.

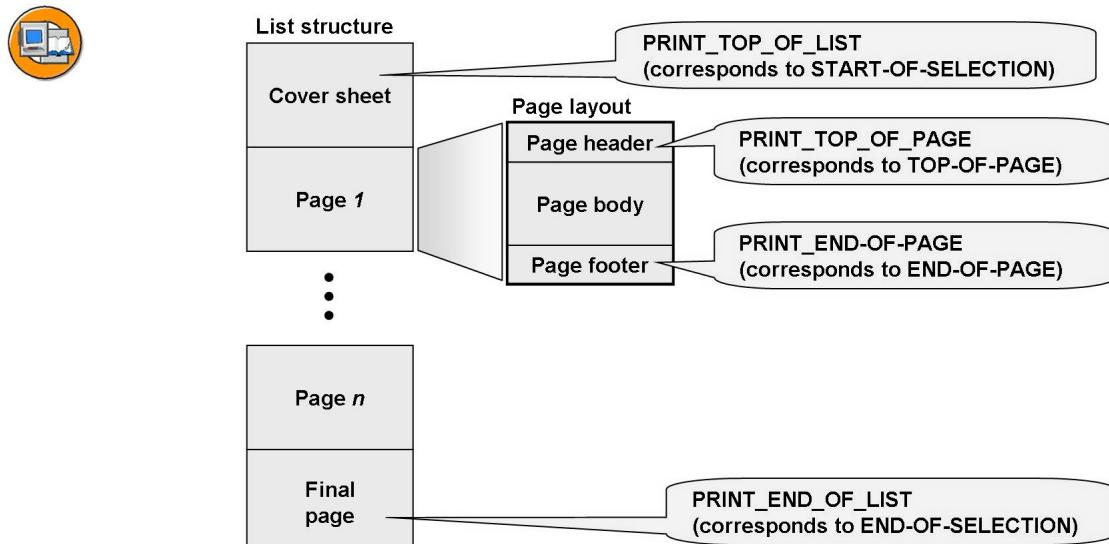
The export parameters contain information about the data table. Note that both parameters are structures.

- **E\_ROW:** The **INDEX** field contains the line number in the internal table that corresponds to the double-clicked row in the data area.
- **E\_COLUMN:** The **FIELDNAME** field contains the name of the column in the internal table that corresponds to the double-clicked cell in the data area.

If a column is characterized as a hot spot, the **HOTSPOT\_CLICK** event is triggered when the user clicks this column in the data area. The export parameters are filled in a similar way to those of the **DOUBLE\_CLICK** event.

For further information about other mouse operation events, refer to the online documentation.

## Print List Events



**Figure 145: Print List Events**

In the calling program you can extend the print list automatically generated by the proxy instance. To do this, use the above events, which allow you to perform the same logical work steps as in “classical” ABAP print lists.

Thus, in the event handler methods to be implemented, you can use the usual statements for ABAP list output (**WRITE**, **SKIP**, **ULINE**, and so on) as well as their many additions.

You can use the **SUBTOTAL\_TEXT** event to output text for subtotals in the grid control if the subtotal criterion (column used by the user for calculating the subtotal) is hidden. In the default setting, the ALV Grid Control outputs the column header of the subtotal criterion and the value to which the subtotal calculated refers. For further details, refer to the online documentation.

## Event When Creating the Application Toolbar

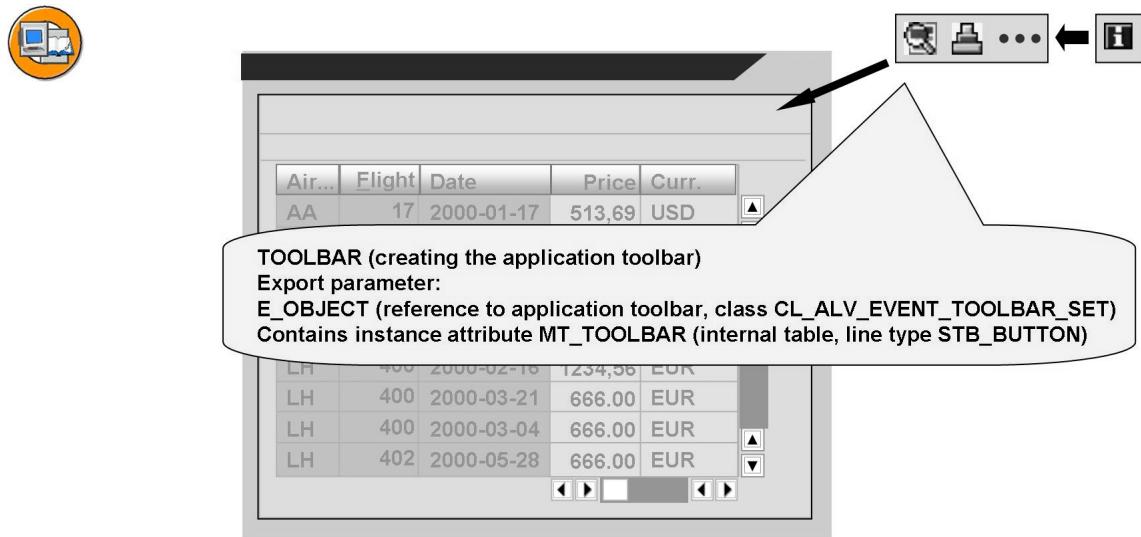


Figure 146: Event When Creating the Application Toolbar

Before the standard application toolbar is inserted in the SAP Grid Control, **the proxy instance** triggers the **TOOLBAR** event – that is, while executing the **SET\_TABLE\_FOR\_FIRST\_DISPLAY** and **REFRESH\_TABLE\_DISPLAY** methods.

During the lifetime of the proxy instance, you can trigger this event **again** by calling the **SET\_TOOLBAR\_INTERACTIVE** method.

To add elements to the application toolbar, you must implement a handler method for the **TOOLBAR** event of the **CL\_GUI\_ALV\_GRID** class.

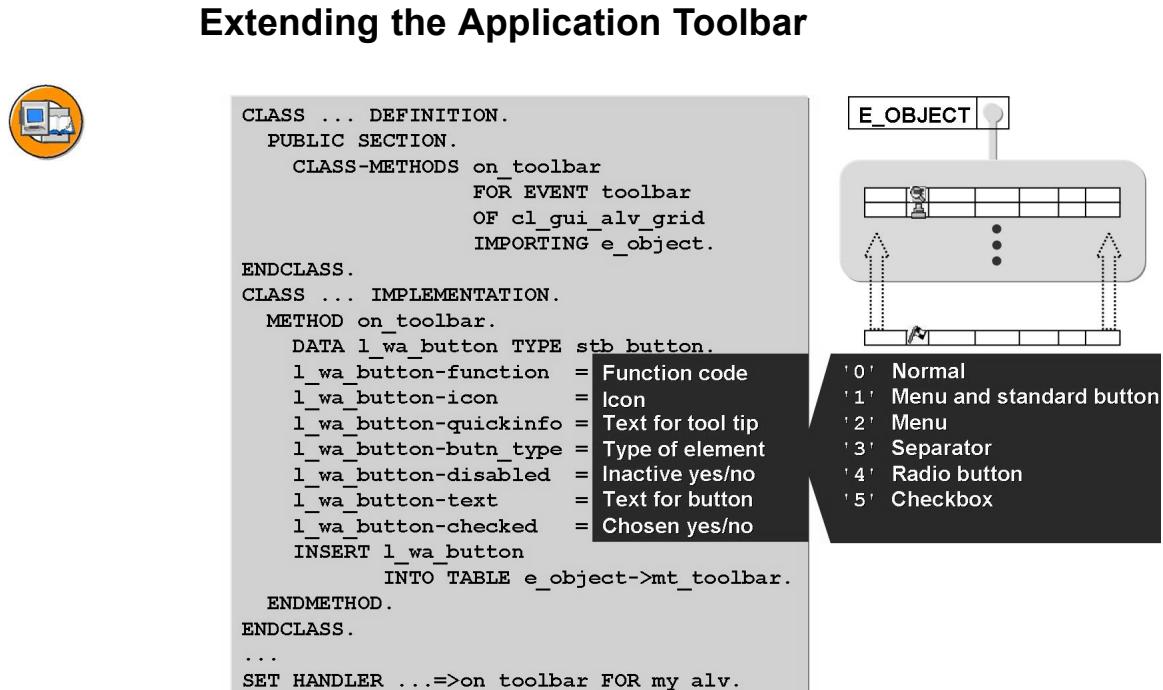


Figure 147: Extending the Application Toolbar

The **TOOLBAR** event has a pointer to an instance of the `CL_ALV_EVENT_TOOLBAR_SET` class as an export parameter. This application toolbar instance in turn contains the public attribute **MT\_TOOLBAR**. `MT_TOOLBAR` is an internal table with the line type **STB\_BUTTON**. Therefore, to add additional buttons or other elements, you need to fill this internal table in the handler method.

If the user chooses an element of type “1” or “2”, the proxy instance triggers the **MENU\_BUTTON** event. Therefore, to add a menu to the application toolbar, you must implement a handler method in which you create a menu **exactly** as you would for a **context menu**. You can find out which menu has been chosen by querying the function code.

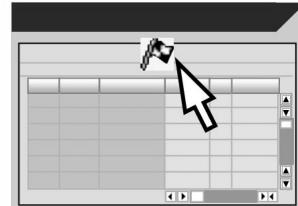
## Storing a Function



```

CLASS ... DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS on_user_command
      FOR EVENT user_command
      OF cl_gui_alv_grid
      IMPORTING e_ucomm.
  ENDCLASS.
CLASS ... IMPLEMENTATION.
  METHOD on_user_command.
    CASE e_ucomm.
      WHEN '....'.
      ...
      WHEN '....'.
      ...
    ENDCASE.
  ENDMETHOD.
ENDCLASS.
...
SET HANDLER ...=>on_user_command FOR my_alv.

```



**Figure 148: Storing a Function**

If the user chooses a function element that has been added to the standard application toolbar, the proxy instance triggers the **USER\_COMMAND** event. This contains the function code as the **export parameter E\_UCOMM**.

To store this additional function, you must implement a handler method for this event. You can find out which function element has been chosen by querying the function code.

## Demo Programs Delivered for the SAP Grid Control



Package	Programs	Demonstrates
SLIS	BCALV_GRID_01 BCALV_GRID_02 BCALV_GRID_03 BCALV_GRID_04 BCALV_GRID_05 BCALV_GRID_06 BCALV_GRID_07 BCALV_GRID_08 BCALV_GRID_09 BCALV_GRID_10	Print list events Basic and detail lists with an instance Detail list in a modal dialog box Threshold value handling (exceptions/traffic light icons) Adding your own functions to the application toolbar Context menus Adding your own menus to the application toolbar Menu with standard pushbutton Options for managing display variants Initial use of display variants

**Figure 149: Demo Programs Delivered for the SAP Grid Control**

As well as these demonstration and test programs, you will also find other Repository objects for the SAP Grid Control in the **SLIS** package.

This package is part of the standard **SAP R/3 System** – that is, it is always delivered with the ABAP Workbench.

You can also analyze and execute these and other demonstration and test programs in a comfortable test environment. In the ABAP Workbench, choose *Environment* → *Examples* → *Controls examples*.

# Exercise 12: Events of the SAP Grid Control

## Exercise Objectives

After completing this exercise, you will be able to:

- Use the double\_click event of the SAP Grid Control to implement interactive reporting.

## Business Example

Allow users to choose a line in the basic list (of SPFLI data) by double-clicking it. When this occurs, your program should display a list of all flights (from SFLIGHT) for the connection the user has chosen in a modeless dialog box. Use an SAP Grid Control to display the detail list and basic list.

**Program:** ZBC412\_##\_GRD\_EX3

**Template:** SAPBC412\_GRDS\_EXERCISE\_1

**Model solution:** SAPBC412\_GRDS\_EXERCISE\_3

where ## is the group number.

### Task 1:

Copy the Template.

1. Copy your solution from the **first** exercise in this chapter (ZBC412\_##\_GRD\_EX1) or the appropriate model solution (SAPBC412\_GRDS\_EXERCISE\_1) to the name ZBC412\_##\_GRD\_EX3. Get to know how your copy of the program works.

### Task 2:

Define and implement an event handler for the double\_click event of the SAP Grid Control. Proceed as follows:

1. Find out about the interface of the *double\_click* event in the SAP Grid Control class.
2. Define a local class (suggested name: *lcl\_event\_handler*). The class should contain a static method for handling the double\_click event of instances of the class *c1\_gui\_alv\_grid*.
3. Implement the *on\_double\_click* method in the implementation part of the local class. The method should contain the following steps:
  - a) Find out which data record has been selected

*Continued on next page*

- b) Get the flight details for that record (SFLIGHT)
- c) Instantiate other controls
- d) Transfer the SFLIGHT data obtained.

Proceed as follows:

Read the flight data record chosen from the internal table it\_spfli, using the READ TABLE ... INDEX... statement. If an error occurs, display message 075 from the message class BC412 and terminate processing of the handler method.

Define a suitable global internal table (suggested name: it\_popup), which will be used by the second SAP Grid Control. Copy the appropriate flight data from the internal table it\_sflight into the new internal table.

Create a dialog box container and a second SAP Grid Control. To do this, define a suitable global reference variable (suggested name: ref\_box, ref\_box\_alv). Use the following values for the interface parameters of the dialog box container constructor:

width	800
height	200
top	120
left	120
caption	Any (suggested caption: Flight connections)

Handle errors using messages 010 and 045 from message class BC412.

Pass the flight data from SFLIGHT to the new SAP Grid Control, using the method set\_table\_for\_first\_display. Handle errors using message 012 from message class BC412.

4. Register your event handler method with the SAP Grid Control instance for the basic list (in the *init\_control\_processing* module).
5. Activate and test your program

### Task 3:

Make sure that the program does not create a new window each time the user takes an action on the screen but updates the content of the existing window instead.

1. Make the program skip the instantiation of the dialog box container and the additional SAP Grid Control instance if such an instance already exists. To do this, use the reference variables ref\_box and ref\_box\_alv as indicators.

*Continued on next page*

2. Additionally, make sure the program only calls the `set_table_for_first_display` method for newly created SAP Grid Control instances. To update the contents of the detail list for an existing SAP Grid Control instance use instance method `refresh_table_display`.

## Solution 12: Events of the SAP Grid Control

### Task 1:

Copy the Template.

1. Copy your solution from the **first** exercise in this chapter (ZBC412\_##\_GRD\_EX1) or the appropriate model solution (SAPBC412\_GRDS\_EXERCISE\_1) to the name ZBC412\_##\_GRD\_EX3. Get to know how your copy of the program works.
  - a) –

### Task 2:

Define and implement an event handler for the double\_click event of the SAP Grid Control. Proceed as follows:

1. Find out about the interface of the *double\_click* event in the SAP Grid Control class.
  - a) –
2. Define a local class (suggested name: *lcl\_event\_handler*). The class should contain a static method for handling the *double\_click* event of instances of the class *cl\_gui\_alv\_grid*.
  - a)

```
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      on_double_click FOR EVENT double_click OF cl_gui_alv_grid
        IMPORTING e_row.
  ENDCLASS.
```

3. Implement the *on\_double\_click* method in the implementation part of the local class. The method should contain the following steps:
  - a) Find out which data record has been selected
  - b) Get the flight details for that record (SFLIGHT)
  - c) Instantiate other controls
  - d) Transfer the SFLIGHT data obtained.

Proceed as follows:

*Continued on next page*

Read the flight data record chosen from the internal table `it_spfli`, using the `READ TABLE ... INDEX...` statement. If an error occurs, display message 075 from the message class BC412 and terminate processing of the handler method.

Define a suitable global internal table (suggested name: `it_popup`), which will be used by the second SAP Grid Control. Copy the appropriate flight data from the internal table `it_sflight` into the new internal table.

Create a dialog box container and a second SAP Grid Control. To do this, define a suitable global reference variable (suggested name: `ref_box`, `ref_box_alv`). Use the following values for the interface parameters of the dialog box container constructor:

<code>width</code>	800
<code>height</code>	200
<code>top</code>	120
<code>left</code>	120
<code>caption</code>	Any (suggested caption: Flight connections)

Handle errors using messages 010 and 045 from message class BC412.

Pass the flight data from `SFLIGHT` to the new SAP Grid Control, using the method `set_table_for_first_display`. Handle errors using message 012 from message class BC412.

a)

```

CLASS lcl_event_handler IMPLEMENTATION.
  METHOD on_double_click.
    *   find out selected line (double click)
    READ TABLE it_spfli INTO wa_spfli INDEX e_row-index.
    IF sy-subrc NE 0.
      MESSAGE i075(bc412). " internal error
      EXIT.
    ENDIF.
    ...
  ENDMETHOD.
ENDCLASS.

```

b)

```

DATA:
...
it_popup      TYPE TABLE OF sflight.

```

*Continued on next page*

```

CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_double_click.
...
*   copy corresponding flight data to it_popup
CLEAR it_popup.

LOOP AT it_sflight INTO wa_sflight
      WHERE carrid = wa_spfli-carrid
      AND connid = wa_spfli-connid.
      INSERT wa_sflight INTO TABLE it_popup.
ENDLOOP.

...
ENDMETHOD.

ENDCLASS.

```

c)

```

DATA:
...
ref_box      TYPE REF TO cl_gui_dialogbox_container,
ref_box_alv  TYPE REF TO cl_gui_alv_grid.

CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_double_click.
...
IF ref_box IS INITIAL.
*   create dialog box container
CREATE OBJECT ref_box
EXPORTING
width          = 800
height         = 200
top            = 120
left           = 120
caption        = text-002
EXCEPTIONS
others         = 1.
IF sy-subrc <> 0.
MESSAGE a010(bc412).
ENDIF.
ENDIF.
IF ref_box_alv IS INITIAL.
*   create avl grid object and link to dialogbox container
CREATE OBJECT ref_box_alv
EXPORTING
i_parent       = ref_box
EXCEPTIONS

```

*Continued on next page*

```

          others      = 1.
IF sy-subrc <> 0.
MESSAGE a045(bc412).
ENDIF.

...
ENDIF.
ENDMETHOD.
ENDCLASS.
```

4. Register your event handler method with the SAP Grid Control instance for the basic list (in the *init\_control\_processing* module).
  - a)

```
SET HANDLER lcl_event_handler=>on_double_click FOR ref_alv.
```

5. Activate and test your program
  - a) -

### Task 3:

Make sure that the program does not create a new window each time the user takes an action on the screen but updates the content of the existing window instead.

1. Make the program skip the instantiation of the dialog box container and the additional SAP Grid Control instance if such an instance already exists. To do this, use the reference variables `ref_box` and `ref_box_alv` as indicators.
  - a) -
2. Additionally, make sure the program only calls the `set_table_for_first_display` method for newly created SAP Grid Control instances. To update the contents of the detail list for an existing SAP Grid Control instance use instance method `refresh_table_display`.
  - a)

```
CALL METHOD ref_box_alv->refresh_table_display.
```

### Result

The complete source code for the model solution is shown below.

#### ABAP program - Data Declarations

```
DATA:
```

*Continued on next page*

```

ok_code          TYPE sy-ucomm,
copy_ok_code    LIKE ok_code,

ref_container   TYPE REF TO cl_gui_docking_container,
ref_alv         TYPE REF TO cl_gui_alv_grid,

ref_box         TYPE REF TO cl_gui_dialogbox_container,
ref_box_alv    TYPE REF TO cl_gui_alv_grid,

it_spfli        TYPE TABLE OF spfli,
it_sflight      TYPE TABLE OF sflight,

it_popup        TYPE TABLE OF sflight,

wa_spfli        LIKE LINE OF it_spfli,
wa_sflight      LIKE LINE OF it_sflight.

```

### Local Classes

```

*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
CLASS lcl_event_handler DEFINITION.
PUBLIC SECTION.
CLASS-METHODS:
  on_double_click FOR EVENT double_click OF cl_gui_alv_grid
    IMPORTING e_row.

ENDCLASS.

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_double_click.
*  find out selected line (double click)
  READ TABLE it_spfli INTO wa_spfli INDEX e_row-index.
  IF sy-subrc NE 0.
    MESSAGE i075(bc412).  " internal error
    EXIT.
  ENDIF.

*  copy corresponding flight data to it_popup
  CLEAR it_popup.

```

*Continued on next page*

```

LOOP AT it_sflight INTO wa_sflight
  WHERE carrid = wa_spfli-carrid
    AND connid = wa_spfli-connid.
  INSERT wa_sflight INTO TABLE it_popup.
ENDLOOP.

IF ref_box IS INITIAL.

*   create dialog box container
  CREATE OBJECT ref_box
    EXPORTING
      width          = 800
      height         = 200
      top            = 120
      left           = 120
      caption        = text-002
    EXCEPTIONS
      others         = 1.

IF sy-subrc <> 0.
  MESSAGE a010(bc412).
ENDIF.

ENDIF.

IF ref_box_alv IS INITIAL.

*   create avl grid object and link to dialogbox container
  CREATE OBJECT ref_box_alv
    EXPORTING
      i_parent       = ref_box
    EXCEPTIONS
      others         = 1.

IF sy-subrc <> 0.
  MESSAGE a045(bc412).
ENDIF.

*   send popup data to new alv object
  CALL METHOD ref_box_alv->set_table_for_first_display
    EXPORTING
      i_structure_name = 'SFLIGHT'
    CHANGING
      it_outtab        = it_popup

```

*Continued on next page*

```

*      IT_FIELDCATALOG      =
*      IT_SORT                =
*      IT_FILTER               =
EXCEPTIONS
      OTHERS                  = 1.

IF sy-subrc <> 0.
  MESSAGE a012(bc412).
ENDIF.

ELSE.                                " do only refresh alv contents
  CALL METHOD ref_box_alv->refresh_table_display.

ENDIF.
ENDMETHOD.
ENDCLASS.
```

## Modules

```

*&-----*
*&     Module   INIT_CONTROL_PROCESSING  OUTPUT
*&-----*
*      control related processing
*-----*
MODULE init_control_processing OUTPUT.
  IF ref_container IS INITIAL.

  ...

*      send basic list to alv grid control

  CALL METHOD ref_alv->set_table_for_first_display
    EXPORTING
      i_structure_name      = 'SPFLI'
*      IS_VARIANT           =
*      I_SAVE                =
*      I_DEFAULT              = 'X'
*      IS_LAYOUT              =
*      IS_PRINT               =
*      IT_SPECIAL_GROUPS     =
*      IT_TOOLBAR_EXCLUDING  =
    CHANGING
      it_outtab              = it_spfli
```

*Continued on next page*

```
*      IT_FIELDCATALOG          =
*      IT_SORT                  =
*      IT_FILTER                =
EXCEPTIONS
OTHERS           = 1.

IF sy-subrc <> 0.
MESSAGE a012(bc412).
ENDIF.

*  event handling --> only ABAP Objects part,
*          CFW registration is performed by ALV proxy object

SET HANDLER lcl_event_handler=>on_double_click FOR ref_alv.

ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING  OUTPUT
```



# Exercise 13: Extending the Application Toolbar

## Exercise Objectives

After completing this exercise, you will be able to:

- Add new pushbuttons to the application toolbar
- Provide additional functions
- Show and hide elements on the application toolbar

## Business Example

Offer the user an additional pushbutton that lets him or her hide the standard elements on the application toolbar. Your program should also display another pushbutton as a single element, allowing users to show the standard elements and the Hide button again.

**Program:** ZBC412\_##\_GRD\_EX4

**Template:** SAPBC412\_GRDS\_EXERCISE\_1

**Model solution:** SAPBC412\_GRDS\_EXERCISE\_4

where ## is the group number.

### Task 1:

Copy the Template.

1. Copy your solution from the **first** exercise in this chapter (ZBC412\_##\_GRD\_EX1) or the appropriate model solution (SAPBC412\_GRDS\_EXERCISE\_1), to the name ZBC412\_##\_GRD\_EX4. Get to know how your copy of the program works.

### Task 2:

Extend the application toolbar of the SAP Grid Control in a context-sensitive way by adding a *Show* function and a *Hide* function as follows:

1. Define a static method to handle the `toolbar` event (suggested name: `on_toolbar`). Become familiar with the export parameters of this event.
2. Implement the method:

You need a work area for the function attributes. Define this locally in the method (we suggest the name: `l_wa_button`).

*Continued on next page*

Fill the fields with appropriate function codes, icons (use the type pool icon), tooltips, and key types – depending on whether the standard application toolbar is to be shown or hidden.

Suggested values:

```
function      'EXCLUDE' / 'INCLUDE'
icon          icon_pdir_foreward_switch /
                icon_pdir_back_switch
quickinfo    "Show buttons" / "Hide buttons"
butn_type    '0'
```

For simplicity's sake, use a special "OK code field" for the SAP Grid Control to differentiate between the two options. Define this globally in the program (suggested name: `alv_ok_code`). It will be filled with a meaningful value later.

Initialize the internal table with the standard application toolbar (→ `mt_toolbar`) for the case that the application toolbar is to be hidden – except for one button. Import a reference to this instance of the application toolbar using the `toolbar` event.

Insert the filled work area for the new pushbutton in the `instance` attribute for the application toolbar (→ `mt_toolbar`).

3. Register the method with the SAP Grid Control instance (in the `init_control_processing` module) before calling the method `set_table_for_first_display`.
4. Activate and test your program.

### Task 3:

Now define and implement a handler method for the new pushbuttons. Make sure that the `toolbar` event is raised within this handler method. This allows you to show or hide the toolbar using the already implemented handler method `on_toolbar`. Proceed as follows:

1. Define a static method to handle the `user_command` event (suggested name: `on_user_command`). Become familiar with the export parameters of this event.
2. Implement the method:

Import the function code from the `user_command` event and copy it to the "OK code field" `alv_ok_code`.

*Continued on next page*

Complete the implementation of the `on_toolbar` method. Differentiate between the two states of the toolbar (shown or hidden) using the content of the `alv_ok_code` field. Choose an appropriate default value for the OK code field `alv_ok_code`.

Make sure that the `toolbar` event is raised whenever the user tries to show or hide the toolbar by calling the `set_toolbar_interactive` method.

3. Register the handler method for the SAP Grid Control instance (in the `init_control_processing` module).

## Solution 13: Extending the Application Toolbar

### Task 1:

Copy the Template.

1. Copy your solution from the **first** exercise in this chapter (ZBC412\_##\_GRD\_EX1) or the appropriate model solution (SAPBC412\_GRDS\_EXERCISE\_1), to the name ZBC412\_##\_GRD\_EX4. Get to know how your copy of the program works.
  - a) –

### Task 2:

Extend the application toolbar of the SAP Grid Control in a context-sensitive way by adding a *Show* function and a *Hide* function as follows:

1. Define a static method to handle the `toolbar` event (suggested name: `on_toolbar`). Become familiar with the export parameters of this event.
  - a)

```
CLASS lcl_event_handler DEFINITION.  
  PUBLIC SECTION.  
    CLASS-METHODS:  
      on_toolbar      FOR EVENT toolbar OF cl_gui_alv_grid  
                      IMPORTING e_object,  
...  
  ENDCLASS.
```

2. Implement the method:

You need a work area for the function attributes. Define this locally in the method (we suggest the name: `l_wa_button`).

Fill the fields with appropriate function codes, icons (use the type pool `icon`), tooltips, and key types – depending on whether the standard application toolbar is to be shown or hidden.

Suggested values:

*Continued on next page*

<i>function</i>	'EXCLUDE' / 'INCLUDE'
<i>icon</i>	icon_pdir_foreward_switch / icon_pdir_back_switch
<i>quickinfo</i>	"Show buttons" / "Hide buttons"
<i>butn_type</i>	'0'

For simplicity's sake, use a special "OK code field" for the SAP Grid Control to differentiate between the two options. Define this globally in the program (suggested name: `alv_ok_code`). It will be filled with a meaningful value later.

Initialize the internal table with the standard application toolbar (→ `mt_toolbar`) for the case that the application toolbar is to be hidden – except for one button. Import a reference to this instance of the application toolbar using the `toolbar` event.

Insert the filled work area for the new pushbutton in the instance attribute for the application toolbar (→ `mt_toolbar`).

a)

```

DATA:
ok_code      TYPE sy-ucomm,
...
alv_ok_code  LIKE ok_code VALUE 'INCLUDE'.
...
CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_toolbar.
DATA l_wa_button TYPE stb_button.
CASE alv_ok_code.
WHEN 'INCLUDE'.
l_wa_button-function = 'EXCLUDE'.
l_wa_button-icon     = icon_pdir_foreward_switch.
l_wa_button-quickinfo = text-ecl. "Hide Functions
WHEN 'EXCLUDE'.
CLEAR e_object->mt_toolbar.
l_wa_button-function = 'INCLUDE'.
l_wa_button-icon     = icon_pdir_back_switch.
l_wa_button-quickinfo = text-icl. "Show Functions
ENDCASE.
l_wa_button-butn_type = 0.
INSERT l_wa_button INTO TABLE e_object->mt_toolbar.
ENDMETHOD.

```

*Continued on next page*

3. Register the method with the SAP Grid Control instance (in the `init_control_processing` module) before calling the method `set_table_for_first_display`.

a)

```

IF ref_container IS INITIAL.
  ...
  SET HANDLER:
    lcl_event_handler=>on_toolbar      FOR ref_alv.
  ...
ENDIF.
```

4. Activate and test your program.

a) –

### Task 3:

Now define and implement a handler method for the new pushbuttons. Make sure that the `toolbar` event is raised within this handler method. This allows you to show or hide the toolbar using the already implemented handler method `on_toolbar`. Proceed as follows:

1. Define a static method to handle the `user_command` event (suggested name: `on_user_command`). Become familiar with the export parameters of this event.

a)

```

CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
    ...
      on_user_command FOR EVENT
        user_command OF cl_gui_alv_grid
        IMPORTING e_ucomm.
  ENDCLASS.
```

2. Implement the method:

Import the function code from the `user_command` event and copy it to the “OK code field” `alv_ok_code`.

Complete the implementation of the `on_toolbar` method. Differentiate between the two states of the toolbar (shown or hidden) using the content of the `alv_ok_code` field. Choose an appropriate default value for the OK code field `alv_ok_code`.

*Continued on next page*

Make sure that the toolbar event is raised whenever the user tries to show or hide the toolbar by calling the `set_toolbar_interactive` method.

a)

```
METHOD on_user_command.
CASE e_ucomm.
  WHEN 'EXCLUDE' OR 'INCLUDE'.
    alv_ok_code = e_ucomm.
    CALL METHOD ref_alv->set_toolbar_interactive.
ENDCASE.
ENDMETHOD.
```

3. Register the handler method for the SAP Grid Control instance (in the `init_control_processing` module).

a)

```
SET HANDLER:
...
lcl_event_handler=>on_user_command FOR ref_alv.
```

## Result

### ABAP program

```
REPORT  sapbc412_grds_exercise_4 MESSAGE-ID bc412.
```

```
TYPE-POOLS icon.
```

```
DATA:
ok_code      TYPE sy-ucomm,
copy_ok_code LIKE ok_code,
alv_ok_code  LIKE ok_code VALUE 'INCLUDE',
...

```

### Local Classes

```
-----*
*      CLASS lcl_event_handler DEFINITION
*-----
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
  CLASS-METHODS:
    on_toolbar      FOR EVENT toolbar OF cl_gui_alv_grid
                  IMPORTING e_object,
  -----*
```

*Continued on next page*

```

        on_user_command FOR EVENT user_command
            OF cl_gui_alv_grid
                IMPORTING e_ucomm.

ENDCLASS.

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

METHOD on_toolbar.
    DATA l_wa_button TYPE stb_button.

    CASE alv_ok_code.
        WHEN 'INCLUDE'.
            l_wa_button-function = 'EXCLUDE'.
            l_wa_button-icon     = icon_pdir_foreward_switch.
            l_wa_button-quikinfo = text-ecl. "Hide Functions
        WHEN 'EXCLUDE'.
            CLEAR e_object->mt_toolbar.
            l_wa_button-function = 'INCLUDE'.
            l_wa_button-icon     = icon_pdir_back_switch.
            l_wa_button-quikinfo = text-icl. "Show Functions
    ENDCASE.

    l_wa_button-butn_type = 0.
    INSERT l_wa_button INTO TABLE e_object->mt_toolbar.

ENDMETHOD.

* -----
METHOD on_user_command.
    CASE e_ucomm.
        WHEN 'EXCLUDE' OR 'INCLUDE'.
            alv_ok_code = e_ucomm.
            CALL METHOD ref_alv->set_toolbar_interactive.
    ENDCASE.
ENDMETHOD.

ENDCLASS.

```

## Modules

*Continued on next page*

```

*&-----*
*&     Module  INIT_CONTROL_PROCESSING  OUTPUT
*&-----*
*      control related processing
*-----*
MODULE init_control_processing OUTPUT.
IF ref_container IS INITIAL.

    ...

SET HANDLER:
    lcl_event_handler=>on_toolbar      FOR ref_alv,
    lcl_event_handler=>on_user_command FOR ref_alv.

*   send basic list to alv grid control
CALL METHOD ref_alv->set_table_for_first_display
    EXPORTING
*       I_BUFFER_ACTIVE          =
*       i_structure_name         = 'SPFLI'
*       IS_VARIANT               =
*       I_SAVE                   =
*       I_DEFAULT                = 'X'
*       IS_LAYOUT                =
*       IS_PRINT                 =
*       IT_SPECIAL_GROUPS        =
*       IT_TOOLBAR_EXCLUDING     =
*       IT_HYPERLINK             =
*       IT_ALV_GRAPHICS          =
    CHANGING
        it_outtab                = it_spfli
*       IT_FIELDCATALOG          =
*       IT_SORT                  =
*       IT_FILTER                =
    EXCEPTIONS
        OTHERS                   = 4.

...
ENDIF.

ENDMODULE.          " INIT_CONTROL_PROCESSING  OUTPUT

```



## Lesson Summary

You should now be able to:

- Explain how events are triggered and how they can be handled
- Identify existing events
- Create and register an event handler method
- Add new functionality to the standard toolbar
- Enhance the print list by means of event handling



## Unit Summary

You should now be able to:

- Name the default functions of the SAP Grid Control
- Describe the screen layout of the SAP Grid Control
- Describe the data area layout of the SAP Grid Control
- Describe the print list layout of the SAP Grid Control
- Understand the technical view of the SAP Grid Control
- Pass list data, the field catalog and additional information to the SAP Grid Control instance
- Work with layout variants in a SAP Grid Control
- Fill and pass the layout structure of the SAP Grid Control
- Explain the purpose of the field catalogue
- Add columns to or change columns of the SAP Grid Control
- Handle colors of rows and cells in the SAP Grid Control
- Suppress standard functionality of the toolbar
- Explain how events are triggered and how they can be handled
- Identify existing events
- Create and register an event handler method
- Add new functionality to the standard toolbar
- Enhance the print list by means of event handling



# Unit 7

## Tree Control

### Unit Overview

This unit covers the layout and features of the three types of Tree Controls. Also, the unit presents the selected methods and events plus the data description.



### Unit Objectives

After completing this unit, you will be able to:

- Name the different type of Tree Control and describe their main features
- Display data in the different Tree Control variants
- Create hierarchies
- Integrate additional data if necessary
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container
- Create a hierarchy in a simple tree
- Types of relationships
- Passing node entries
- Print the Tree Content
- Search in the Tree
- Name some of the events raised by the Tree Control
- Find the application data belonging to the node raising the event
- Create an application toolbar
- Use the additional functions of the Tree Model Classes

### Unit Contents

Lesson: Tree Control: Introduction .....	349
Lesson: Tree Control: Creating a Tree Control Instance .....	361
Exercise 14: Display Data in a Simple Tree Model .....	365
Lesson: Tree Control: Creating a Hierarchy .....	380
Lesson: Tree Control: Additional Functions of the Tree Model .....	389
Lesson: Tree Control: Events .....	392
Exercise 15: Application Toolbar .....	399

Lesson: Tree Control: Additional Data in the Column and List Trees....	419
Exercise 16: Display Data in a Column Tree Model .....	429
Exercise 17: Icons as Column Entries (OPTIONAL).....	445

# Lesson: Tree Control: Introduction

## Lesson Overview

This lesson presents the three different types of the Tree Control and their main features.



## Lesson Objectives

After completing this lesson, you will be able to:

- Name the different type of Tree Control and describe their main features

## Business Example

The application to be implemented requires the use of a Tree Control. You want to know the different types of the Tree Control to be able to choose the proper one.

## Tree Control



Object name	Description
▽ BC412	Training course: BC...
▷ □ Dictionary objects	
▷ □ Class Library	
▽ □ Programs	
SAPBC412_ALVD_FUNC...	BC412: Demo ...
SAPBC412_BASD_AREA...	BC412: Demo ...
...	...
...	...
...	...
▷ □ Function groups	
▷ □ Message classes	

- Data displayed hierarchically in a list
- List can be expanded and collapsed
- Used as a navigational aid, value list, and object list

Figure 150: Tree Control

The tree control is suitable for displaying data in **hierarchical structures**.

The tree control shows its data in the form of a list. You can display the tree in either expanded or collapsed form. It is therefore particularly useful for displaying data in a structured format. You can use it as a

- navigation aid
- value list
- item list.

Like the SAP Grid Control, the tree control is used to display and manage data. You cannot enter data as you would in a text editor. However, user interaction is possible with the following components:

- Single lines
- Areas consisting of several lines
- Elements within lines
- Header elements

## Screen Layout of the Tree Control

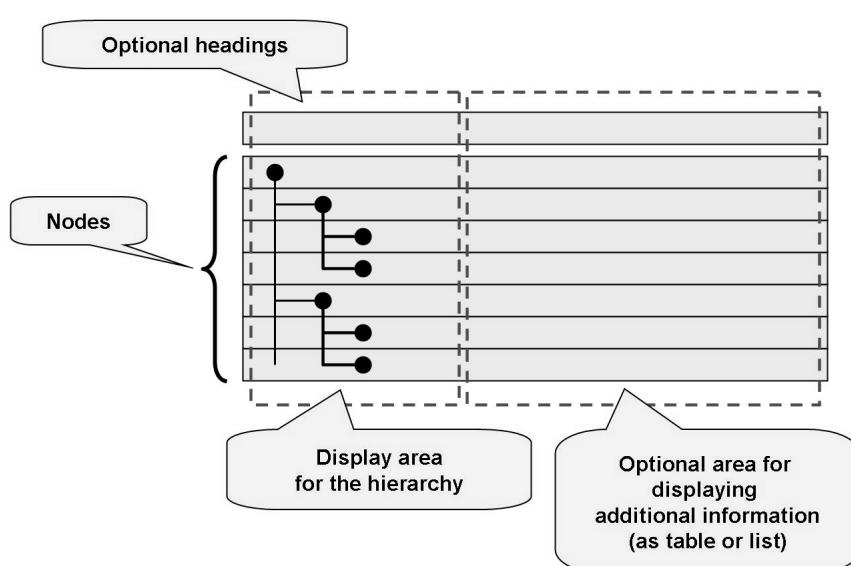


Figure 151: Screen Layout of the Tree Control

The screen layout of the tree control is made up of three areas:

- The hierarchy area, in which the hierarchy data is displayed as **nodes**.
- An (optional) additional area, where you can display other data belonging to an individual node. This data can be in the form of a table or a list
- Another optional area, displaying headings.

In many cases, the node texts belong to the optional additional data area for technical reasons.



## Types of Tree Control

Areas used	Tree Control form	Name of global class used
	Simple Tree	<code>CL_SIMPLE_TREE_MODEL (CL_GUI_SIMPLE_TREE)</code>
	Column Tree	<code>CL_COLUMN_TREE_MODEL (CL_GUI_COLUMN_TREE)</code>
	List Tree	<code>CL_LIST_TREE_MODEL (CL_GUI_LIST_TREE)</code>

**Figure 152: Types of Tree Control**

There are three different tree control implementations. Each one uses the three areas illustrated above differently.

The **simple tree** only uses the hierarchy area. It has no headings, and cannot display any special structured data.

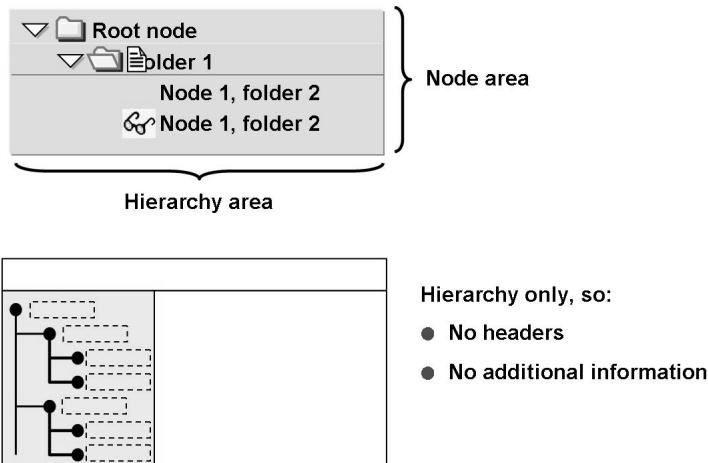
Conversely, the **Column Tree** and **List Tree** structure the data in the additional data area in different ways:

From SAP R/3 Basis Release 4.6C onwards, there are three global “tree model” classes, to make it easier for you to implement the tree control. These classes use the global classes shown in parentheses above.

This chapter deals with the use of the tree model classes

For the SAP R/3 Basis Releases 4.6A and 4.6B you must use the classes in parentheses directly, although the implementation techniques described in this chapter are similar in these older releases. In some cases, you need to obtain further details or information on other techniques from the online documentation.

## Displaying Data in a Simple Tree



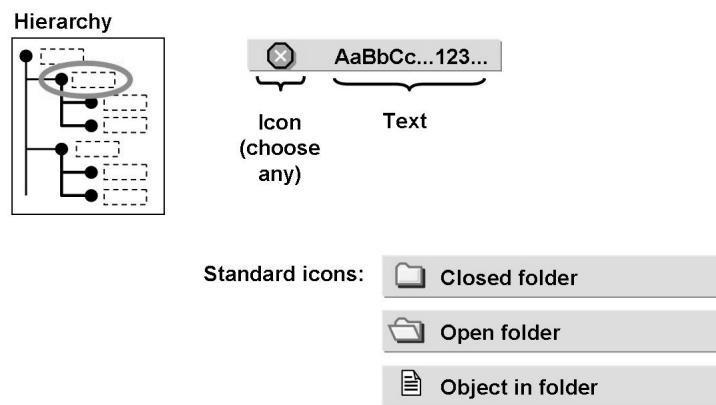
**Figure 153: Displaying Data in a Simple Tree**

The Simple Tree only displays data in a hierarchy area.

The data to be displayed is stored as node texts.

You cannot use the areas intended for headings or additional information.

## Displaying Nodes in a Simple Tree



**Figure 154: Displaying Nodes in a Simple Tree**

Nodes in a simple tree can consist of an icon and a text.

You can use any icon and any text.

As a rule, you use folder symbols to indicate that the node has subnodes.

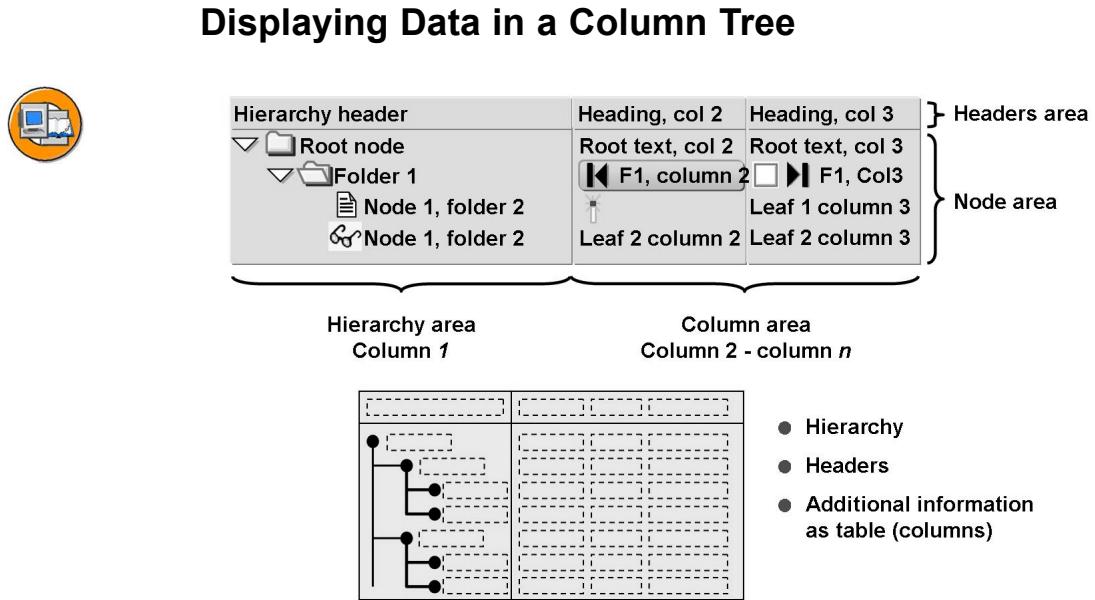


Figure 155: Displaying Data in a Column Tree

The Column Tree uses the hierarchy area, headings, and the additional information area.

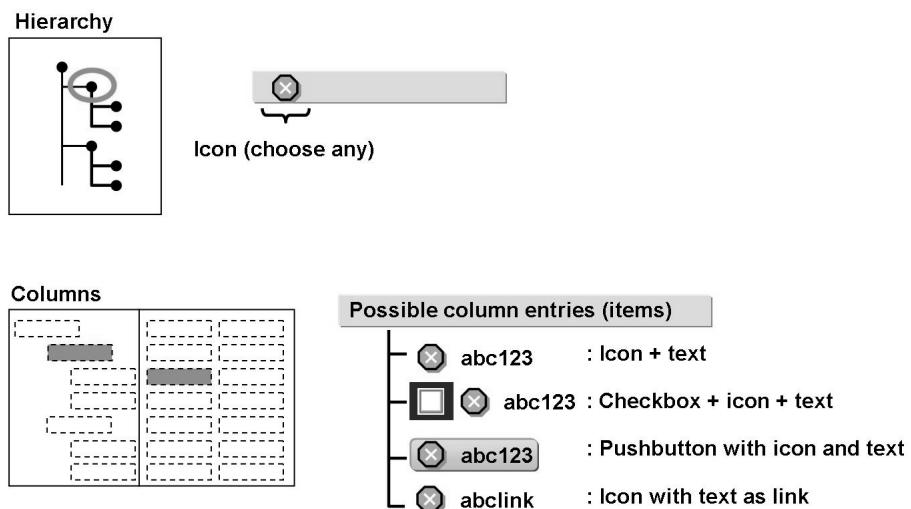
The contents of the **first column** are displayed **in the hierarchy area**. They appear left-justified next to the node.

All of the other columns are displayed in the additional information area. They are aligned vertically (displayed as a table).

You can use the headings area as both a hierarchy area and a column area



## Displaying Nodes in a Column Tree



**Figure 156: Displaying Nodes in a Column Tree**

The nodes in the column tree consist of one icon **only**.

You can display items in the columns as follows:

- As an icon with text (both are optional)
- As a checkbox with icon and text (icon and text are optional)
- As a pushbutton with icon and text (icon and text are optional)
- As a link (text displayed as hotspot. The mouse pointer changes shape when it is positioned over a link.)

If you want to create columns that consist entirely of mouse operation elements, display the data as links for ergonomic reasons.

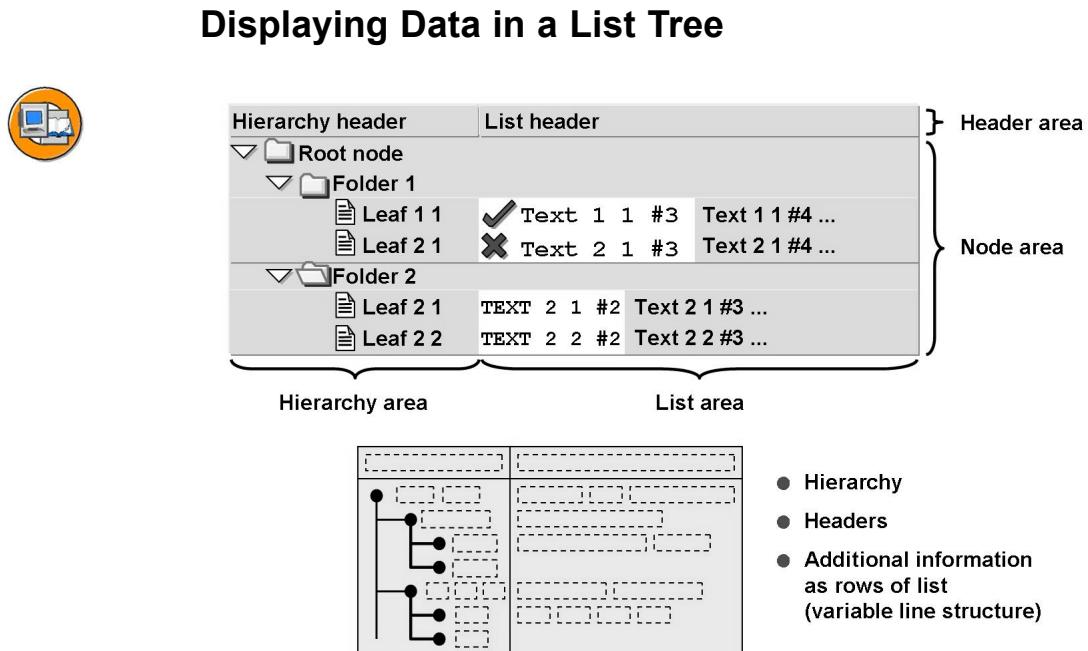


Figure 157: Displaying Data in a List Tree

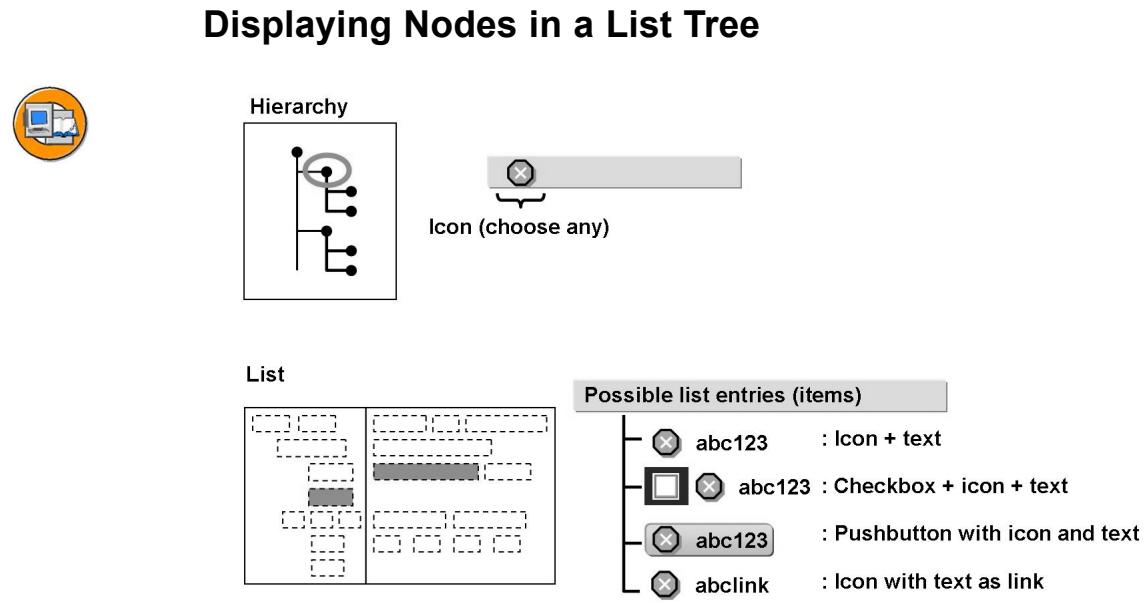
The List Tree uses the hierarchy area, headings, and the additional information area.

In contrast to the Column Tree, the List Tree is **not** structured as a table. Each line (node) can have a different structure. The additional information is displayed left-aligned after each node. The display width is adapted to the width of the information.

When you display text in a List Tree, you can specify the font size for each entry, along with whether the system should use a proportional or non-proportional font.

For each node, you can specify the number of entries you want to display in the hierarchy area.

You can use the headings area as both a hierarchy area and a list area



**Figure 158: Displaying Nodes in a List Tree**

The nodes in the list tree consist of one icon **only**.

You can display the items of a node as follows:

- As an icon with text (both are optional)
- As a checkbox with icon and text (icon and text are optional)
- As a pushbutton with icon and text (icon and text are optional)
- As a link

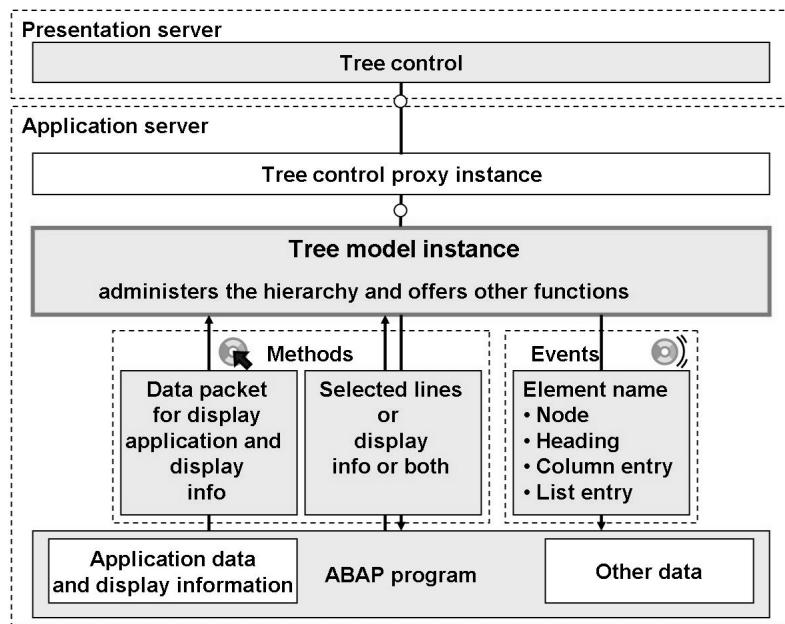
You should avoid excessive use of pushbuttons.

The List Tree allows you to choose between a proportional and non-proportional font for text entries.

For entries with proportional font, you should use the Automatic length attribute.



## The Tree Model Instance and the Proxy Instance of the Tree Control



**Figure 159: The Tree Model Instance and the Proxy Instance of the Tree Control**

A proxy instance of the CL\_GUI\_...\_TREE class encapsulates the technical details of communication with the tree control instance on the presentation server, as for other EnjoySAP Controls

Conversely, the calling program has to provide all information that cannot meaningfully be encapsulated in the proxy instance. This includes both the data to be displayed and the hierarchical relationships between data items. To do this, you must add special data objects to the program.

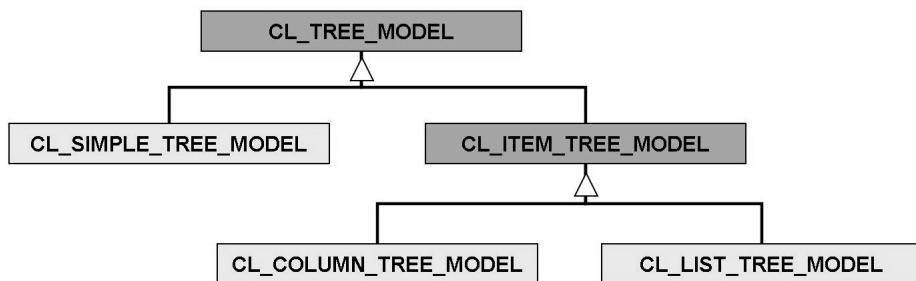
From Release 4.6C onwards, there are three additional tree model classes available. Instances of these classes should be used as an “intermediate layer” between the application data and the tree control proxy instance. Among other functions, they allow you to:

- Manage the passed data in a hierarchical table
- Use technical node IDs of any length
- Search through or print the contents of the application tree.
- Transport data between the calling program and the tree control on the presentation server in a way that optimizes runtime performance

Communication through the ABAP program – tree model instance – proxy object – tree control chain takes place interactively at runtime.

→ **Note:** You can only display character-type data formats with the tree control. Therefore, the application program must format all data itself.

## Inheritance Hierarchy of the Tree Model Classes



**Figure 160: Inheritance Hierarchy of the Tree Model Classes**

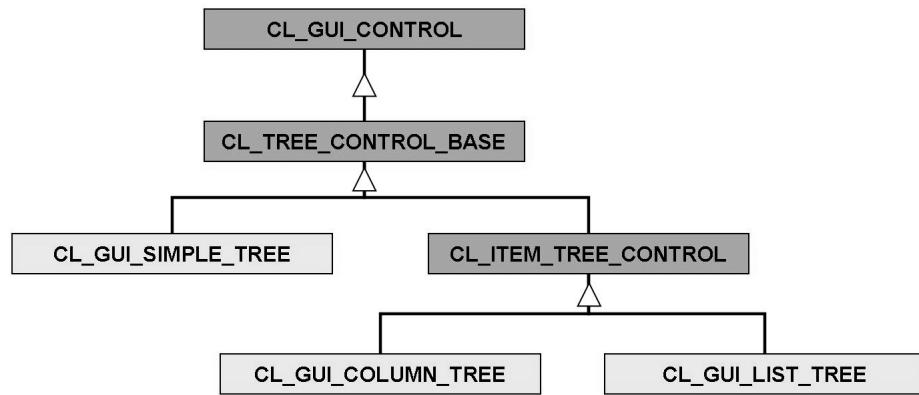
There are three global tree model classes corresponding to the three forms of the tree control:

- CL\_SIMPLE\_TREE\_MODEL
- CL\_COLUMN\_TREE\_MODEL
- CL\_LIST\_TREE\_MODEL

They encapsulate the additional functions and communication with the tree control proxy instances, discussed previously.

In your own applications, you should only ever instantiate these three classes directly.

## Inheritance Hierarchy of the Tree Control Classes



**Figure 161: Inheritance Hierarchy of the Tree Control Classes**

The actual tree control proxy instances, themselves used by the tree model instances, are created from the three classes shown above.

In your own applications, you need not instantiate these classes or communicate with them, unless you are working with SAP R/3 Basis Release 4.6A or 4.6B.



## Lesson Summary

You should now be able to:

- Name the different type of Tree Control and describe their main features

# Lesson: Tree Control: Creating a Tree Control Instance

## Lesson Overview

This lesson will present the techniques of displaying data in the different Tree Controls.



## Lesson Objectives

After completing this lesson, you will be able to:

- Display data in the different Tree Control variants
- Create hierarchies
- Integrate additional data if necessary
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container

## Business Example

The application to be implemented needs to display data in a Tree Control. You should create hierarchies and integrate additional data if necessary

## Creating a Simple Tree Instance

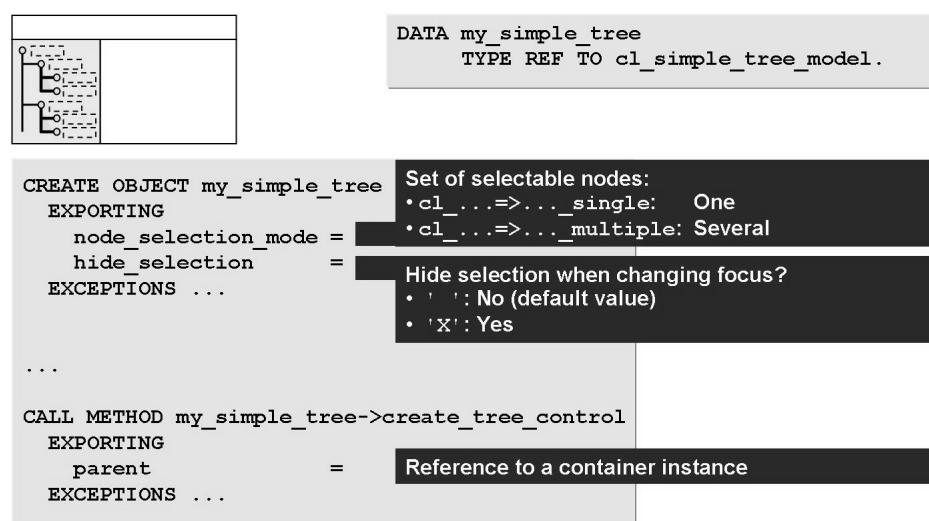


Figure 162: Creating a Simple Tree Instance

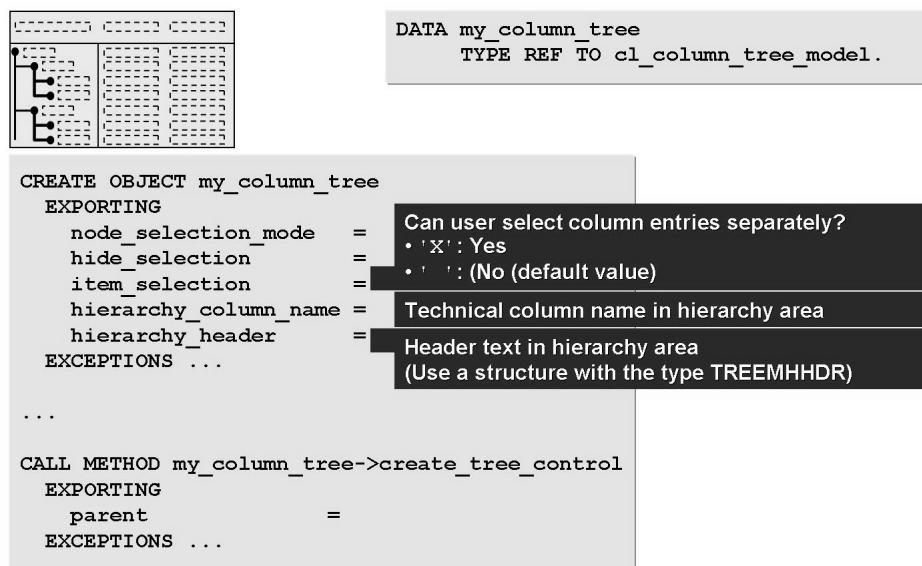
The “intermediary” role of tree model instances is particularly clear when you create tree control instances:

Firstly, you create a tree **model** instance, here a simple tree model. Thus you create a reference to the appropriate tree model class, here CL\_SIMPLE\_TREE\_MODEL.

The constructor allows you to set the general behavior of the tree (which has not yet been created). You specify the set of selectable nodes using the class constants. (cl\_simple\_tree\_model=>node\_sel\_mode\_single or cl\_simple\_tree\_model=>node\_sel\_mode\_multiple).

To create the actual **tree control proxy instance**, call a method of the **tree model instance** CREATE\_TREE\_CONTROL. Only now you can specify the container in which you will place the tree control.

## Creating a Column Tree Instance



**Figure 163: Creating a Column Tree Instance**

The procedure is similar to that for the simple tree. This slide deals only with those **additional** parameters that you can or need to fill **specifically** for the column tree.

You will need the technical column names for the hierarchy area in order to display application data in this area as well.

To fill the header text parameter you need a structure with the global type TREEMHDR. You must fill the HEADING field before you call the constructor.

## Creating a List Tree Instance



`DATA my_list_tree  
TYPE REF TO cl_list_tree_model.`

```

CREATE OBJECT my_list_tree
EXPORTING
  node_selection_mode =
  hide_selection =
  item_selection =
  with_headers =
  hierarchy_header =
  list_header =
EXCEPTIONS ...
...
CALL METHOD my_list_tree->create_tree_control
EXPORTING
  parent =
EXCEPTIONS ...

```

Can user select list entries separately?  
 • 'X': Yes  
 • '' : No (default value)

Use list heading?  
 • 'X': Yes  
 • '' : No (default value)

Header text in hierarchy area  
 (Use a structure with the type TREEMLHDR)

Figure 164: Creating a List Tree Instance

The procedure is similar to that for the column tree. This slide deals only with those **additional** parameters that you can or need to fill **specifically** or in a differently for the column tree.

In the list tree, you can set headings either for the hierarchy area only, or for both the list and hierarchy areas. Use list headings only if you want to display data in a tabular form but want to use the list tree control instead of the column tree control for technical reasons.

To fill the header text parameter you need a structure with the global type TREEMLHDR. You must fill the HEADING field before you call the constructor.



# Exercise 14: Display Data in a Simple Tree Model

## Exercise Objectives

After completing this exercise, you will be able to:

- Display hierarchically arranged key data in a simple tree model instance.

## Business Example

Display data from the database tables SCARR (airlines), SPFLI (connections) and SFLIGHT (flights) hierarchically in a tree model.

**Program:** ZBC412\_##\_TRM\_EX1

**Template:** SAPBC412\_TRMT\_EXERCISE\_1

**Model solution:** SAPBC412\_TRMS\_EXERCISE\_1

where ## is the group number.

### Task 1:

Copy the Template.

1. Copy the model solution SAPBC412\_TRMT\_EXERCISE\_1 to the new name ZBC412\_##\_TRM\_EX1 and get to know how it works.

### Task 2:

In the PBO module init\_tree\_processing\_0100, create a simple tree model instance, which is displayed in the existing docking container. Proceed as follows:

1. Create a reference variable for the simple tree model instance (We suggest you use the name ref\_tree\_model). Use the class cl\_simple\_tree\_model to specify the type.
2. In the PBO module init\_container\_processing\_0100 create an instance of the simple tree model. Users should only be able to select one node at a time. If an error occurs while the object is being generated, your program should terminate, displaying the message 043.

Create a proxy instance for the simple tree control and link it with the docking container control. Use the method create\_tree\_control of the tree model instance. If an error occurs, the program should terminate, displaying the message 042.

*Continued on next page*

### Task 3:

Send some values from the data (buffered in the internal table `it_scarr`, `it_spfli` and `it_sfflight`) to your tree model instance as node entries. To do this, use the existing (empty) subroutine `add_nodes` that you call from the `init_tree_processing_0100` module. Proceed as follows:

1. In the subroutine `add_nodes`, define a local work area for a node entry (suggested name: `l_wa_node`) with the type `treemsnodi`.
2. For each node entry to be added, call the `add_node` method. If errors occur, terminate processing and display the message 012. Fill the parameters `node_key`, `relative_node_key`, `relationship`, `folder`, `text`, and `expander` with meaningful values, either directly or by using the work area `l_wa_node`.

#### Other notes on single parameters:

Using the **CONCATENATE** statement, create the **node IDs** from the application keys, making sure they are unique.

Use the **relationship type** "last child".

For the time being, use the primary key value as a **text entry**. Once your program runs error-free, you can improve this aspect.



**Hint:** Use an auxiliary variable with the type `c`, length 10 (suggested name: `date_text`) to display the value of the `wa_sfflight-fldate` field formatted. The `WRITE TO` statement applies the same formats to data as the `WRITE` statement.

## Solution 14: Display Data in a Simple Tree Model

### Task 1:

Copy the Template.

1. Copy the model solution SAPBC412\_TRMT\_EXERCISE\_1 to the new name ZBC412\_##\_TRM\_EX1 and get to know how it works.
  - a) -

### Task 2:

In the PBO module init\_tree\_processing\_0100, create a simple tree model instance, which is displayed in the existing docking container. Proceed as follows:

1. Create a reference variable for the simple tree model instance (We suggest you use the name ref\_tree\_model). Use the class cl\_simple\_tree\_model to specify the type.
  - a)

```
DATA:  
...  
    ref_tree_model TYPE REF TO cl_simple_tree_model.
```

2. In the PBO module init\_container\_processing\_0100 create an instance of the simple tree model. Users should only be able to select one node at a time. If an error occurs while the object is being generated, your program should terminate, displaying the message 043.

*Continued on next page*

Create a proxy instance for the simple tree control and link it with the docking container control. Use the method `create_tree_control` of the tree model instance. If an error occurs, the program should terminate, displaying the message 042.

a)

```

IF ref_tree_model IS INITIAL.
  CREATE OBJECT ref_tree_model
    EXPORTING
      node_selection_mode = cl_simple_tree_model->node_sel_mode_single
    EXCEPTIONS
      others                  = 1.
    IF sy-subrc <> 0.
      MESSAGE a043.
    ENDIF.

  CALL METHOD ref_tree_model->create_tree_control
    EXPORTING
      parent                  = ref_cont_left
    EXCEPTIONS
      OTHERS                  = 1.
    IF sy-subrc <> 0.
      MESSAGE a042.
    ENDIF.
  ENDIF.

```

### Task 3:

Send some values from the data (buffered in the internal table `it_scarr`, `it_spfli` and `it_sfflight`) to your tree model instance as node entries. To do this, use the existing (empty) subroutine `add_nodes` that you call from the `init_tree_processing_0100` module. Proceed as follows:

1. In the subroutine `add_nodes`, define a local work area for a node entry (suggested name: `l_wa_node`) with the type `treemsnodi`.

a)

```

FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_simple_tree_model.
  DATA:
    l_wa_node TYPE treemsnodi.
  ...

```

*Continued on next page*

2. For each node entry to be added, call the `add_node` method. If errors occur, terminate processing and display the message 012. Fill the parameters `node_key`, `relative_node_key`, `relationship`, `folder`, `text`, and `expander` with meaningful values, either directly or by using the work area `l_wa_node`.

**Other notes on single parameters:**

Using the **CONCATENATE** statement, create the **node IDs** from the application keys, making sure they are unique.

Use the **relationship type** "last child".

For the time being, use the primary key value as a **text entry**. Once your program runs error-free, you can improve this aspect.



**Hint:** Use an auxiliary variable with the type `c`, length 10 (suggested name: `date_text`) to display the value of the `wa_sflicht-fldate` field formatted. The `WRITE TO` statement applies the same formats to data as the `WRITE` statement.

a)

```

FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_simple_tree_model.

DATA:
  l_wa_node TYPE treemsnodi,
  date_text(10) TYPE c.

  l_wa_node-text = text-car.

CALL METHOD l_ref_tree_model->add_node
  EXPORTING
    node_key           = 'ROOT'
    *      RELATIVE_NODE_KEY      =
    *      RELATIONSHIP          =
    isfolder           = 'X'
    text               = l_wa_node-text
    expander          = 'X'
  EXCEPTIONS
    OTHERS             = 5

  .
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

```

*Continued on next page*

```

* scarr-nodes:
LOOP AT it_scarr INTO wa_scarr.
  CLEAR l_wa_node.
  l_wa_node-node_key = wa_scarr-carrid.
  l_wa_node-text = wa_scarr-carrname.
  CALL METHOD l_ref_tree_model->add_node
    EXPORTING
      node_key          = l_wa_node-node_key
      relative_node_key = 'ROOT'
      relationship      = cl_simple_tree_model->relat_last_child
      isfolder          = 'X'
      text              = l_wa_node-text
      expander          = 'X'
    EXCEPTIONS
      OTHERS             = 5
    .
    IF sy-subrc <> 0.
      MESSAGE a012.
    ENDIF.
  ENDLOOP.

* spfli-nodes:
LOOP AT it_spfli INTO wa_spfli.
  CLEAR l_wa_node.
  CONCATENATE wa_spfli-carrid
    wa_spfli-connid
    INTO l_wa_node-node_key
    SEPARATED BY space.
  CONCATENATE wa_spfli-carrid
    wa_spfli-connid
    ':'
    wa_spfli-cityfrom
    '->'
    wa_spfli-cityto
    INTO l_wa_node-text
    SEPARATED BY space.
  l_wa_node-relatkey = wa_spfli-carrid.
  CALL METHOD l_ref_tree_model->add_node
    EXPORTING
      node_key          = l_wa_node-node_key
      relative_node_key = l_wa_node-relatkey
      relationship      = cl_simple_tree_model->relat_last_child
      isfolder          = 'X'
      text              = l_wa_node-text
      expander          = 'X'

```

*Continued on next page*

```
EXCEPTIONS
  OTHERS          = 5.
```

## Result

The complete source code for the model solution is shown below.

### Screen flow logic

#### *SCREEN 100*

```
PROCESS BEFORE OUTPUT.
  MODULE status_0100.
  MODULE init_container_processing_0100.
  MODULE init_tree_processing_0100.

PROCESS AFTER INPUT.
  MODULE exit_command_0100 AT EXIT-COMMAND.
  MODULE copy_ok_code.
  MODULE user_command_0100.
```

### ABAP program

#### *Data Declarations*

```
REPORT  sapbc412_trms_exercise_1 MESSAGE-ID bc412.

DATA:
  * screen-specific:
    ok_code TYPE sy-ucomm,
    copy_ok LIKE ok_code,

  * application data:
    it_scarr TYPE SORTED TABLE OF scarr
      WITH UNIQUE KEY carrid,
    it_spfli TYPE SORTED TABLE OF spfli
      WITH UNIQUE KEY carrid connid,
    it_sflight TYPE SORTED TABLE OF sflight
      WITH UNIQUE KEY carrid connid fldate,

    wa_scarr LIKE LINE OF it_scarr,
    wa_spfli LIKE LINE OF it_spfli,
    wa_sflight LIKE LINE OF it_sflight,

  * container:
```

*Continued on next page*

```

    ref_cont_left    TYPE REF TO cl_gui_docking_container,
    * content:
    ref_tree_model  TYPE REF TO cl_simple_tree_model.

```

### ***ABAP Program: Event Blocks***

```

* get application data:
SELECT * FROM scarr
      INTO TABLE it_scarr.
IF sy-subrc <> 0.
  MESSAGE a060.
ENDIF.

SELECT * FROM spfli
      INTO TABLE it_spfli.
IF sy-subrc <> 0.
  MESSAGE a060.
ENDIF.

SELECT * FROM sflight
      INTO TABLE it_sflight.
IF sy-subrc <> 0.
  MESSAGE a060.
ENDIF.

CALL SCREEN 100.

```

### ***Modules***

```

*&-----*
*&     Module STATUS_0100  OUTPUT
*&-----*
*      Set GUI title and GUI status for screen 100.
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'NORM_0100'.
  SET TITLEBAR 'TITLE_1'.
ENDMODULE.                                " STATUS_0100  OUTPUT

*&-----*
*&     Module INIT_CONTROL_PROCESSING  OUTPUT
*&-----*

```

*Continued on next page*

```

*      Start control handling.
*      create container object
*-----
MODULE init_container_processing_0100 OUTPUT.
IF ref_cont_left IS INITIAL.

      CREATE OBJECT ref_cont_left
      EXPORTING
          ratio           = 35
      EXCEPTIONS
          others          = 1.
      IF sy-subrc NE 0.
      MESSAGE a010.
      ENDIF.

      ENDIF.
ENDMODULE.                                     " INIT_CONTROL_PROCESSING  OUTPUT

*&-----
*&     Module  init_tree_processing_0100  OUTPUT
*&-----
*     create tree object, link to container and fill with data
*-----
MODULE init_tree_processing_0100 OUTPUT.
IF ref_tree_model IS INITIAL.

      CREATE OBJECT ref_tree_model
      EXPORTING
          node_selection_mode = cl_simple_tree_model->node_sel_mode_single
      EXCEPTIONS
          others             = 1.
      IF sy-subrc <> 0.
      MESSAGE a043.
      ENDIF.

      CALL METHOD ref_tree_model->create_tree_control
      EXPORTING
          parent            = ref_cont_left
      EXCEPTIONS
          OTHERS            = 1.
      IF sy-subrc <> 0.
      MESSAGE a042.
      ENDIF.

      PERFORM add_nodes USING ref_tree_model.

```

*Continued on next page*

```

ENDIF.

ENDMODULE.          " init_tree_processing_0100  OUTPUT

*-----*
*&      Module  EXIT_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type 'E' for screen 100.
*-----*

MODULE exit_command_0100 INPUT.
CASE ok_code.
WHEN 'CANCEL'.           " Cancel screen processing
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
WHEN 'EXIT'.              " Exit program
    PERFORM free_control_ressources.
    LEAVE PROGRAM.
WHEN OTHERS.
ENDCASE.
ENDMODULE.          " EXIT_COMMAND_0100  INPUT

*-----*
*&      Module  COPY_OK_CODE  INPUT
*-----*
*      Save the current user command in order to
*      prevent unintended field transport for the screen field ok_code
*      for next screen processing (ENTER)
*-----*

MODULE copy_ok_code INPUT.
copy_ok = ok_code.
CLEAR ok_code.
ENDMODULE.          " COPY_FCODE  INPUT

*-----*
*&      Module  USER_COMMAND_0100  INPUT
*-----*
*      Implementation of user commands of type ' ' for screen 100.
*-----*

MODULE user_command_0100 INPUT.
CASE copy_ok.
WHEN 'BACK'.             " Go back to program
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
WHEN OTHERS.
ENDCASE.

```

*Continued on next page*

```
ENDMODULE.                                     " USER_COMMAND_0100  INPUT
```

### **Subroutines**

```
*&-----*
*&      Form  FREE_CONTROL_RESSOURCES
*&-----*
*      free control ressources on the presentation server
*      free all reference variables (ABAP object) -> garbage collector
*-----*
*      no interface
*-----*
FORM free_control_ressources.
CALL METHOD:
    ref_cont_left->free.

FREE:
    ref_tree_model,
    ref_cont_left.

ENDFORM.                                     " FREE_CONTROL_RESSOURCES

*-----*
*      FORM ADD_NODES
*-----*
*      build up a hierarchy consisting of
*      carriers, connections and flight dates
*-----*
*  --> L_REF_TREE_MODEL
*-----*
FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_simple_tree_model.

DATA:
    l_wa_node TYPE treemsnodyn,
    date_text(10) TYPE c.

    l_wa_node-text = text-car.

CALL METHOD l_ref_tree_model->add_node
    EXPORTING
        node_key              = 'ROOT'
*        RELATIVE_NODE_KEY   =
*        RELATIONSHIP         =
        isfolder               = 'X'
```

*Continued on next page*

```

text          = l_wa_node-text
expander     = 'X'
EXCEPTIONS
OTHERS        = 5

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

* scarr-nodes:
LOOP AT it_scarr INTO wa_scarr.
CLEAR l_wa_node.
l_wa_node-node_key = wa_scarr-carrid.
l_wa_node-text = wa_scarr-carrname.
CALL METHOD l_ref_tree_model->add_node
EXPORTING
node_key      = l_wa_node-node_key
relative_node_key = 'ROOT'
relationship   = cl_simple_tree_model->relat_last_child
isfolder       = 'X'
text          = l_wa_node-text
expander     = 'X'
EXCEPTIONS
OTHERS        = 5

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
ENDLOOP.

* spfli-nodes:
LOOP AT it_spfli INTO wa_spfli.
CLEAR l_wa_node.
CONCATENATE wa_spfli-carrid
wa_spfli-connid
INTO l_wa_node-node_key
SEPARATED BY space.
CONCATENATE wa_spfli-carrid
wa_spfli-connid
':'
wa_spfli-cityfrom
' -> '
wa_spfli-cityto
INTO l_wa_node-text
SEPARATED BY space.

```

*Continued on next page*

```

l_wa_node-relatkey = wa_spfli-carrid.
CALL METHOD l_ref_tree_model->add_node
EXPORTING
    node_key          = l_wa_node-node_key
    relative_node_key = l_wa_node-relatkey
    relationship      = cl_simple_tree_model->relat_last_child
    isfolder          = 'X'
    text              = l_wa_node-text
    expander          = 'X'
EXCEPTIONS
    OTHERS            = 5
.

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
ENDLOOP.

* sflight-nodes:
LOOP AT it_sflight INTO wa_sflight.
    CLEAR l_wa_node.
    CONCATENATE wa_sflight-carrid
        wa_sflight-connid
        wa_sflight-fldate
        INTO l_wa_node-node_key
        SEPARATED BY space.
    CONCATENATE wa_sflight-carrid
        wa_sflight-connid
        INTO l_wa_node-relatkey
        SEPARATED BY space.
    WRITE wa_sflight-fldate TO date_text.
    l_wa_node-text = date_text.

CALL METHOD l_ref_tree_model->add_node
EXPORTING
    node_key          = l_wa_node-node_key
    relative_node_key = l_wa_node-relatkey
    relationship      = cl_simple_tree_model->relat_last_child
    isfolder          = space
    text              = l_wa_node-text
    expander          = space
EXCEPTIONS
    OTHERS            = 5
.

IF sy-subrc <> 0.
MESSAGE a012.

```

*Continued on next page*

```
ENDIF.  
ENDLOOP.  
  
ENDFORM.          " ADD_NODES
```



## Lesson Summary

You should now be able to:

- Display data in the different Tree Control variants
- Create hierarchies
- Integrate additional data if necessary
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container

# Lesson: Tree Control: Creating a Hierarchy

## Lesson Overview

This lesson explains how to create a hierarchy and display it on the presentation server using the example of the simple tree



## Lesson Objectives

After completing this lesson, you will be able to:

- Create a hierarchy in a simple tree
- Types of relationships
- Passing node entries

## Business Example

The application to be implemented needs to display data in a Tree Control. You should create hierarchies and integrate additional data if necessary

## Creating a Hierarchy in a Simple Tree

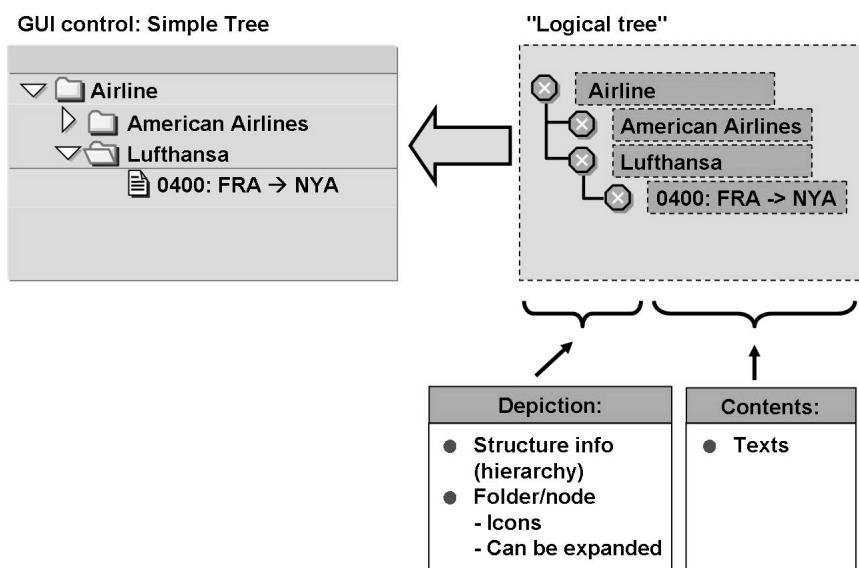


Figure 165: Creating a Hierarchy in a Simple Tree

The following slides explain how you create a hierarchy and display it on the presentation server using the example of the simple tree.

The steps involved are identical for the column and list trees. You only need to process the additional data.

The basis for displaying the tree on the presentation server is a node table managed by the tree model instance and automatically synchronized with the presentation server.

This node table contains the tree in a “logical form” – that is, entries for each node and their relationships with each other.

The entries for the nodes also contain information on which icons should be displayed with them

**Only in the simple tree** do the node entries also contain the texts that are displayed in the hierarchy area left-aligned after the icons.

## Filling the Node Table

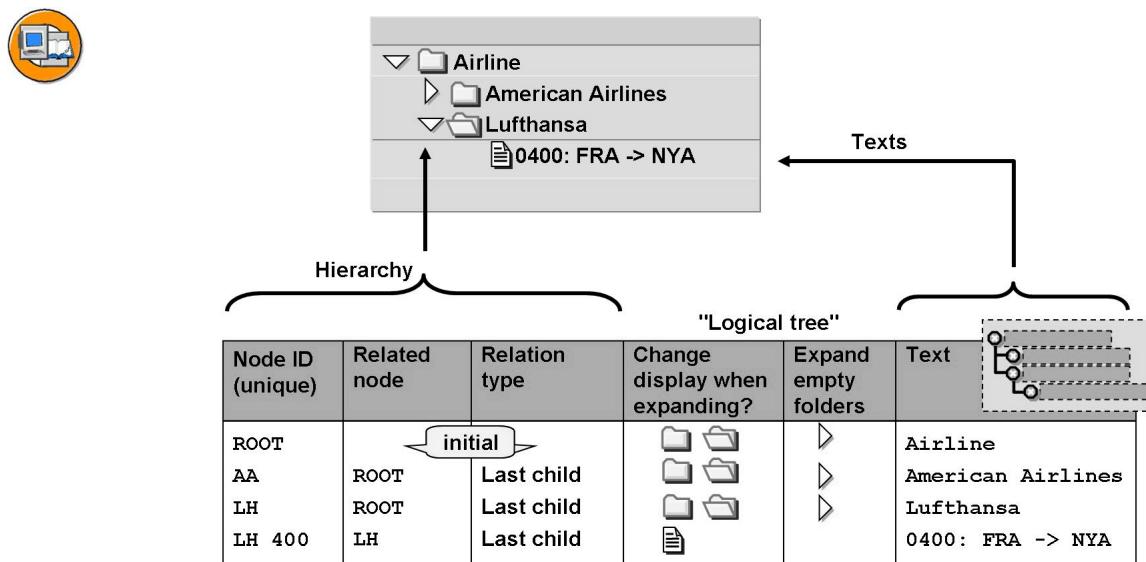


Figure 166: Filling the Node Table

The columns specified above **must** be filled in the node table.

The first three fields specify the position of the node in the hierarchy. You must give every node an **unique** name. Moreover, this node must have a relationship with another node, unless it is a root node. You store the type of this relationship in the third column. **You must assign initial values in the second and third columns to root nodes.**

You also specify whether or not the icons should change if the user opens a node and this node possesses subnodes. If you use **standard** icons, the system will display a closed or open folder as appropriate. If the icon is not to change, a sheet of paper is used as the **standard** icon.



The next column specifies whether or not empty folders should be shown as expandable. It makes sense to do this if this folder will contain subnodes at runtime. In the above example, the airline American Airlines does not (yet) contain any subnodes. However, its node should still be shown as expandable.

You should always supply text entries as well.

## The Line Structure of the Node Table in Detail

TREEMSNDOT		Global structure type for the simple tree model
NODE_KEY	✓	Node ID (unique, any length)
RELATKEY	✓	ID of a node to which the node is related
RELATSHIP	✓	Type of relationship between NODE_KEY and RELATKEY
HIDDEN		Do not display node: Yes/no
DISABLED		Do not make node selectable: Yes/no
ISFOLDER	✓	Node is folder or a leaf
N_IMAGE		Specifies the image used for the node
EXP_IMAGE		Specifies the image used for an open folder
STYLE		Node style
NO_BRANCH		Do not display connection line to node: Yes/no
EXPANDER	✓	Also expand empty folders: Yes/no
DRAGDROPID		Identifier for Drag&Drop behavior table
USEROBJECT		Any object reference
TEXT	✓	Text (any length)

Figure 167: The Line Structure of the Node Table in Detail

The structure type shown above is defined globally in the ABAP Dictionary and is available from SAP R/3 Basis Release 4.6C onwards. In Releases 4.6A and 4.6B you must define your own global structure type based on the global structure type TREEV\_NODE.

Fill the NODE\_KEY field with a string of any length for the technical name. Since the node is identified by this name, it must be **unique** throughout the hierarchy.

Define a hierarchical dependency of a node by entering the relationship (RELATSHIP) it has with other – **already defined** – nodes (RELATKEY).

The string in the TEXT field as displayed left-aligned after the icon as a node text.

Usually, you need only fill these fields. You need the others only in special cases. For further information about these fields, refer to the online documentation.

The NO\_BRANCH field only has an effect if the user is using the “classical GUI display”.

With the **column tree model**, use the global structure type TREEMCNODT. In addition to the fields shown here, it also contains the ITEMSINCOM field (Load entries on request: yes/no).

With the **list tree model**, use the global structure type TREEMLNODT. In addition to the fields shown here, it also contains the ITEMSINCOM and LAST\_HITEM fields (number of the last list entry in the hierarchy area).

## Types of Relationship: Hierarchical Dependency

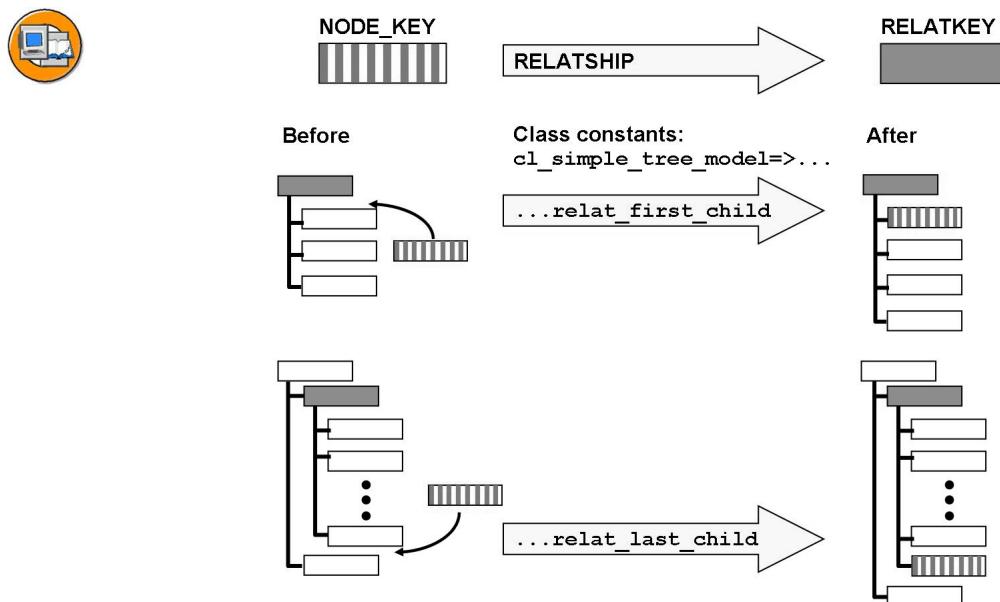


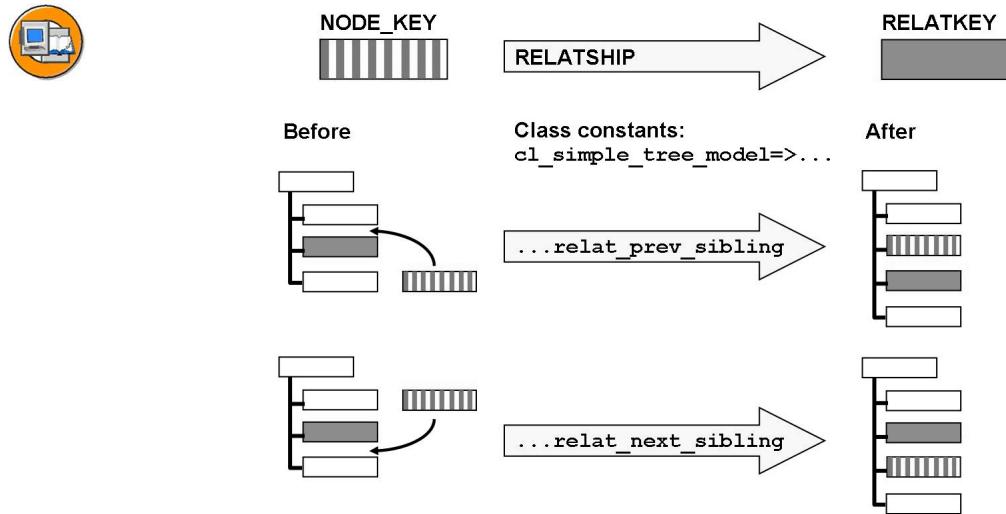
Figure 168: Types of Relationship: Hierarchical Dependency

The following slides show the six possibilities ways available for defining relationships between nodes.

Fill the FIELDNAME field with one of the six class constants.

You can insert a node either as the first subnode of the reference node, or as the last subnode.

## Types of Relationship: Previous/Next Same-Level Nodes

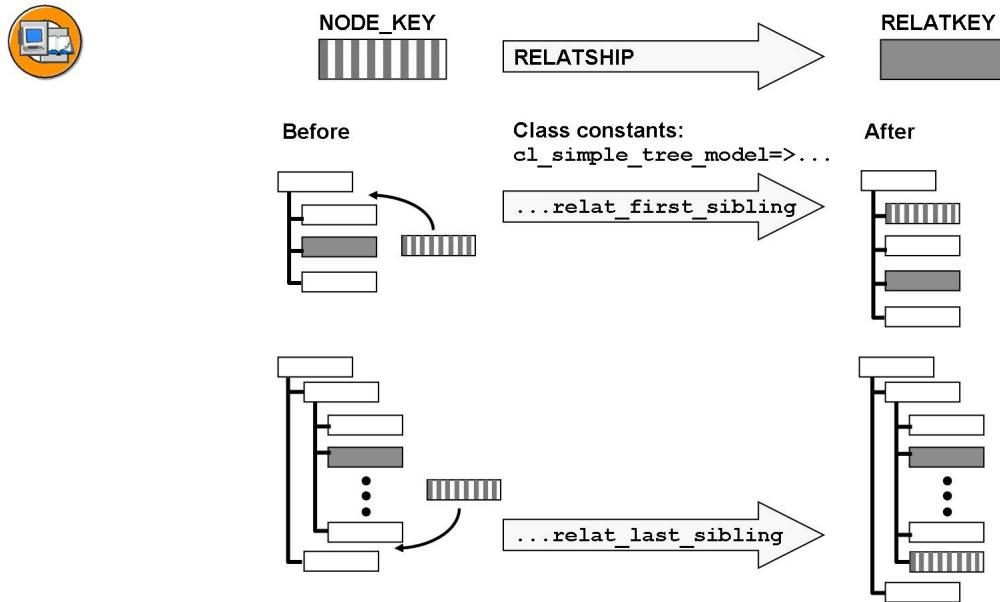


**Figure 169: Types of Relationship: Previous/Next Same-Level Nodes**

You can insert a node directly before or after the reference node at the same hierarchy level.

This is a practical option if you cannot or do not want to use the supernode ID.

## Types of Relationship: First/Last Node in Hierarchy Level



**Figure 170: Types of Relationship: First/Last Node in Hierarchy Level**

You can insert a node as the first or last node in a the hierarchy level of the reference node.

This is a practical option if you cannot or do not want to use the supernode ID.



## Passing Node Entries to the Simple Tree Model Instance

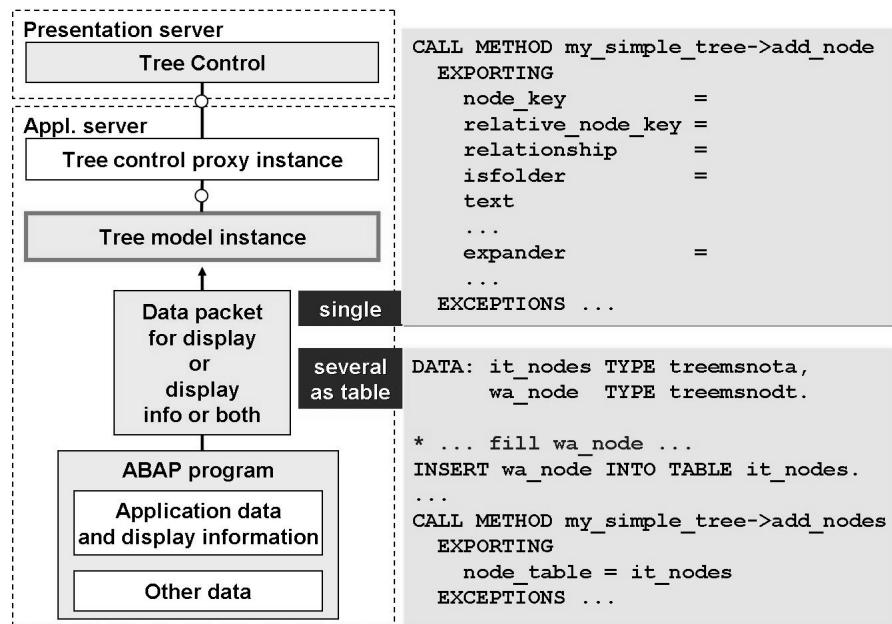


Figure 171: Passing Node Entries to the Simple Tree Model Instance

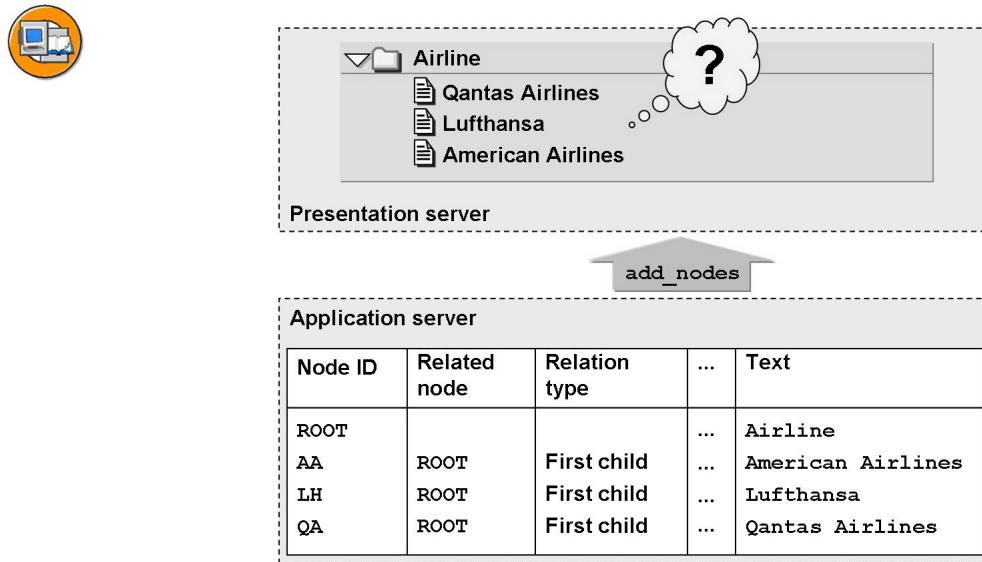
To display a hierarchy with the simple tree model, you must pass the node entries of the tree model instance. This instance first performs a consistency check and then updates the display on the presentation server by communicating with the tree control proxy instance.

You have two options:

- Either call the ADD\_NODE method and fill its parameters as before. (The names and sequence of the parameters do not differ substantially from those shown in the previous structure.) The method call is thus equivalent to inserting an entry in the node table.
- You use the ADD\_NODES method. In this case, you must first define your own node table locally from the global type TREEMSNOTA and fill it with the number of new node entries you want.

In SAP R/3 Basis Releases 4.6A and B you must also pass the name of your global structure type for the node entries (the TABLE\_STRUCTURE\_NAME parameter). You must also define the local node table using this.

## Recapitulation: Passing the Node Table to the Presentation Server



**Figure 172: Recapitulation: Passing the Node Table to the Presentation Server**

The node entries are evaluated on the presentation server in the order in which they are stored in the node table. Generally, this is the order in which they are inserted.

You should note the following points when constructing the hierarchy:

- If you specify a reference node for a node, the latter must be known to the presentation server – that is, the reference node entry must already be in the node table. If you pass the reference node together with the new node to the tree model instance, the reference node must come **before** the new node in the node table.
- The relationships specified in the node table are valid for the state of the hierarchy **while** the current table line is being passed. In the above example, “AA” is not really the “first subnode” of “ROOT” any more, once “LH” has been added. Similarly, “LH” is only the “first subnode” of “ROOT” until “QA” is inserted.

### Summary:

- You should create an algorithm for defining the unique node ID – for example, by combining application keys if available. These are generally unique.
- In hierarchical dependencies, use the relationship type “**last child**”. Then the sequence of entries in the node table matches that on the presentation server.



## Lesson Summary

You should now be able to:

- Create a hierarchy in a simple tree
- Types of relationships
- Passing node entries

# Lesson: Tree Control: Additional Functions of the Tree Model

## Lesson Overview

This lesson explains how to print and search in the Tree Control.



## Lesson Objectives

After completing this lesson, you will be able to:

- Print the Tree Content
- Search in the Tree

## Business Example

The application to be implemented needs to print the Tree Content. Also, the user might want to search in the Tree.

## Printing the Tree Contents



```
CALL METHOD my_simple_tree->print_tree
  EXPORTING
    all_nodes =  Print all nodes: Yes/no
    *       title   =  Heading text (any length)
    preview   =  Print preview: Yes/no
  EXCEPTIONS ...
```

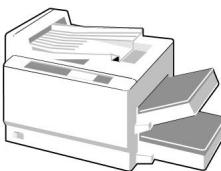


Figure 173: Printing the Tree Contents

The tree model classes provide the PRINT\_TREE method so that users can print the contents of the tree. The method first creates an ABAP print list from the node table. It then sends it to the spool and calls an ABAP list screen.

You can fill the ALL\_NODES parameter as follows: “ ”: Only the nodes displayed on the screen are printed. “X”: All the nodes in the hierarchy are printed.

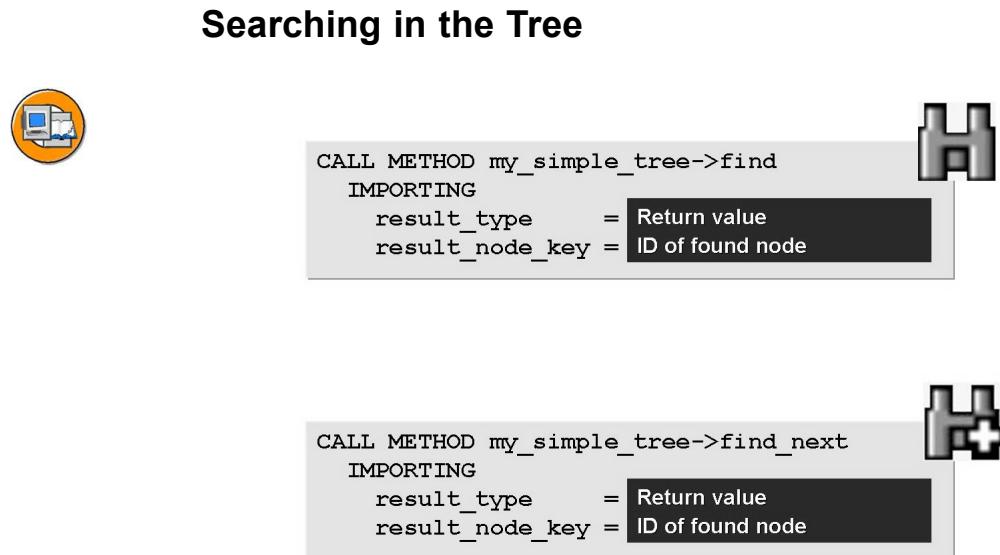


Figure 174: Searching in the Tree

The FIND method first calls a dialog box where the user can enter a string or pattern, for which the program then searches in all the nodes of the tree. The return value indicates whether or not the search was successful. Use the class constant cl\_simple\_tree\_model=>find\_match and so on. If the search string has been found, the tree model instance takes note of the node ID. You should mark this using the SET\_SELECTED\_NODE method.

There are also methods for find **further hits**, such as FIND\_NEXT: This method continues the search begun by FIND, without displaying any more dialog boxes, from the last node containing the search string. If the search string has been found, the tree model instance takes note of the node ID. This information is not deleted until the user calls the FIND method again. In this way, you can fill an internal table incrementally with found nodes and mark them all using the SELECT\_NODES method. Pass the hits table to this method.

You can remove all selections using the UNSELECT\_NODES method.

If you want to implement search functions, you should naturally allow the user to select **several** nodes. Bear this in mind **when creating the instance** of the control.

For further information, refer to the keyword documentation for the CLASS statement.



## Lesson Summary

You should now be able to:

- Print the Tree Content
- Search in the Tree

## Lesson: Tree Control: Events

### Lesson Overview

This lesson explains how to react to events raised by the Tree Control, how the Tree Model instance controls the data flow and how to register handler methods



### Lesson Objectives

After completing this lesson, you will be able to:

- Name some of the events raised by the Tree Control
- Find the application data belonging to the node raising the event
- Create an application toolbar

### Business Example

You need to react to the user action, such as double-click and find the application data belonging to the node that raised the event.

### Double-Clicking Events

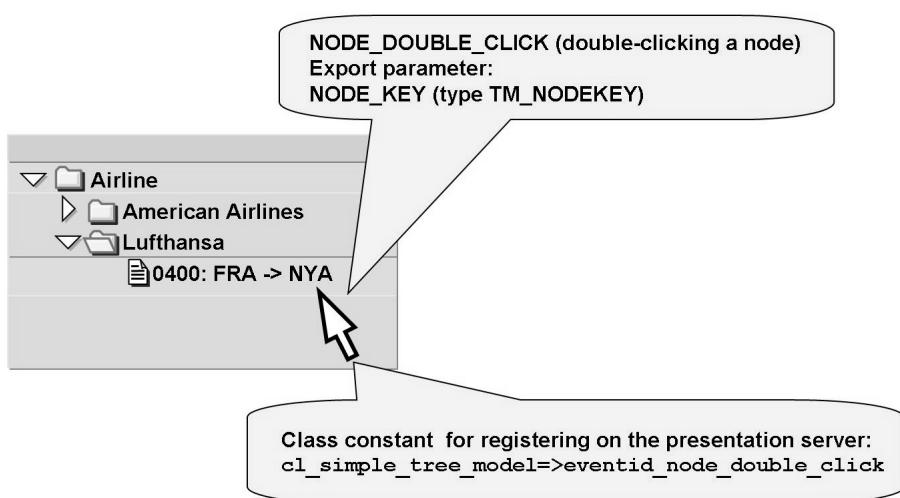


Figure 175: Double-Clicking Events

If the user double-clicks a data area, the **NODE\_DOUBLE\_CLICK** event is raised in the calling program.

To register events on the presentation server use the class constant `cl_simple_tree_model=>eventid_node_double_click`.

The export parameter contains the ID of the appropriate node.

For further information about other mouse operation events, refer to the online documentation.

## Data Administration

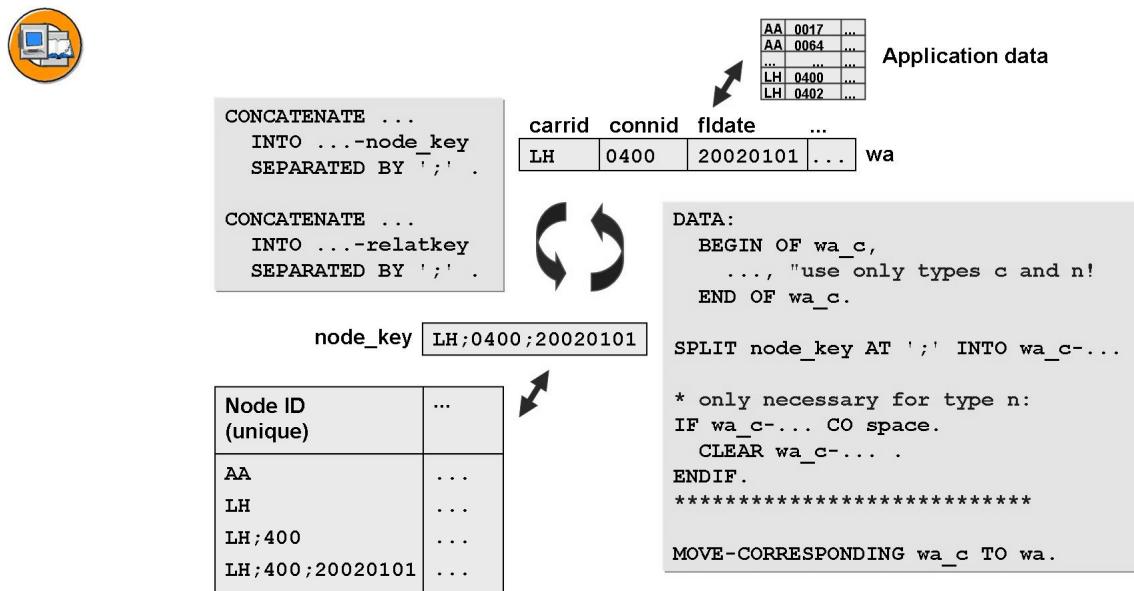


Figure 176: Data Administration

Generally when a node-specific event is raised, the problem is finding the application data that belongs to the node. When forming the unique node IDs, use a separator that **never** occurs in application keys. Here, we used the semicolon (;). Then form the node IDs from the key values and separators using the **CONCATENATE** statement as shown above. You can then extract the application key values from the node ID using the algorithm shown at the top right of the above slide.

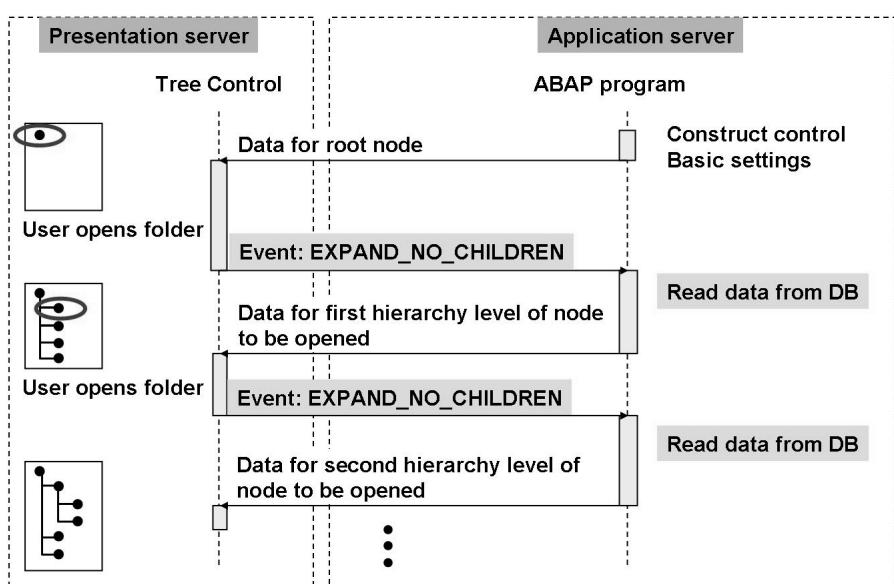
→ **Note:** The **SPLIT** statement requires its target fields to be **character-type**. Spaces are retained. If there are “too few” separators in the node ID – for example, “LH;0400” above - **spaces** are written to the remaining target fields **even if these are of type n**. For this reason, you also need to initialize the target fields in such cases. (The variant of the **SPLIT** statement performed on internal tables takes roughly twice as long to run). For a complete implementation including alternative algorithms, see the SAPBC412\_TRMD\_GET\_APPL\_KEY program in the package BC412.

In SAP R/3 Basis Releases 4.6A and B you must ensure that you assign unique node IDs to the application key in a different way - generally using internal tables. For examples of this, runtime-optimized, see the programs SAPBC412\_TRED\_DATA\_MANAGE\_1, ...\_2, and ...\_2A in the package BC412.



**Caution:** If you use other character-manipulation algorithms, make sure they are **Unicode-enabled**.

## Reading Application Data Only After User Request



**Figure 177: Reading Application Data Only After User Request**

To improve runtime performance, make sure that data in the hierarchy is read from the database server and transported to the presentation server only if the user requests it.

The **EXPAND\_NO\_CHILDREN** event is available to guarantee this. It is only ever raised by the tree control instance if the user opens a folder (fills the node attribute EXPANDER with an “X”), for which **no** subnodes have yet been transported to the presentation server.

Thus when you transport the hierarchy to the presentation server you should start with a “basic set” of nodes. The data for the other nodes should then be read from the database server and transported to the presentation server in the handler method for this event.

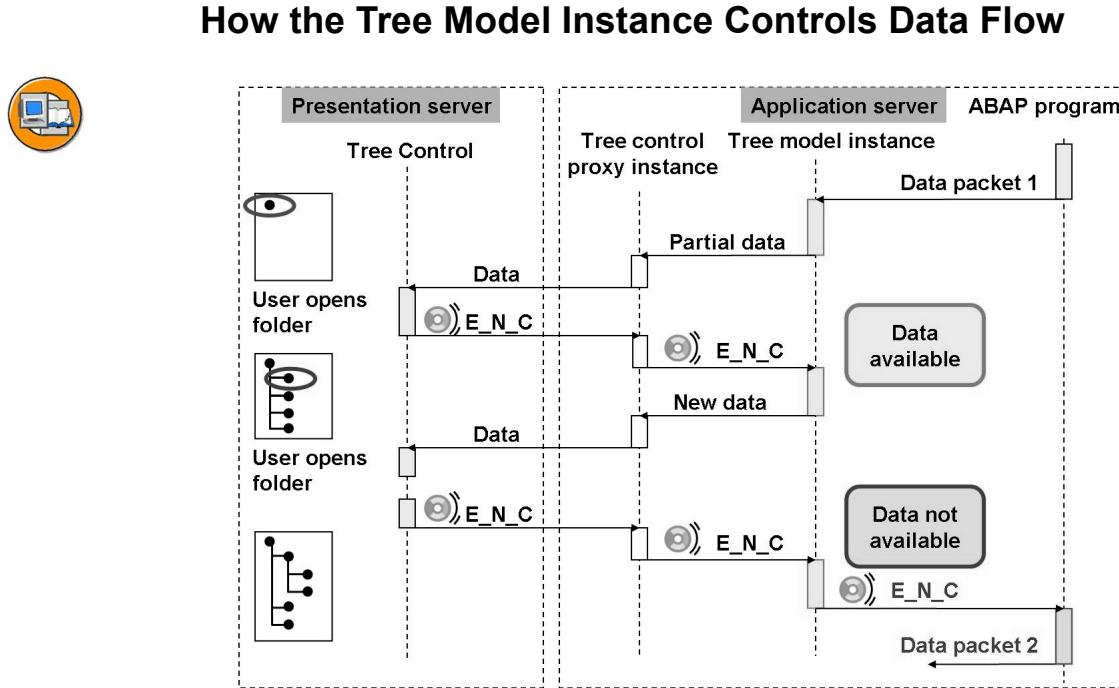


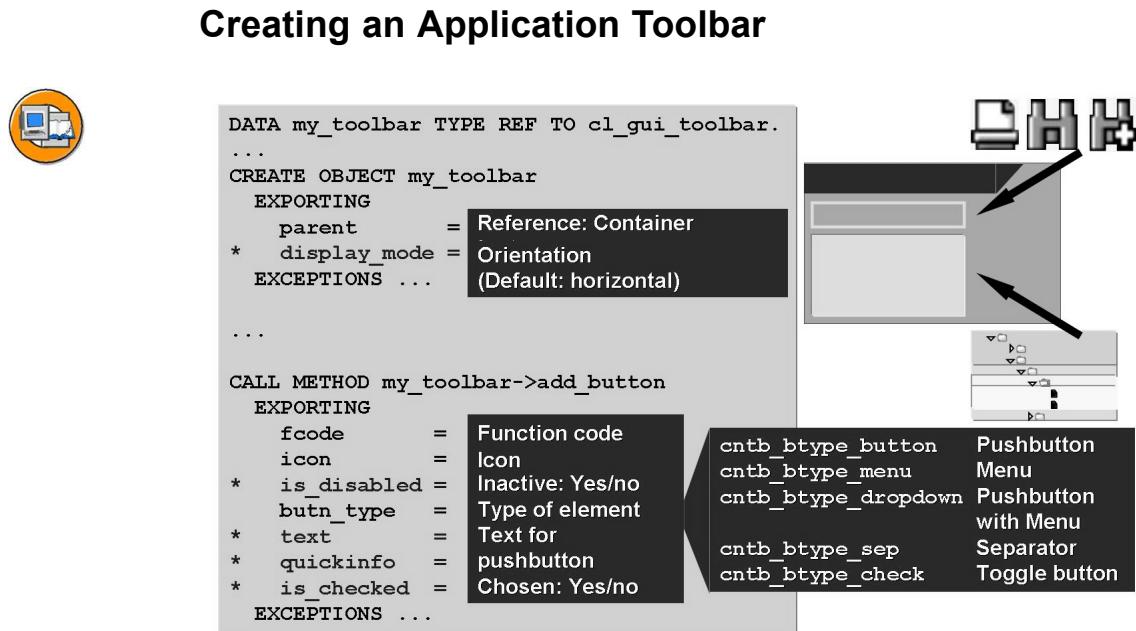
Figure 178: How the Tree Model Instance Controls Data Flow

The requirement outlined in the previous slide is fulfilled to a very great extent **by the tree model classes**. The tree model instance transports only subnodes of the folder opened by the user to the presentation server, regardless of how many node entries you have already transported to the tree model instance. That is, the tree model instance handles the EXPAND\_NO\_CHILDREN event **independently**.

The tree model instance “passes along” an event only if it receives a request from the tree control for subnodes of a folder whose data has not been passed yet to the tree model instance.

You can then pass the missing data in a handler method for this event of the **tree model instance**. Ideally, the “incremental subsequent reading” of the appropriate application data from the database server only follows now.

In SAP R/3 Basis Releases 4.6A and 4.6B you must make sure that the flow depicted above actually occurs. To do this, implement a handler method for the EXPAND\_NO\_CHILDREN event of the **tree control proxy instance**.



**Figure 179: Creating an Application Toolbar**

To add an application toolbar to a tree control, first create the container instance to hold it. Generally you use a vertically-divided splitter container to do this: The upper area contains the application toolbar while the lower contains the actual tree. For ergonomic reasons, adapt the position of the separator to the space needed for the toolbar. Make sure that the user **cannot** move it.

Then create an instance of the class CL\_GUI\_TOOLBAR. Fill the parameter PARENT with the reference to the container instance.

Then add entries for pushbuttons to the instance, by calling the CL\_GUI\_TOOLBAR method for each one.

To specify the type of split, use one of the two class constants cl\_gui\_toolbar=>m\_mode\_vertical or cl\_gui\_toolbar=>m\_mode\_horizontal.

To specify the element type use the constants shown of the type group CNTB.

For further details on the ADD\_BUTTON method, refer to the online documentation.

## Storing a Function



```

CLASS ... DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS on_function_selected FOR EVENT function_selected
      OF cl_gui_toolbar IMPORTING fcode.
  ENDCLASS.
  CLASS ... IMPLEMENTATION.
    METHOD on_function_selected.
      CASE fcode.
        WHEN '....'.
        ...
      ENDCASE.
    ENDMETHOD.
  ENDCLASS.

DATA:                                         Registering Events and Handler Methods
  it_events TYPE cntl_simple_events,
  wa_event  LIKE LINE OF it_events.
...
wa_event-eventid = cl_gui_toolbar->m_id_function_selected.
INSERT wa_event INTO TABLE it_events.
CALL METHOD my_toolbar->set_registered_events
  EXPORTING events = it_events ....
SET HANDLER ...=>on_function_selected FOR my_toolbar.

```

**Figure 180: Storing a Function**

If the user chooses a function element that has been added to the application toolbar, the proxy instance triggers the **FUNCTION\_SELECTED** event. This contains the function code as the **export parameter FCODE**.

To store this additional function, you must implement a handler method for this event. You can find out which function element has been chosen by querying the function code.

To register events on the presentation server use the class constant `cl_gui_toolbar=>m_id_function_selected`.



## Exercise 15: Application Toolbar

### Exercise Objectives

After completing this exercise, you will be able to:

- Assign an application toolbar to the tree control
- Format the contents of the tree model such that they can be printed.
- Search for a string in the tree model (optional).

### Business Example

Extend your program to display the flight data in such a way that:

- You offer the user a pushbutton (optional: three pushbuttons) for the tree control *Print*: (optional: *Find* and *Find next*)
- Users can print the contents of the tree
- Users can search for a string in the tree model (optional).

The program should select the node, once it has found the relevant string:

when users choose *Find next*, all the nodes found until that point should be shown as selected

when users choose *Find* for a second time, the hit list should be cleared and constructed from scratch.

**Program:** ZBC412\_##\_TRM\_EX2

**Template:** SAPBC412\_TRMT\_EXERCISE\_2

**Model solution:** SAPBC412\_TRMS\_EXERCISE\_2

where ## is the group number.

### Task 1:

Copy the template.

1. Copy the template SAPBC412\_TRMT\_EXERCISE\_2, assigning it the new name ZBC412\_##\_TRM\_EX2 and get to know how it works.

*Continued on next page*

## Task 2:

Create and configure a toolbar control instance.

Note, that in the template a splitter container is used to divide the container area of the docking control horizontally into two areas. Use the lower cell (`ref_cell_bott`) as a container for the tree control and the upper cell (`ref_cell_top`) as a container for the toolbar control.

1. Create a reference variable for the picture control instance (We suggest you use the name `ref_toolbar`).
2. Extend the module `init_tree_processing_0100` as follows:  
Create a toolbar instance and link it to the upper cell of the splitter container.  
Define a subroutine that configures the toolbar control instance (suggested name: `configure_toolbar`) and call it here.

Add the following three buttons using the `add_button` method. (To use the icon names, you also need to load the type `groupicon`).

### Pushbutton 1

<code>fcode</code>	any (we suggest 'PRINT')
<code>icon</code>	<code>icon_print</code>
<code>butn_type</code>	<code>cntb_btype_button</code>
<code>quickinfo</code>	any (we suggest <i>Print tree contents</i> )

### Pushbutton 2 (Optional)

<code>fcode</code>	any (we suggest 'SEARCH')
<code>icon</code>	<code>icon_search</code>
<code>butn_type</code>	<code>cntb_btype_button</code>
<code>quickinfo</code>	any (we suggest <i>Find</i> )

### Pushbutton 3 (Optional)

<code>fcode</code>	any (we suggest 'SEARCHNEXT')
<code>icon</code>	<code>icon_search_next</code>
<code>butn_type</code>	<code>cntb_btype_button</code>
<code>quickinfo</code>	any (we suggest <i>Find next</i> )

3. Activate and test your program.

*Continued on next page*

### Task 3:

Define a static method to handle the `function_selected` event of the toolbar instance as follows:

1. Find out about the interface of the event.
2. Define a local class (suggested name: `lcl_event_handler`) that contains the event handler method (suggested name: `on_function_selected`) for the event.
3. Register the toolbar control event with the automation handler. Then register the static event handler method for the toolbar instance.

### Task 4:

To implement the handler method `on_function_selected`, use the simple tree model functions for printing (and optionally searching) as follows:

1. **PRINT function:** Use the method `print_tree` of your tree model instance to print the contents of the tree. Read the online documentation to learn more about the method interface. Use the preview mode for printing. Print **only** the nodes that the user can see on the screen.

If errors occur, terminate processing of the method and display the message 012. Activate and test your program (the function).

2. **SEARCH function (optional):**

If the user chooses *Find next*, the program may have to select more than one node. Change the actual parameters of the constructor interface of the tree model instance, so that this is possible.

Use the `unselect_all` method of your tree model instance to remove existing selections applied to any node in the tree.

Use the method `find` of your tree model instance to search the contents of the tree. Read the online documentation to learn more about the scope and interface of the method.

Define a local auxiliary variable (suggested name `l_result_type`) for the search result value.

If the program finds the search string, it should display the relevant node as selected. To do this, buffer its ID in a **local program** hit table (suggested name: `it_hit_nodes`). When the user chooses *Find*, the hits from previous searches are no longer relevant. For this reason, clear the hits table first.

If none of the node texts matches the search string, inform the user of this by displaying message 070.

*Continued on next page*

3. SEARCHNEXT function (**optional**):

Use the method `find_next` of your tree model instance to search the tree for the next node containing the search string. Read the online documentation to learn more about the scope and interface of the method. If the program finds the search string, it should add the ID of the relevant node to the hits table `it_hit_nodes`.

If none of the other node texts matches the search string, inform the user of this by displaying message 071.

4. SEARCH and SEARCHNEXT functions:

Now use the `select_nodes` method to highlight all the hits buffered until now on the screen.



**Hint:** Use the class constants `find_match`, `find_no_match`, `find_expander_node_hit` and `find_canceled` of the `cl_simple_tree_model` class, to handle return values for the methods `find` and `find_next`.

## Solution 15: Application Toolbar

### Task 1:

Copy the template.

1. Copy the template SAPBC412\_TRMT\_EXERCISE\_2, assigning it the new name ZBC412\_##\_TRM\_EX2 and get to know how it works.
  - a) -

### Task 2:

Create and configure a toolbar control instance.

Note, that in the template a splitter container is used to divide the container area of the docking control horizontally into two areas. Use the lower cell (`ref_cell_bott`) as a container for the tree control and the upper cell (`ref_cell_top`) as a container for the toolbar control.

1. Create a reference variable for the picture control instance (We suggest you use the name `ref_toolbar`).
  - a)

```
DATA:  
...  
    ref_toolbar    TYPE REF TO cl_gui_toolbar.
```

2. Extend the module `init_tree_processing_0100` as follows:  
Create a toolbar instance and link it to the upper cell of the splitter container.  
Define a subroutine that configures the toolbar control instance (suggested name: `configure_toolbar`) and call it here.  
Add the following three buttons using the `add_button` method. (To use the icon names, you also need to load the type `groupicon`).

#### Pushbutton 1

<code>fcode</code>	any (we suggest 'PRINT')
<code>icon</code>	<code>icon_print</code>
<code>butn_type</code>	<code>cntb_btype_button</code>
<code>quickinfo</code>	any (we suggest <i>Print tree contents</i> )

*Continued on next page*

**Pushbutton 2 (Optional)**

fcode	any (we suggest 'SEARCH')
icon	icon_search
butn_type	cntb_btype_button
quickinfo	any (we suggest <i>Find</i> )

**Pushbutton 3 (Optional)**

fcode	any (we suggest 'SEARCHNEXT')
icon	icon_search_next
butn_type	cntb_btype_button
quickinfo	any (we suggest <i>Find next</i> )

a)

```

MODULE init_tree_processing_0100 OUTPUT.
  IF ref_tree_model IS INITIAL.

    CREATE OBJECT ref_toolbar
      EXPORTING
        parent = ref_cell_top
      EXCEPTIONS
        others          = 4.
    IF sy-subrc <> 0.
      MESSAGE a050.
    ENDIF.
    PERFORM configure_toolbar CHANGING ref_toolbar.
    ...
  FORM configure_toolbar
    CHANGING p_ref_toolbar TYPE REF TO cl_gui_toolbar.

  * print:
    CALL METHOD p_ref_toolbar->add_button
      EXPORTING
        fcode           = 'PRINT'
        icon            = icon_print
        butn_type       = cntb_btype_button
        quickinfo      = text-pri
      EXCEPTIONS
        OTHERS          = 4.
    IF sy-subrc <> 0.
      MESSAGE a012.
    ENDIF.
  ENDIF.
ENDMODULE.

```

*Continued on next page*

```

ENDIF.

* search:
CALL METHOD p_ref_toolbar->add_button
EXPORTING
  fcode      = 'SEARCH'
  icon       = icon_search
  butn_type  = cntb_btype_button
  quickinfo   = text-src
EXCEPTIONS
  OTHERS      = 4.
IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

* search next:
CALL METHOD p_ref_toolbar->add_button
EXPORTING
  fcode      = 'SEARCHNEXT'
  icon       = icon_search_next
  butn_type  = cntb_btype_button
  quickinfo   = text-srn
EXCEPTIONS
  OTHERS      = 4.
IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.
ENDFORM.

```

3. Activate and test your program.

a) —

### Task 3:

Define a static method to handle the `function_selected` event of the toolbar instance as follows:

1. Find out about the interface of the event.

a) —

*Continued on next page*

2. Define a local class (suggested name: `lcl_event_handler`) that contains the event handler method (suggested name: `on_function_selected`) for the event.

a)

```
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS on_function_selected FOR EVENT function_selected
      OF cl_gui_toolbar
      IMPORTING fcode.
  ENDCLASS.
```

3. Register the toolbar control event with the automation handler. Then register the static event handler method for the toolbar instance.

a)

```
MODULE init_tree_processing_0100 OUTPUT.
  IF ref_tree_model IS INITIAL.
  ...
    wa_event-eventid = cl_gui_toolbar->m_id_function_selected.
    INSERT wa_event INTO TABLE it_events.
    CALL METHOD ref_toolbar->set_registered_events
      EXPORTING
        events           = it_events
      EXCEPTIONS
        OTHERS          = 4
      .
    IF sy-subrc <> 0.
      MESSAGE a042.
    ENDIF.

    SET HANDLER lcl_event_handler=>on_function_selected
      FOR ref_toolbar.
```

## Task 4:

To implement the handler method `on_function_selected`, use the simple tree model functions for printing (and optionally searching) as follows:

1. PRINT function: Use the method `print_tree` of your tree model instance to print the contents of the tree. Read the online documentation to learn more about the method interface. Use the preview mode for printing. Print **only** the nodes that the user can see on the screen.

*Continued on next page*

If errors occur, terminate processing of the method and display the message 012. Activate and test your program (the function).

a)

```

CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_function_selected.
DATA:
  l_titletext TYPE string,
  l_search_result_type      TYPE i,
  l_search_result_node_key  TYPE string.
CASE fcode.
  WHEN 'PRINT'.
    l_titletext = text-tit.
    CALL METHOD ref_tree_model->print_tree
      EXPORTING
        all_nodes      = space "displayed nodes only"
        title         = l_titletext
        preview       = 'X'   "preview mode
      EXCEPTIONS
        OTHERS        = 1.
      IF sy-subrc <> 0.
        MESSAGE a012.
      ENDIF.
...

```

## 2. SEARCH function (**optional**):

If the user chooses *Find next*, the program may have to select more than one node. Change the actual parameters of the constructor interface of the tree model instance, so that this is possible.

Use the `unselect_all` method of your tree model instance to remove existing selections applied to any node in the tree.

Use the method `find` of your tree model instance to search the contents of the tree. Read the online documentation to learn more about the scope and interface of the method.

Define a local auxiliary variable (suggested name `l_result_type`) for the search result value.

If the program finds the search string, it should display the relevant node as selected. To do this, buffer its ID in a **local program** hit table (suggested name: `it_hit_nodes`). When the user chooses *Find*, the hits from previous searches are no longer relevant. For this reason, clear the hits table first.

*Continued on next page*

If none of the node texts matches the search string, inform the user of this by displaying message 070.

- a) –
- 3. **SEARCHNEXT function (optional):**  
Use the method `find_next` of your tree model instance to search the tree for the next node containing the search string. Read the online documentation to learn more about the scope and interface of the method. If the program finds the search string, it should add the ID of the relevant node to the hits table `it_hit_nodes`.
- If none of the other node texts matches the search string, inform the user of this by displaying message 071.
- a) –
- 4. **SEARCH and SEARCHNEXT functions:**

Now use the `select_nodes` method to highlight all the hits buffered until now on the screen.



**Hint:** Use the class constants `find_match`, `find_no_match`, `find_expander_node_hit` and `find_canceled` of the `cl_simple_tree_model` class, to handle return values for the methods `find` and `find_next`.

- a)

```
CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_function_selected.
DATA:
  l_titletext TYPE string,
  l_search_result_type      TYPE i,
  l_search_result_node_key TYPE string.
CASE fcode.
  WHEN 'PRINT'.
  ...
  WHEN 'SEARCH' OR 'SEARCHNEXT'.
    IF fcode = 'SEARCH'.
      CALL METHOD ref_tree_model->unselect_all.
      *       user dialog for search pattern, then find first occurrence:
      CALL METHOD ref_tree_model->find
        IMPORTING
          result_type      = l_search_result_type
          result_node_key = l_search_result_node_key.
    CASE l_search_result_type.
```

*Continued on next page*

```

        WHEN cl_simple_tree_model=>find_match.
*
*          store hit in order to be able to continue later ...
*          CLEAR it_hit_nodes.
*          INSERT l_search_result_node_key INTO TABLE it_hit_nodes.
*          ... and select it (coding downwards!)
WHEN cl_simple_tree_model=>find_no_match.
MESSAGE s070.
EXIT.
WHEN cl_simple_tree_model=>find_canceled.
*
*          search dialog canceled: do nothing
*          EXIT.
ENDCASE.
ELSE.           " fcode = 'SEARCHNEXT':
*
*          find next occurrence,
*          doesn't matter which node has been selected:
CALL METHOD ref_tree_model->find_next
IMPORTING
        result_type      = l_search_result_type
        result_node_key = l_search_result_node_key.
CASE l_search_result_type.
WHEN cl_simple_tree_model=>find_match.
*
*          store hit in order to select it with the others:
*          INSERT l_search_result_node_key INTO TABLE it_hit_nodes.
WHEN cl_simple_tree_model=>find_no_match.
MESSAGE s071(bc412).
EXIT.
WHEN cl_simple_tree_model=>find_canceled.
*
*          search dialog canceled: do nothing
*          EXIT.
ENDCASE.
ENDIF.
*
*          select all hits now:
CALL METHOD ref_tree_model->select_nodes
EXPORTING
        node_key_table      = it_hit_nodes
EXCEPTIONS
        OTHERS              = 1.
IF sy-subrc <> 0.
MESSAGE a012(bc412).
ENDIF.
ENDCASE.
ENDMETHOD.

```

*Continued on next page*

## Result

The complete source code for the model solution is shown below.

### ABAP program

#### *Data Declarations*

```
REPORT  sapbc412_trms_exercise_2 MESSAGE-ID bc412.

TYPE-POOLS icon.

DATA:
* screen-specific:
.

* container:
ref_cont_left    TYPE REF TO cl_gui_docking_container,
ref_cont_split   TYPE REF TO cl_gui_splitter_container,
ref_cell_top     TYPE REF TO cl_gui_container,
ref_cell_bott   LIKE ref_cell_top,

* content:
ref_tree_model   TYPE REF TO cl_simple_tree_model,
ref_toolbar      TYPE REF TO cl_gui_toolbar,

* auxiliary:
it_hit_nodes    TYPE treemnotab,

* event registration:
it_events        TYPE cntl_simple_events,
wa_event         LIKE LINE OF it_events.
```

#### *Local Classes*

```
*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
CLASS lcl_event_handler DEFINITION.
PUBLIC SECTION.
  CLASS-METHODS on_function_selected FOR EVENT function_selected
              OF cl_gui_toolbar
              IMPORTING fcode.
ENDCLASS.
```

*Continued on next page*

```

*-----*
*      CLASS lcl_event_handler IMPLEMENTATION
*-----*
CLASS lcl_event_handler IMPLEMENTATION.
METHOD on_function_selected.

DATA:
    l_titletext TYPE string,
    l_search_result_type      TYPE i,
    l_search_result_node_key  TYPE string.

CASE fcode.
WHEN 'PRINT'.

    l_titletext = text-tit.
    CALL METHOD ref_tree_model->print_tree
        EXPORTING
            all_nodes          = space "displayed nodes only"
            title              = l_titletext
            preview             = 'X'    "preview mode"
        EXCEPTIONS
            OTHERS              = 1.
    IF sy-subrc <> 0.
        MESSAGE a012.
    ENDIF.

WHEN 'SEARCH' OR 'SEARCHNEXT'.

    IF fcode = 'SEARCH'.

        CALL METHOD ref_tree_model->unselect_all.

        *      user dialog for search pattern, then find first occurrence:
        CALL METHOD ref_tree_model->find
            IMPORTING
                result_type      = l_search_result_type
                result_node_key = l_search_result_node_key.

        CASE l_search_result_type.
        WHEN cl_simple_tree_model=>find_match.
        *      store hit in order to be able to continue later ...
            CLEAR it_hit_nodes.
            INSERT l_search_result_node_key INTO TABLE it_hit_nodes.
        *      ... and select it (coding downwards!)
    ENDIF.

```

*Continued on next page*

```

WHEN cl_simple_tree_model=>find_no_match.
MESSAGE s070.
EXIT.

WHEN cl_simple_tree_model=>find_canceled.
*
  search dialog canceled: do nothing
  EXIT.
ENDCASE.

ELSE.          " fcode = 'SEARCHNEXT':
*
  find next occurrence,
*
  doesn't matter which node has been selected:
CALL METHOD ref_tree_model->find_next
  IMPORTING
    result_type      = l_search_result_type
    result_node_key = l_search_result_node_key.

CASE l_search_result_type.
WHEN cl_simple_tree_model=>find_match.
*
  store hit in order to select it with the others:
  INSERT l_search_result_node_key INTO TABLE it_hit_nodes.

WHEN cl_simple_tree_model=>find_no_match.
MESSAGE s071(bc412).
EXIT.

WHEN cl_simple_tree_model=>find_canceled.
*
  search dialog canceled: do nothing
  EXIT.
ENDCASE.
ENDIF.

*
  select all hits now:
CALL METHOD ref_tree_model->select_nodes
  EXPORTING
    node_key_table      = it_hit_nodes
  EXCEPTIONS
    OTHERS              = 1.
  IF sy-subrc <> 0.
    MESSAGE a012(bc412).
  ENDIF.

ENDCASE.
ENDMETHOD.
ENDCLASS.

```

### ***Modules***

*Continued on next page*

```

*&-----*
*&      Module  INIT_CONTAINER_PROCESSING_0100  OUTPUT
*&-----*
*      Start control handling.
*      create container objects
*-----*
MODULE init_container_processing_0100 OUTPUT.
IF ref_cont_left IS INITIAL.

      CREATE OBJECT ref_cont_left
      EXPORTING
      ...

      CREATE OBJECT ref_cont_split
      EXPORTING
      parent          = ref_cont_left
      rows            = 2
      columns         = 1
      EXCEPTIONS
      others          = 1.
      IF sy-subrc <> 0.
      MESSAGE a010.
      ENDIF.

      * configure splitter container ...
      * Easy Splitter Container unfortunately
      * not able to be configurable enough:
      * user should not be able to move border bar!
      CALL METHOD ref_cont_split->set_border
      EXPORTING
      border          = cl_gui_cfw=>false
      EXCEPTIONS
      OTHERS          = 1.
      IF sy-subrc <> 0.
      MESSAGE a012.
      ENDIF.

      CALL METHOD ref_cont_split->set_row_mode
      EXPORTING
      mode            = cl_gui_splitter_container=>mode_absolute
*      IMPORTING
*      RESULT          =
      EXCEPTIONS
      OTHERS          = 1.

```

*Continued on next page*

```

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
CALL METHOD ref_cont_split->set_row_height
EXPORTING
  id          = 1
  height      = 20
*   IMPORTING
*   RESULT      =
EXCEPTIONS
  OTHERS       = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

CALL METHOD ref_cont_split->set_row_sash
EXPORTING
  id          = 1
  type        = cl_gui_splitter_container=>type_movable
  value       = cl_gui_splitter_container=>false
*   IMPORTING
*   RESULT      =
EXCEPTIONS
  OTHERS       = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
*****  

ref_cell_top  =
  ref_cont_split->get_container( row = 1 column = 1 ).
ref_cell_bott =
  ref_cont_split->get_container( row = 2 column = 1 ).

ENDIF.
ENDMODULE.           " init_container_processing_0100  OUTPUT

*&-----*
*&     Module  init_tree_processing_0100  OUTPUT
*&-----*
*     create tree object, link to container and fill with data
*&-----*
MODULE init_tree_processing_0100 OUTPUT.
IF ref_tree_model IS INITIAL.

```

*Continued on next page*

```

CREATE OBJECT ref_toolbar
EXPORTING
  parent = ref_cell_top
EXCEPTIONS
  others          = 4.
IF sy-subrc <> 0.
  MESSAGE a050.
ENDIF.

PERFORM configure_toolbar CHANGING ref_toolbar.

wa_event-eventid = cl_gui_toolbar->m_id_function_selected.
INSERT wa_event INTO TABLE it_events.
CALL METHOD ref_toolbar->set_registered_events
EXPORTING
  events           = it_events
EXCEPTIONS
  OTHERS          = 4

IF sy-subrc <> 0.
  MESSAGE a042.
ENDIF.

SET HANDLER lcl_event_handler->on_function_selected
FOR ref_toolbar.

* more than one node must be selectable in order to enable
* search functionalities:
CREATE OBJECT ref_tree_model
EXPORTING
node_selection_mode = cl_simple_tree_model->node_sel_mode_multiple
EXCEPTIONS
  others          = 1.
IF sy-subrc <> 0.
  MESSAGE a043.
ENDIF.

CALL METHOD ref_tree_model->create_tree_control
EXPORTING
  parent = ref_cell_bott
EXCEPTIONS
  OTHERS = 1.
IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

```

*Continued on next page*

```

        PERFORM add_nodes USING ref_tree_model.

        ENDIF.

ENDMODULE.          " init_tree_processing_0100  OUTPUT

```

### *Subroutines*

```

*-----*
*&      Form  FREE_CONTROL_RESSOURCES
*-----*
*      free control ressources on the presentation server
*      free all reference variables (ABAP object) -> garbage collector
*-----*
*      no interface
*-----*
FORM free_control_ressources.
CALL METHOD ref_cont_left->free.

FREE:
  ref_toolbar,
  ref_tree_model,
  ref_cont_split,
  ref_cont_left.

ENDFORM.           " FREE_CONTROL_RESSOURCES
*-----*
*      FORM configure_toolbar
*-----*
*  --> P_REF_TOOLBAR
*-----*
FORM configure_toolbar
  CHANGING p_ref_toolbar TYPE REF TO cl_gui_toolbar.

* print:
  CALL METHOD p_ref_toolbar->add_button
    EXPORTING
      fcode          = 'PRINT'
      icon           = icon_print
      butn_type      = cntb_btype_button
      quickinfo     = text-pri
    EXCEPTIONS
      OTHERS         = 4.

```

*Continued on next page*

```
IF sy-subrc <> 0.  
MESSAGE a012.  
ENDIF.  
  
* search:  
CALL METHOD p_ref_toolbar->add_button  
    EXPORTING  
        fcode          = 'SEARCH'  
        icon           = icon_search  
        butn_type     = cntb_btype_button  
        quickinfo     = text-src  
    EXCEPTIONS  
        OTHERS         = 4.  
IF sy-subrc <> 0.  
MESSAGE a012.  
ENDIF.  
  
* search next:  
CALL METHOD p_ref_toolbar->add_button  
    EXPORTING  
        fcode          = 'SEARCHNEXT'  
        icon           = icon_search_next  
        butn_type     = cntb_btype_button  
        quickinfo     = text-srn  
    EXCEPTIONS  
        OTHERS         = 4.  
IF sy-subrc <> 0.  
MESSAGE a012.  
ENDIF.  
  
ENDFORM.
```



## Lesson Summary

You should now be able to:

- Name some of the events raised by the Tree Control
- Find the application data belonging to the node raising the event
- Create an application toolbar

# Lesson: Tree Control: Additional Data in the Column and List Trees

## Lesson Overview

This lesson explains how to use the additional functions of the Tree Model Classes.



## Lesson Objectives

After completing this lesson, you will be able to:

- Use the additional functions of the Tree Model Classes

## Business Example

You would like to display additional data in the Column or List Tree. Therefore you want to know about the column entries in the Column Tree Model and the list entries in the List Tree Model.

## Displaying Additional Data in a Column Tree



GUI Control: Column Tree	
Flight info	Details
▼  Airline	
►  American Airlines	20 connections
▼  Lufthansa	15 connections
0400: FRA → NYA	Frankfurt → New York

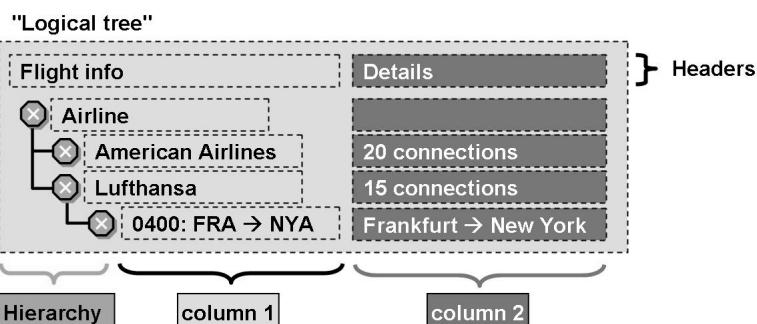


Figure 181: Displaying Additional Data in a Column Tree

In the column tree the node entries contain only the icon and the simple hierarchy reference.

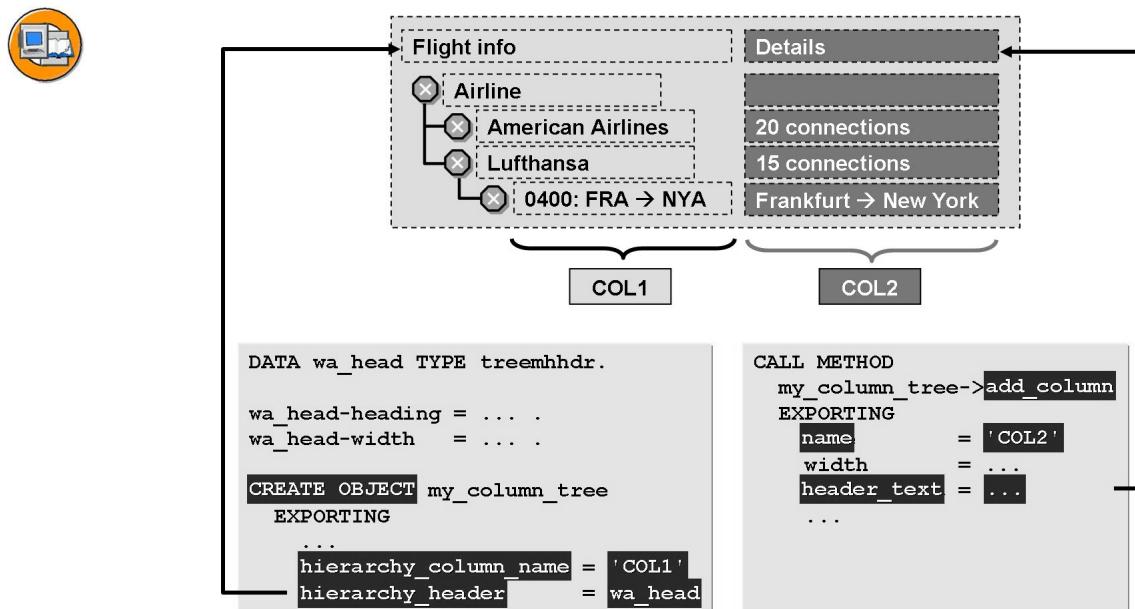
Technically speaking, the texts in the hierarchy area form the **entries** of the first **column** although they are displayed left-aligned after the node icon. You must also transport the column entries to the node entries.

You can also define headers.

The following slides extend the procedure familiar from the simple tree model by including the elements needed for column entries.

Essentially, you can also apply this procedure in SAP R/3 Basis Releases 4.6A and 4.6B.

## Columns in the Column Tree



**Figure 182: Columns in the Column Tree**

Before the column entries can be transported, you need to have created the columns for the column tree control.

The system creates the column in the hierarchy area when it instantiates the tree control model instance. In the constructor interface, give the column a technical name ("COL1" in the example above). Fill the HIERARCHY\_HEADER parameter with a structure of the global type TREEMHHDR whose fields you use to specify other column attributes. (Heading texts, column width, tool tip text, and so on)

You can also define other columns using the ADD\_COLUMN method. Fill the interface parameters NAME, WIDTH, and HEADER\_TEXT with the technical name, width, and column heading.

## The Node Table in the Column Tree Model

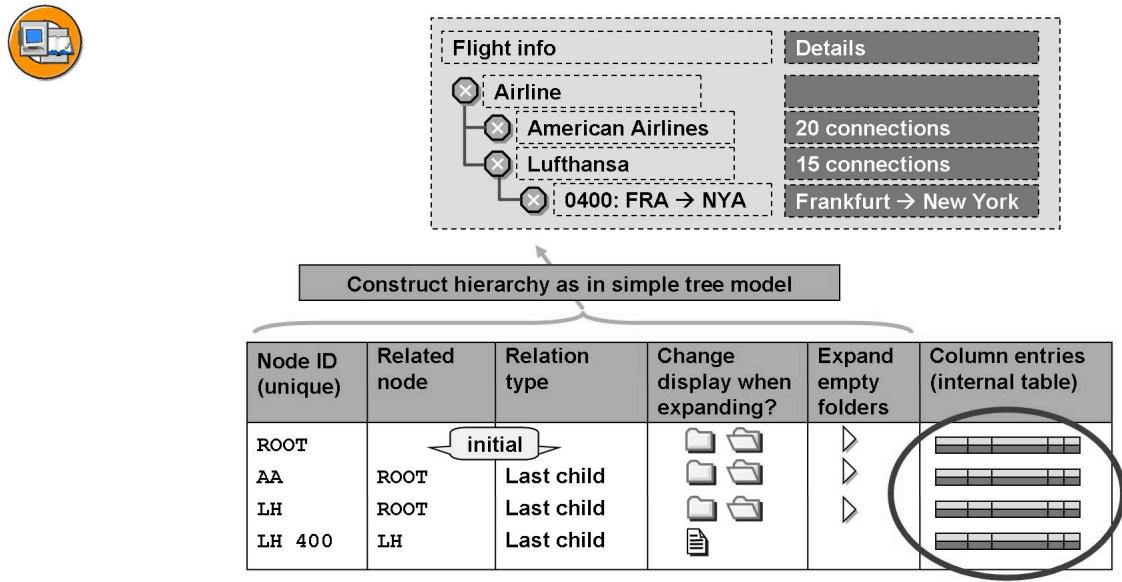


Figure 183: The Node Table in the Column Tree Model

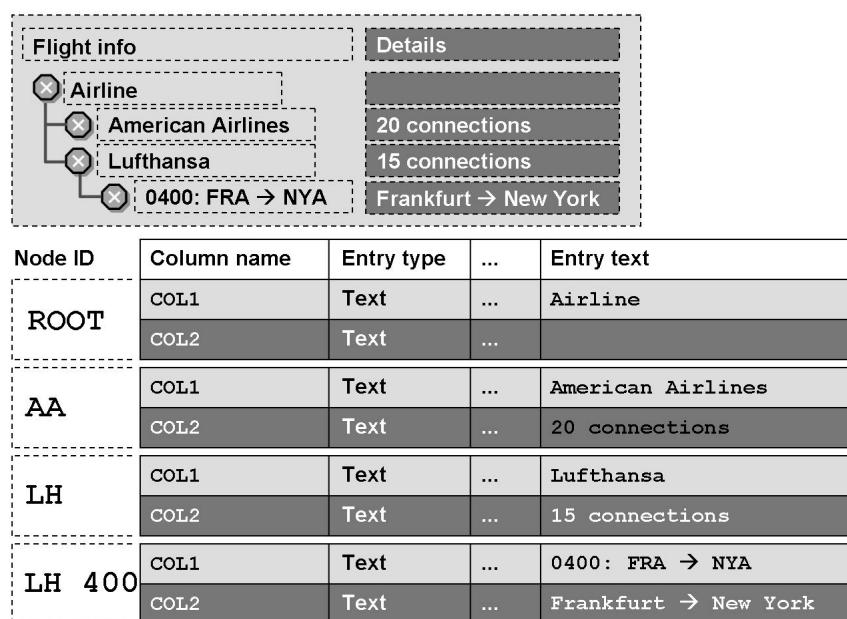
The columns specified above **must** be filled in the node table.

The first three fields specify the position of the node in the hierarchy. The principles of the simple tree model apply here too.

In general, each node entry has several column entries.



## Column Entries in the Column Tree Model



**Figure 184: Column Entries in the Column Tree Model**

An internal table is passed for the column entries of each node. These contain a line for each defined column.

The lines contain the type of entry that **this column** should contain – generally text entries. Therefore the text must also be stored in the line.

Thus each column entry is generally uniquely identified using a value pair (node ID, column name), similar to a two-dimensional coordinates system.

## The Line Structure of the Column Entries in Detail



TREEMCITEM	Global structure type for the column tree model
ITEM_NAME ✓	Column name
CLASS ✓	Entry type: Text, checkbox, pushbutton, link
FONT	Font: proportional, monospace or standard
DISABLED	Do not make entry selectable: Yes/no
EDITABLE	Entry input-ready: Yes/no
HIDDEN	Do not display entry: Yes/no
T_IMAGE	Icon name
CHOSEN	Entry chosen: Yes/no
STYLE	Node style
TXTISQINFO	Entry text as tool tip: Yes/no
TEXT ✓	Text (any length)

**Figure 185: The Line Structure of the Column Entries in Detail**

The structure type shown above is defined globally in the ABAP Dictionary and is available from SAP R/3 Basis Release 4.6C onwards. In Releases 4.6A and 4.6B you must define your own global structure type based on the global structure type TREEV\_ITEM.

In the procedure described here the column entries are passed along with the node entry to the tree model instance. Thus, you do not need a field for the node ID.

The entry is assigned to exactly one column using the ITEM\_NAME field.

You can use the following class constants for the field CLASS:

- cl\_column\_tree\_model=>item\_class\_text
- cl\_column\_tree\_model=>item\_class\_button
- cl\_column\_tree\_model=>item\_class\_checkbox
- cl\_column\_tree\_model=>item\_class\_link.

You can store a character string of any length in the TEXT field.

You store icons in the T\_IMAGE field, also an entry of the class “text”.

Usually, you need only fill these fields. You need the others only in special cases. For further information about these fields, refer to the online documentation.



## Passing Node Entries to the Column Tree Model Instance

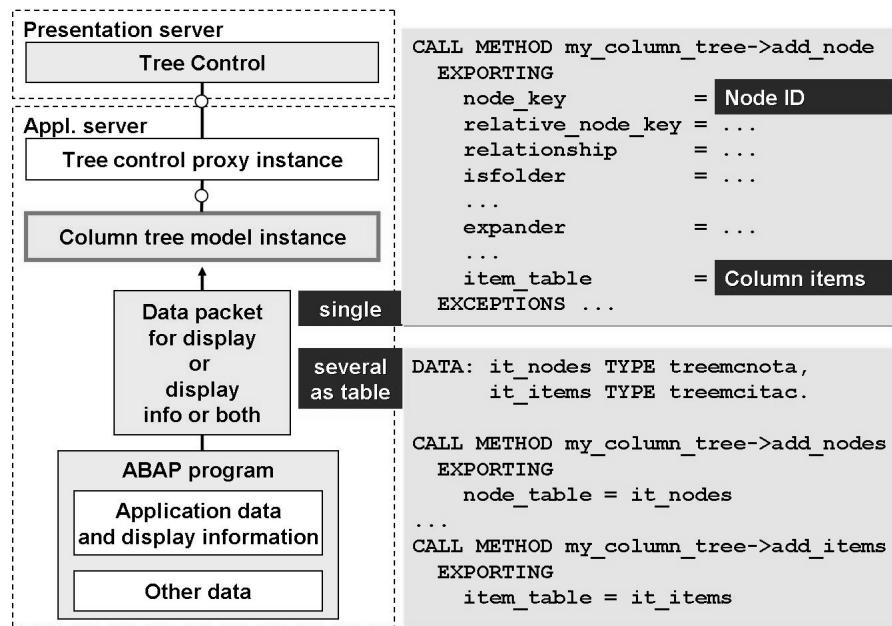


Figure 186: Passing Node Entries to the Column Tree Model Instance

As with the simple tree model, there are two possible ways to pass the node and column entries to the tree model instance:

- Either call the ADD\_NODE method and fill its parameters for the node entry as with the simple tree model. Pass the previously filled internal table for the column entries (global type TREEMCITAB) to the ITEM\_TABLE parameter. The method call is thus equivalent to inserting an entry in the node table.
- You use the ADD\_NODES method. In this case, you must first define your own node table from the global type TREEMCNOTA and fill it with the number of new node entries you want. In this case, you must first define your own column entry table from the global type TREEMCITAC and fill it with the number of new column entries you want. In this case you need to store the node ID again in the first column of the column entries table, to ensure that the entry is uniquely identified.

In SAP R/3 Basis Releases 4.6A and B you must also pass the name of your global structure type for the column entries (the TABLE\_STRUC\_NAME parameter) to the ADD\_NODES\_AND\_ITEMS method. You must also define the column entries table using it.



## Summary: Order of Procedures for the Column Tree Model

1. Create column tree model instance  
`CREATE OBJECT ...:`  
    - Column name for the hierarchy area
    - Column header and width
  2. Create column tree control instance  
`CALL METHOD ...->create_tree_control:`  
    - Reference to container
  3. Create columns  
`CALL METHOD ...->add_column:`  
    - Column name
    - Column heading
  4. Pass hierarchical and additional data to tree model instance
    - 4.1. Fill table for column entries that is, specify column contents for the node to be added now
    - 4.2. Pass node entry  
`CALL METHOD ...->add_node:`  
      - Node entry
      - Column entries
- For each column
- For each line (node)

Figure 187: Summary: Order of Procedures for the Column Tree Model

## List Entries in the List Tree Model

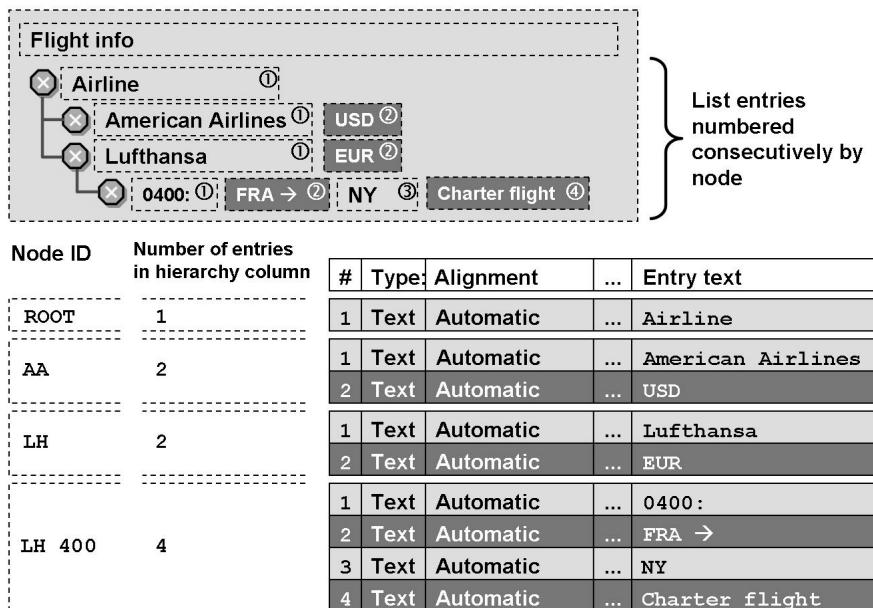


Figure 188: List Entries in the List Tree Model

For each node, specify how many list entries it will contain in the hierarchy area. You do this by filling the LAST\_HITEM parameter of the ADD\_NODE method.

For the list entries of each node, pass an internal table of the global type TREEMLITAB. These contain a line for each entry.

The lines contain the type of entry that **this column** should contain – generally text entries. Therefore the text must also be stored in the line.

Thus each list entry is generally uniquely identified using a value pair (node ID, entry), similar to a two-dimensional coordinates system.

To specify the alignment of the text entry, fill the ALIGNMENT field with one of the following class constants:

- cl\_list\_tree\_model=>align\_auto: Column resized to fit text
- cl\_list\_tree\_model=>align\_left: Left-aligned
- cl\_list\_tree\_model=>align\_right: Right-aligned

In Releases SAP R/3 Basis 4.6A and 4.6B you must define your own global type based on the global structure type TREEV\_ITEM.

## Summary: Order of Procedures for the List Tree Model



1. Create list tree model instance  
`CREATE OBJECT ...;`  
 • Allow user to select entries  
 • Header
2. Create column tree control instance  
`CALL METHOD ...->create_tree_control;`  
 • Reference to Container
3. Pass hierarchical and additional data to tree model instance
  - 3.1. Fill table for list entries  
 that is, specify contents for the node to be added now
 

For each line (node)
  - 3.2. Pass node entry  
`CALL METHOD ...->add_node;`
    - Node entry
    - List entries
    - Fill the LAST\_HITEM

Figure 189: Summary: Order of Procedures for the List Tree Model

## Demo Programs Delivered for the Tree Control

Package	Programs	Demonstrates
SEU_TREE_MODEL	SAPSIMPLE_TREE_MODEL_DEMO	Simple Tree Model
	SAPCOLUMN_TREE_MODEL_DEMO	Column Tree Model
	SAPLIST_TREE_MODEL_DEMO	List Tree Model
SEU_TREE_CONTROL	SAPSIMPLE_TREE_CONTROL_DEMO	Simple Tree Control
	SAPCOLUMN_TREE_CONTROL_DEMO	Column Tree Control
	SAPLIST_TREE_CONTROL_DEMO	List Tree Control
	SAPLIST_TREE_CONTROL_DEMO_HDR	List Tree Control with headers

- As well as these demonstration and test programs, you will also find other Repository objects for the SAP Grid Control in the **SEU\_TREE\_MODEL** and **SEU\_TREE\_CONTROL** packages.
- These packages are part of the standard **SAP R/3 System** – that is, they are always delivered with the ABAP Workbench.
- You can also analyze and execute these and other demonstration and test programs in a comfortable test environment. In the ABAP Workbench, choose *Environment* → *Examples* → *Controls Examples*.



# Exercise 16: Display Data in a Column Tree Model

## Exercise Objectives

After completing this exercise, you will be able to:

- Display hierarchically arranged data in a column tree model instance

## Business Example

Display data from the tables SCARR (airlines), SPFLI (connections) and SFLIGHT (flights) hierarchically in a tree model

**Program:** ZBC412\_##\_TRM\_EX3

**Template:** SAPBC412\_TRMS\_EXERCISE\_1

**Model solution:** SAPBC412\_TRMS\_EXERCISE\_3

where ## is the group number.

### Task 1:

Copy the template.

1. Copy your solution from the **first** exercise in this chapter (ZBC412\_##\_TRM\_EX1) or the appropriate model solution (SAPBC412\_TRMS\_EXERCISE\_1) assigning it the new name ZBC412\_##\_TRM\_EX3. Get to know how your copy of the program works

### Task 2:

In the `init_tree_processing_0100` module, create a column tree model instance, which is displayed in the existing docking container as follows:

1. Change the type of the reference variable `ref_tree_model` so that it points to an instance of the **column** tree model.
2. Create a column tree instance. Users should only be able to select one node at a time. You need a work area for the function attributes (suggested name: `wa_hier_header`). If an error occurs while the object is being generated, your program should terminate, displaying the message 043.

Create the tree control proxy instance and link it with the docking container control (use method `create_tree_control`). If an error occurs, the program should terminate, displaying message 042.

*Continued on next page*

3. Define a subroutine that creates the columns (suggested name: `add_columns`) and call it here. Add the four columns *PRICE*, *CURRENCY*, *PLANETYPE* and *SEATSFREE*. Provide appropriate widths and header texts for the columns.

### Task 3:

Pass some values from the data (buffered in internal tables `it_scarr`, `it_spfli` and `it_sfflight`) to your tree model instance. To do this, extend the existing subroutine `add_nodes` that you call from the `init_tree_processing_0100` module as follows:

1. Change the type of the work area for the node entry `l_wa_node` to `treemcnodt`.  
Define a local internal table for the column entries (suggested name: `l_it_items`).
2. Adapt the interface of the `add_node` method. Instead of the node texts pass the internal table `l_it_items` with the column entries. If errors occur, terminate processing and display message 012.
3. The internal table must be filled with the column entries for the current node to be added. To do this, define a subroutine that fills the internal table with passed texts (suggested name `fill_itemtab`).

In this subroutine, define a local work area for a column entry (suggested name: `l_wa_item`) with the line type of the table you want to change. First initialize the entry table.

For each column entry fill the field for the technical column name, for the entry type, and for the entry text, in the work area. Then insert the filled work area in the entry table

4. Call your subroutine `fill_itemtab` now before calling the `add_node` method for the first time. Fill only the first column with meaningful texts, up to the penultimate hierarchy level. (Base your texts on those entered until now.) For the flight nodes add the flight date, price, currency code, airplane type, and number of free seats. You must first calculate the last of these. If an error occurs, end processing with the termination message 075.

 **Note:** Make sure you format the price as you did with the date.

## Solution 16: Display Data in a Column Tree Model

### Task 1:

Copy the template.

1. Copy your solution from the **first** exercise in this chapter (ZBC412\_##\_TRM\_EX1) or the appropriate model solution (SAPBC412\_TRMS\_EXERCISE\_1) assigning it the new name ZBC412\_##\_TRM\_EX3. Get to know how your copy of the program works
  - a) -

### Task 2:

In the init\_tree\_processing\_0100 module, create a column tree model instance, which is displayed in the existing docking container as follows:

1. Change the type of the reference variable ref\_tree\_model so that it points to an instance of the **column** tree model.
  - a)

```
DATA:  
  ...  
  ref_tree_model TYPE REF TO cl_column_tree_model.
```
2. Create a column tree instance. Users should only be able to select one node at a time. You need a work area for the function attributes (suggested name: wa\_hier\_header). If an error occurs while the object is being generated, your program should terminate, displaying the message 043.

*Continued on next page*

Create the tree control proxy instance and link it with the docking container control (use method `create_tree_control`). If an error occurs, the program should terminate, displaying message 042.

a)

```

DATA:
...
wa_hier_header TYPE treemhhdr.
...
MODULE init_tree_processing_0100 OUTPUT.
IF ref_tree_model IS INITIAL.
wa_hier_header-t_image = icon_ws_plane.
wa_hier_header-heading = text-hir.
wa_hier_header-width = 65.

CREATE OBJECT ref_tree_model
EXPORTING
node_selection_mode = cl_column_tree_model->node_sel_mode_single
hierarchy_column_name = 'HIER'
hierarchy_header = wa_hier_header
EXCEPTIONS
others = 1.
IF sy-subrc <> 0.
MESSAGE a043.
ENDIF.
...
PERFORM add_columns USING ref_tree_model.
...

```

3. Define a subroutine that creates the columns (suggested name: `add_columns`) and call it here. Add the four columns *PRICE*, *CURRENCY*, *PLANETYPE* and *SEATSFREE*. Provide appropriate widths and header texts for the columns.

a)

```

FORM add_columns USING l_ref_tree_model
TYPE REF TO cl_column_tree_model.
CALL METHOD l_ref_tree_model->add_column
EXPORTING
name = 'PRICE'
width = 40
header_text = text-pri
EXCEPTIONS
OTHERS = 1.
IF sy-subrc <> 0.
MESSAGE a012.

```

*Continued on next page*

```

ENDIF.

CALL METHOD l_ref_tree_model->add_column
EXPORTING
  name          = 'CURRENCY'
  width         = 10
  header_text   = text-cur
EXCEPTIONS
  OTHERS        = 1.

IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

CALL METHOD l_ref_tree_model->add_column
EXPORTING
  name          = 'PLANETYPE'
  width         = 20
  header_text   = text-pty
EXCEPTIONS
  OTHERS        = 1.

IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

CALL METHOD l_ref_tree_model->add_column
EXPORTING
  name          = 'SEATSFREE'
  width         = 20
  header_text   = text-fre
EXCEPTIONS
  OTHERS        = 1.

IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

ENDFORM.

```

### Task 3:

Pass some values from the data (buffered in internal tables `it_scarr`, `it_spfli` and `it_sflight`) to your tree model instance. To do this, extend the existing subroutine `add_nodes` that you call from the `init_tree_processing_0100` module as follows:

1. Change the type of the work area for the node entry `l_wa_node` to `treemcnodt`.

*Continued on next page*

Define a local internal table for the column entries (suggested name: l\_it\_items).

a)

```
FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_column_tree_model.  
DATA:  
    l_wa_node  TYPE treemcnodt,  
    l_it_items TYPE treemcitab,  
    ...
```

*Continued on next page*

2. Adapt the interface of the `add_node` method. Instead of the node texts pass the internal table `l_it_items` with the column entries. If errors occur, terminate processing and display message 012.

a)

```

FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_column_tree_model.

DATA:
  l_wa_node  TYPE treemcnodt,
  l_it_items TYPE treemcitetab,
  l_conn_text(40)  TYPE c,
  l_date_text(10)  TYPE c,
  l_price_text(20)  TYPE c,
  l_seats_text(10)  TYPE c.
  PERFORM fill_itemtab
    USING
      text-car
      space
      space
      space
      space
    CHANGING
      l_it_items.

CALL METHOD l_ref_tree_model->add_node
  EXPORTING
    node_key          = 'ROOT'
  *   RELATIVE_NODE_KEY      =
  *   RELATIONSHIP        =
    isfolder           = 'X'
    expander           = 'X'
    item_table         = l_it_items
  EXCEPTIONS
    OTHERS             = 5.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

```

3. The internal table must be filled with the column entries for the current node to be added. To do this, define a subroutine that fills the internal table with passed texts (suggested name `fill_itemtab`).

In this subroutine, define a local work area for a column entry (suggested name: `l_wa_item`) with the line type of the table you want to change. First initialize the entry table.

*Continued on next page*

For each column entry fill the field for the technical column name, for the entry type, and for the entry text, in the work area. Then insert the filled work area in the entry table

a)

```

FORM fill_itemtab USING      p_hier_text    TYPE c
                           p_price_text   TYPE c
                           p_curr_text   TYPE c
                           p_plane_text   TYPE c
                           p_free_text   TYPE c
                           CHANGING p_it_items TYPE treemcitab.
DATA l_wa_item LIKE LINE OF p_it_items.
CLEAR p_it_items.
l_wa_item-class      = cl_column_tree_model->item_class_text.
l_wa_item-item_name = 'HIER'.
l_wa_item-text       = p_hier_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'PRICE'.
l_wa_item-text       = p_price_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'CURRENCY'.
l_wa_item-text       = p_curr_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'PLANETYPE'.
l_wa_item-text       = p_plane_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'SEATSFREE'.
l_wa_item-text       = p_free_text.
INSERT l_wa_item INTO TABLE p_it_items.
ENDFORM.
```

4. Call your subroutine *fill\_itemtab* now before calling the *add\_node* method for the first time. Fill only the first column with meaningful texts, up to the penultimate hierarchy level. (Base your texts on those entered until now.) For the flight nodes add the flight date, price, currency code, airplane type, and number of free seats. You must first calculate the last of these. If an error occurs, end processing with the termination message 075.



**Note:** Make sure you format the price as you did with the date.

a)

```

FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_column_tree_model.
DATA:
l_wa_node  TYPE treemnodt,
l_it_items TYPE treemcitab,
```

*Continued on next page*

```

        l_conn_text(40)    TYPE c,
        l_date_text(10)    TYPE c,
        l_price_text(20)   TYPE c,
        l_seats_text(10)   TYPE c.

        PERFORM fill_itemtab
           USING
              text-car
              space
              space
              space
              space
           CHANGING
              l_it_items.

        CALL METHOD l_ref_tree_model->add_node
        ...

```

## Result

The complete source code for the model solution is shown below.

### ABAP program

#### *Data Declarations*

```

REPORT  sapbc412_trms_exercise_3 MESSAGE-ID bc412.

...
* content:
ref_tree_model  TYPE REF TO cl_column_tree_model,

* auxiliary:
wa_hier_header  TYPE treemhhdr.

```

#### *Modules*

```

*&-----*
*&      Module  init_tree_processing_0100  OUTPUT
*&-----*
*      create tree object, link to container and fill with data

```

*Continued on next page*

```

*-----*
MODULE init_tree_processing_0100 OUTPUT.
  IF ref_tree_model IS INITIAL.

    wa_hier_header-t_image = icon_ws_plane.
    wa_hier_header-heading = text-hir.
    wa_hier_header-width   = 65.

    CREATE OBJECT ref_tree_model
      EXPORTING
        node_selection_mode = cl_column_tree_model->node_sel_mode_single
        hierarchy_column_name = 'HIER'
        hierarchy_header      = wa_hier_header
      EXCEPTIONS
        others                 = 1.
  IF sy-subrc <> 0.
    MESSAGE a043.
  ENDIF.

  CALL METHOD ref_tree_model->create_tree_control
    EXPORTING
      parent                  = ref_cont_left
    EXCEPTIONS
      OTHERS                 = 1.
  IF sy-subrc <> 0.
    MESSAGE a042.
  ENDIF.

  PERFORM add_columns USING ref_tree_model.
  PERFORM add_nodes   USING ref_tree_model.

  ENDIF.
ENDMODULE.          " init_tree_processing_0100  OUTPUT

```

### ***Subroutines***

```

*&-----*
*&      Form  FREE_CONTROL_RESSOURCES
*&-----*
*      free control ressources on the presentation server
*      free all reference variables (ABAP object) -> garbage collector
*-----*
*      no interface

```

*Continued on next page*

```

*-----*
FORM free_control_ressources.
CALL METHOD ref_cont_left->free.

FREE:
  ref_tree_model,
  ref_cont_left.

ENDFORM.                                     " FREE_CONTROL_RESSOURCES

*-----*
*      FORM add_columns                      *
*-----*
*  -->  L_REF_TREE_MODEL                   *
*-----*
FORM add_columns USING l_ref_tree_model
  TYPE REF TO cl_column_tree_model.

  CALL METHOD l_ref_tree_model->add_column
    EXPORTING
      name          = 'PRICE'
      width         = 40
      header_text   = text-pri
    EXCEPTIONS
      OTHERS        = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

  CALL METHOD l_ref_tree_model->add_column
    EXPORTING
      name          = 'CURRENCY'
      width         = 10
      header_text   = text-cur
    EXCEPTIONS
      OTHERS        = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

  CALL METHOD l_ref_tree_model->add_column
    EXPORTING
      name          = 'PLANETYPE'
      width         = 20

```

*Continued on next page*

```

        header_text      = text-ptx
EXCEPTIONS
        OTHERS          = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

CALL METHOD l_ref_tree_model->add_column
EXPORTING
        name           = 'SEATSFREE'
        width          = 20
        header_text    = text-fre
EXCEPTIONS
        OTHERS          = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

ENDFORM.

*-----*
*      FORM ADD_NODES
*-----*
*-----*
*      build up a hierarchy consisting of
*      carriers, connections and
*      flight dates with additional data
*-----*
*  --> L_REF_TREE_MODEL
*-----*
FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_column_tree_model.

DATA:
        l_wa_node  TYPE treemcnodt,
        l_it_items TYPE treemcitab,

        l_conn_text(40)  TYPE c,
        l_date_text(10)  TYPE c,
        l_price_text(20) TYPE c,
        l_seats_text(10) TYPE c.

PERFORM fill_itemtab
        USING
                text-car
                space

```

*Continued on next page*

```

space
space
space
CHANGING
l_it_items.

CALL METHOD l_ref_tree_model->add_node
EXPORTING
    node_key          = 'ROOT'
    *     RELATIVE_NODE_KEY      =
    *     RELATIONSHIP          =
    isfolder           = 'X'
    expander           = 'X'
    item_table         = l_it_items
EXCEPTIONS
    OTHERS             = 5.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

* scarr-nodes:
LOOP AT it_scarr INTO wa_scarr.
CLEAR l_wa_node.
l_wa_node-node_key = wa_scarr-carrid.

PERFORM fill_itemtab
    USING
        wa_scarr-carrname
        space
        space
        space
        space
        space
CHANGING
l_it_items.

CALL METHOD l_ref_tree_model->add_node
EXPORTING
    node_key          = l_wa_node-node_key
    relative_node_key = 'ROOT'
    relationship       = cl_column_tree_model->relat_last_child
    isfolder           = 'X'
    expander           = 'X'
    item_table         = l_it_items
EXCEPTIONS
    OTHERS             = 5.

```

*Continued on next page*

```

        IF sy-subrc <> 0.
          MESSAGE a012.
        ENDIF.
      ENDOLOOP.

* spfli-nodes:
  LOOP AT it_spfli INTO wa_spfli.
    CLEAR l_wa_node.
    CONCATENATE wa_spfli-carrid
      wa_spfli-connid
      INTO l_wa_node-node_key
      SEPARATED BY space.
    l_wa_node-relatkey = wa_spfli-carrid.

    CONCATENATE wa_spfli-carrid
      wa_spfli-connid
      ':'
      wa_spfli-cityfrom
      '->'
      wa_spfli-cityto
      INTO l_conn_text
      SEPARATED BY space.
    PERFORM fill_itemtab
      USING
        l_conn_text
        space
        space
        space
        space
      CHANGING
        l_it_items.

CALL METHOD l_ref_tree_model->add_node
  EXPORTING
    node_key           = l_wa_node-node_key
    relative_node_key = l_wa_node-relatkey
    relationship       = cl_column_tree_model=>relat_last_child
    isfolder          = 'X'
    expander          = 'X'
    item_table         = l_it_items
  EXCEPTIONS
    OTHERS             = 5.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

```

*Continued on next page*

```

ENDLOOP.

* sflight-nodes:
LOOP AT it_sflight INTO wa_sflight.
  CLEAR l_wa_node.
  CONCATENATE wa_sflight-carrid
    wa_sflight-connid
    wa_sflight-fldate
    INTO l_wa_node-node_key
    SEPARATED BY space.
  CONCATENATE wa_sflight-carrid
    wa_sflight-connid
    INTO l_wa_node-relatkey
    SEPARATED BY space.

  WRITE wa_sflight-fldate TO l_date_text.
  WRITE wa_sflight-price TO l_price_text
    CURRENCY wa_sflight-currency.
  IF wa_sflight-seatsmax >= wa_sflight-seatsocc.
    l_seats_text = wa_sflight-seatsmax - wa_sflight-seatsocc.
  ELSE.
    MESSAGE a075.
  ENDIF.

  PERFORM fill_itemtab
    USING
      l_date_text
      l_price_text
      wa_sflight-currency
      wa_sflight-planetype
      l_seats_text
    CHANGING
      l_it_items.

CALL METHOD l_ref_tree_model->add_node
  EXPORTING
    node_key          = l_wa_node-node_key
    relative_node_key = l_wa_node-relatkey
    relationship      = cl_column_tree_model->relat_last_child
    isfolder          = space
    expander          = space
    item_table        = l_it_items
  EXCEPTIONS
    OTHERS            = 5
.
.
```

*Continued on next page*

```

        IF sy-subrc <> 0.
          MESSAGE a012.
        ENDIF.
      ENDOLOOP.

      ENDFORM.                                     " ADD_NODES

*-----*
*      FORM fill_itemtab                      *
*-----*
*      adds one line into the item table for each column of the tree *
*-----*
*  --> P_IT_ITEMS                           *
*-----*

FORM fill_itemtab USING      p_hier_text  TYPE c
                         p_price_text TYPE c
                         p_curr_text  TYPE c
                         p_plane_text TYPE c
                         p_free_text   TYPE c
CHANGING p_it_items TYPE treemcitab.
DATA l_wa_item LIKE LINE OF p_it_items.

CLEAR p_it_items.
l_wa_item-class      = cl_column_tree_model->item_class_text.
l_wa_item-item_name = 'HIER'.
l_wa_item-text       = p_hier_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'PRICE'.
l_wa_item-text       = p_price_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'CURRENCY'.
l_wa_item-text       = p_curr_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'PLANETYPE'.
l_wa_item-text       = p_plane_text.
INSERT l_wa_item INTO TABLE p_it_items.
l_wa_item-item_name = 'SEATSFREE'.
l_wa_item-text       = p_free_text.
INSERT l_wa_item INTO TABLE p_it_items.

ENDFORM.

```

## Exercise 17: Icons as Column Entries (OPTIONAL)

### Exercise Objectives

After completing this exercise, you will be able to:

- Display icons in a column tree model instance.

### Business Example

Extend the column tree control to include an icon column. Make the program display a “red light” if a flight is full and a green light if there are stills seats available.

**Program:** ZBC412\_##\_TRM\_EX3\_OPT

**Template:** SAPBC412\_TRMS\_EXERCISE\_3

**Model solution:** SAPBC412\_TRMS\_EXERCISE\_3\_OPT

where ## is the group number.

#### Task 1:

Copy the template.

1. Copy your solution from the previous exercise (ZBC412\_##\_TRM\_EX3) or the appropriate model solution (SAPBC412\_TRMS\_EXERCISE\_3) to the name **ZBC412\_##\_TRM\_EX3\_OPT**. Get to know how your copy of the program works.

#### Task 2:

Create an additional column to display a traffic light for flight nodes. Choose a red traffic light for fully booked flights and a green one for flights where there are still seats available.

1. Extend the `add_columns` subroutine.

Add a column named *ICON* with a width of 6 between the *PLANETYPE* and *SEATSFREE* columns. Do not create a header for it.

2. Extend the `fill_itemtab` subroutine.

*Continued on next page*

First check whether you are to provide entries for a flight node. To do this you can use the parameter for free seats: If it is not initial, you can assume that you are to provide entries for a flight.



**Hint:** If you typed this parameter as a string, the “0” value is **not** the initial value.

For a flight node then check whether or not the flight is fully booked: Using the comparison operator CO, query whether the parameter contains only the characters “0” and “ ”. If that is true set the icon value to icon\_red\_light. Otherwise use the icon value icon\_green\_light. (To use the icon names, you need to load the type group *icon*).

## Solution 17: Icons as Column Entries (OPTIONAL)

### Task 1:

Copy the template.

1. Copy your solution from the previous exercise (ZBC412\_##\_TRM\_EX3) or the appropriate model solution (SAPBC412\_TRMS\_EXERCISE\_3) to the name **ZBC412\_##\_TRM\_EX3\_OPT**. Get to know how your copy of the program works.
  - a) —

### Task 2:

Create an additional column to display a traffic light for flight nodes. Choose a red traffic light for fully booked flights and a green one for flights where there are still seats available.

1. Extend the `add_columns` subroutine.

Add a column named *ICON* with a width of 6 between the *PLANETYPE* and *SEATSFREE* columns. Do not create a header for it.

- a)

```

TYPE-POOLS icon.

DATA ...


FORM add_columns USING l_ref_tree_model
  TYPE REF TO cl_column_tree_model.
  ...

CALL METHOD l_ref_tree_model->add_column
  EXPORTING
    name          = 'ICON'
    width         = 6
    header_text   = space
  EXCEPTIONS
    OTHERS        = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.
  ...

```

*Continued on next page*

2. Extend the `fill_itemtab` subroutine.

First check whether you are to provide entries for a flight node. To do this you can use the parameter for free seats: If it is not initial, you can assume that you are to provide entries for a flight.



**Hint:** If you typed this parameter as a string, the “0” value is **not** the initial value.

For a flight node then check whether or not the flight is fully booked:  
Using the comparison operator CO, query whether the parameter contains only the characters “0” and “ ”. If that is true set the icon value to `icon_red_light`. Otherwise use the icon value `icon_green_light`. (To use the icon names, you need to load the type group `icon`).

a)

```

FORM fill_itemtab USING      p_hier_text  TYPE c
                           p_price_text TYPE c
                           p_curr_text  TYPE c
                           p_plane_text  TYPE c
                           p_free_text   TYPE c
CHANGING p_it_items TYPE treemcitab.

DATA l_wa_item LIKE LINE OF p_it_items.

CLEAR p_it_items.
l_wa_item-class      = cl_column_tree_model=>item_class_text.

...
l_wa_item-item_name = 'ICON'.
* figure out, which icon is appropriate:
IF NOT p_free_text IS INITIAL.
  IF p_free_text CO '0'.
    l_wa_item-t_image = icon_red_light.
  ELSE.
    l_wa_item-t_image = icon_green_light.
  ENDIF.
ENDIF.
CLEAR l_wa_item-text.
INSERT l_wa_item INTO TABLE p_it_items.
...
ENDFORM.

```



## Lesson Summary

You should now be able to:

- Use the additional functions of the Tree Model Classes



## Unit Summary

You should now be able to:

- Name the different type of Tree Control and describe their main features
- Display data in the different Tree Control variants
- Create hierarchies
- Integrate additional data if necessary
- Create an instance of the SAP Easy Splitter Container Control
- Use the SAP Easy Splitter Container
- Create a hierarchy in a simple tree
- Types of relationships
- Passing node entries
- Print the Tree Content
- Search in the Tree
- Name some of the events raised by the Tree Control
- Find the application data belonging to the node raising the event
- Create an application toolbar
- Use the additional functions of the Tree Model Classes

# Unit 8

## Drag&Drop Functions

### Unit Overview

This unit covers the techniques needed to implement Drag&Drop Functions such as, Behavior types, Data transport, Event handler methods and Runtime optimization.



### Unit Objectives

After completing this unit, you will be able to:

- Name the uses of Drag&Drop functions
- Describe the technical sequence of a Drag&Drop procedure
- Implement copying procedures between different control objects or sub-objects
- Implement move procedures between different control objects or sub-objects

### Unit Contents

Lesson: Drag & Drop Functions .....	452
Exercise 18: Drag&Drop Functions .....	467

# Lesson: Drag & Drop Functions

## Lesson Overview

This lesson covers the techniques needed to implement Drag&Drop Functions such as, Behavior types, Data transport, Event handler methods and Runtime optimization.



## Lesson Objectives

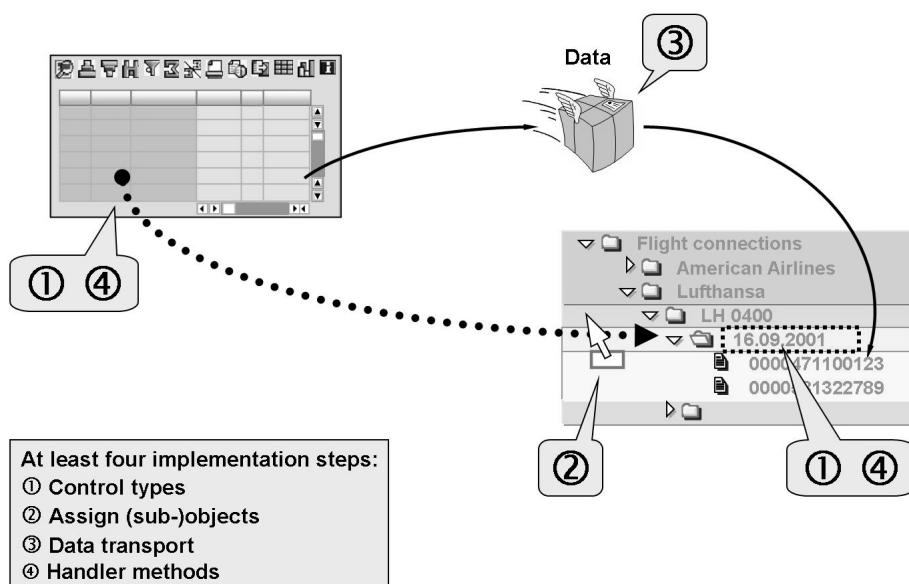
After completing this lesson, you will be able to:

- Name the uses of Drag&Drop functions
- Describe the technical sequence of a Drag&Drop procedure
- Implement copying procedures between different control objects or sub-objects
- Implement move procedures between different control objects or sub-objects

## Business Example

The user should be able to drag a customer data record from a SAP List Viewer Control to a flight connection node in a simple tree control, in order to book this customer on this flight. Secondly the user should be able to drag a booking node from the flight bookings tree to a picture (trash can) to cancel the booking.

## The Concept of the Drag&Drop Relationship



**Figure 190: The Concept of the Drag&Drop Relationship**

To realize a Drag&Drop procedure you need to implement at least the following four steps. These steps also describe how Drag&Drop procedures are put into action by the system:

- When the user carries out a Drag&Drop operation with the mouse, the system checks whether or not this operation can be performed on the control (sub-)objects involved. This means that you must define valid behavior types. You can bundle behavior types together.
- In the second step, you must have assigned the control (sub-)object to one of these **behavior type groups**. In this way, these (sub-)objects belong to a specific **control type** that represents the sum of its possible behavior types.
- Data objects need to be generated for the data that will be transported as part of the Drag&Drop procedure. To this end, you must have defined (local) classes.
- Finally, you must define at least two handler methods – one for the drag event and one for the drop event of the relevant control. In these handler methods you deal with transporting the data, adapting the relevant control, and any follow-up activities.

In this chapter, we will implement a solution to the following two requirements as an example: The user should be able to drag a customer data record from a SAP Grid Control to a flight connection node in a simple tree control, in order to book this customer on this flight. Secondly the user should be able to drag a booking node from the flight bookings tree to a picture (trash can) to cancel the booking. We will **not** be changing the database to reflect the operations we have performed. Instead, we will limit our example to the control-based aspects of implementation in the context of this course.

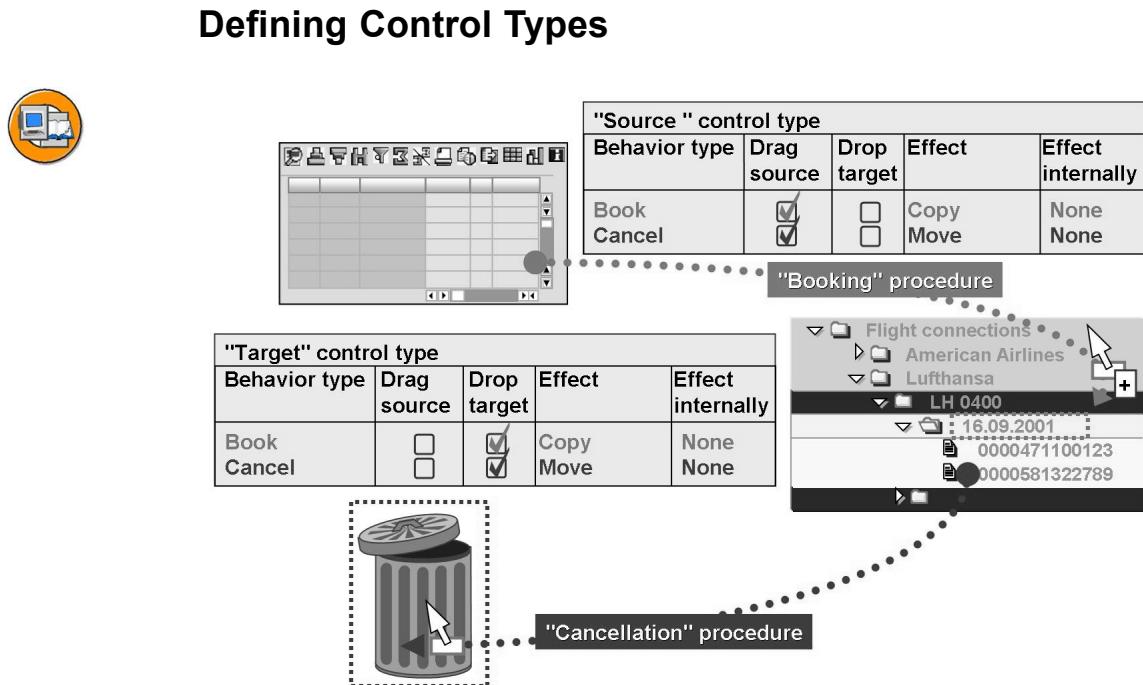


Figure 191: Defining Control Types

A **behavior type** (also known as a **flavor**) has the following attributes:

- A name for identification (can be anything you like)
- A flag indicating whether it is the starting point for a drag operation or the destination of a drop operation
- A flag indicating which mouse operations this flavor should use
- A flag indicating which mouse operations this flavor should use within the control – if any

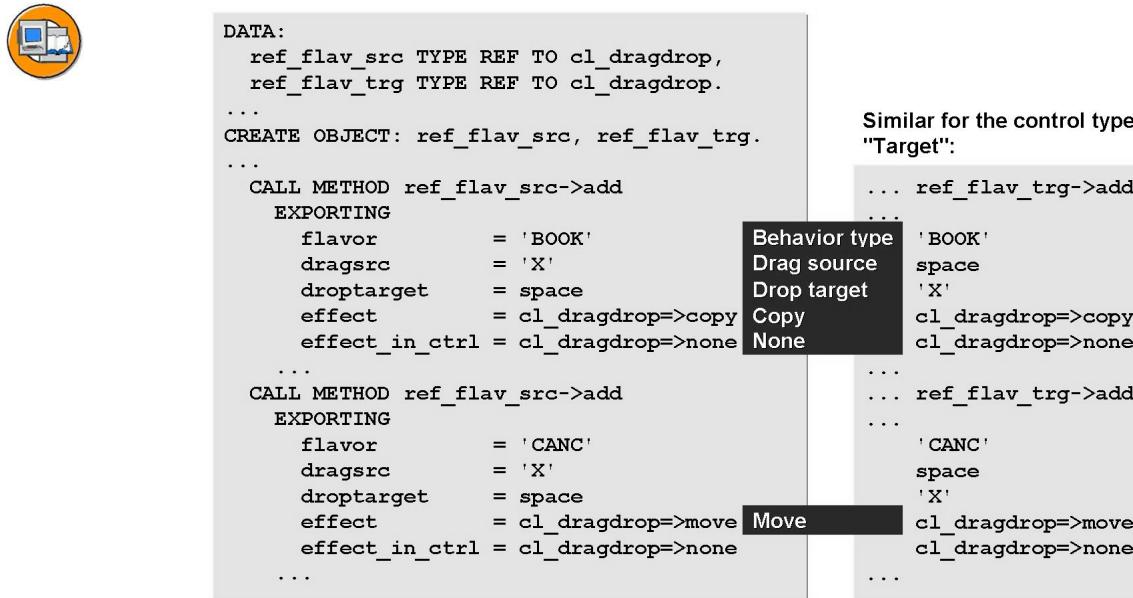
Single flavors are generally bundled together in tables.

Later, individual control (sub-)objects will be assigned to such tables. All the objects grouped together in this way will thus support the same flavors at runtime.

Note that the two control types shown above do **not** correspond to the (business) Drag&Drop procedures "Book" and "Cancel". Instead, such a procedure is the result of a **pair of two identically-named flavors**, which are supported by the relevant control (sub-)objects.

You may have noticed that this means that individual control (sub-)objects may be assigned to flavors that they do not need at all. For example, if the trash can is assigned to the "Target" type, it supports not only the "Cancellation target" but also the "Booking target" flavor. It may seem better to define the control types differently.

## Syntax Example: Defining Control Types



**Figure 192: Syntax Example: Defining Control Types**

To create the behavior type tables mentioned above (Control Types) you must create a separate instance of the global class CL\_DRAGDROP for each control type.

You must then insert each behavior type (flavor) by calling the ADD method for each one. When doing so, take into account the identical names of flavors that belong logically together in the different tables.

To specify the effects use the predefined constants of the class **CL\_DRAGDROP**: **copy**, **move**, and **none**.

## Assigning Control (Sub-)Objects to Control Types

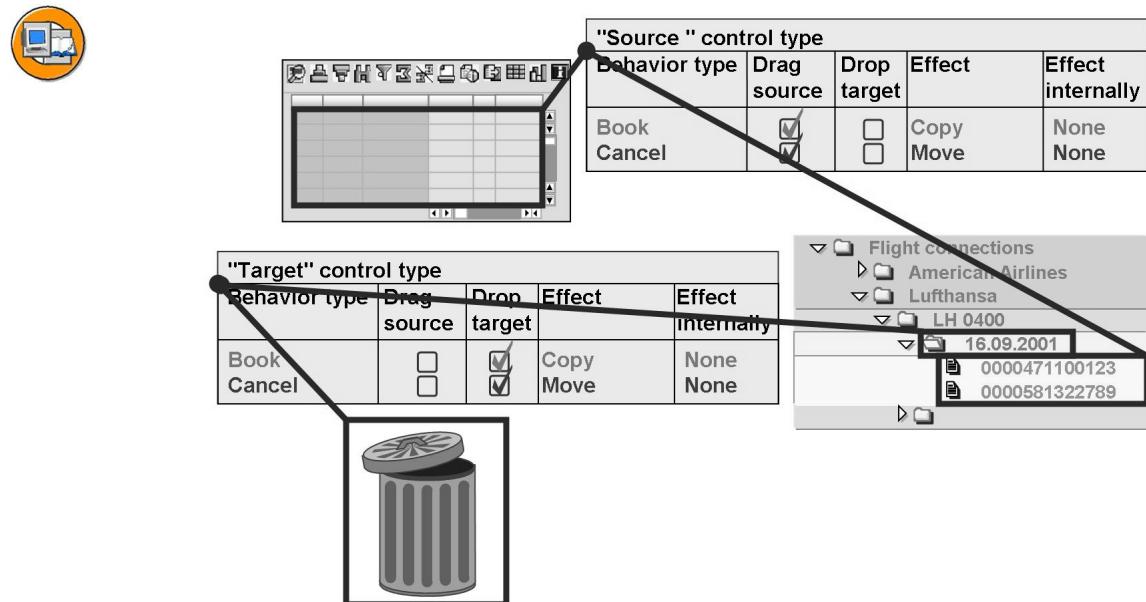


Figure 193: Assigning Control (Sub-)Objects to Control Types

You specify which source-target relationship is valid, and between which objects, by assigning individual control (sub-)objects to the control types.

In this example all the lines of the “Customer” grid control should have the control type “Source”. Theoretically, this means that each line can be seen as a “Booking source” and a “Cancellation source”. Conversely, the flight **connection** node is a “Booking target”. Thus it must have the control type “Target”. So theoretically, each node can also be a “Cancellation target”. The same applies to the trash can picture. The flight **booking** node itself must also have the type “Source”.

For example, if a customer data record in the grid is dragged to a flight connection node in the tree, the following sequence of events takes place: The system finds out what control sub-objects are involved – here the row of the grid and the flight connection node. Then, from the assignments, it finds two flavors to which **both controls involved are jointly assigned**. (Identically-named) The type of mouse operation now establishes which flavor will be used (that is, Copy if the Ctrl key is pressed, otherwise Move) You can then query this information later, to carry out different actions.

In this example, if the **effects** of the two behavior types “Book” and “Cancel” had been given identical values, there would be no way of distinguishing which mouse operation had been chosen. In such cases, you must specify a runtime which flavor is to be used. For further details, refer to the online documentation.

## Syntax Example: Assigning Control (Sub-)Objects to Control Types



```

DATA: handle_src TYPE i, handle_trg TYPE i.
...
CALL METHOD ref_flav_src->get_handle
  IMPORTING
    handle      = handle_src
  ...
* ... assign handle_src to Control elements ...
* ... here: grid area
  ...
* ... and: booking nodes of tree (while adding them on drop event)
  ...
CALL METHOD ref_flav_trg->get_handle
  IMPORTING
    handle      = handle_trg
  ...
* ... assign handle_trg to Control elements ...
* ... here: flight date notes (while creating hierarchy)
  ...
CALL METHOD ref_bin_pic->set_dragdrop_picture
  EXPORTING
    dragdrop = ref_flav_trg
  ...

```

Implementation depends on the control class

**Figure 194: Syntax Example: Assigning Control (Sub-)Objects to Control Types**

Assigning controls or their subobjects to the previously-defined control types takes place in different ways depending on the control class.

In the above example, you must have an identifier returned by the control types. To do this, call the GET\_HANDLE method for each instance. Store this identifier in a type i field (here: **handle\_src** and **handle\_trg**).

Then you need to assign these identifiers to the control sub-objects. To assign the entire grid area, use the SET\_TABLE\_FOR\_FIRST\_DISPLAY method. In the tree, you must fill the attributes of each node appropriately.

You can assign the trash can without an identifier, simply by passing a reference directly to the control type. To do this, call the SET\_DRAGDROP\_PICTURE method.

Thus for this step in the implementation, it is **absolutely essential** to study the documentation on each control. However, the techniques introduced here cover the usual situations you will encounter.



## The Object Used for Data Transport

The object can be instantiated by the handler methods:



```
CLASS lcl_booking DEFINITION.  
  PUBLIC SECTION.  
    DATA:  
      wa_customer TYPE scustom,  
      wa_node     TYPE treemsnadt.  
  ENDCLASS.
```

**Figure 195: The Object Used for Data Transport**

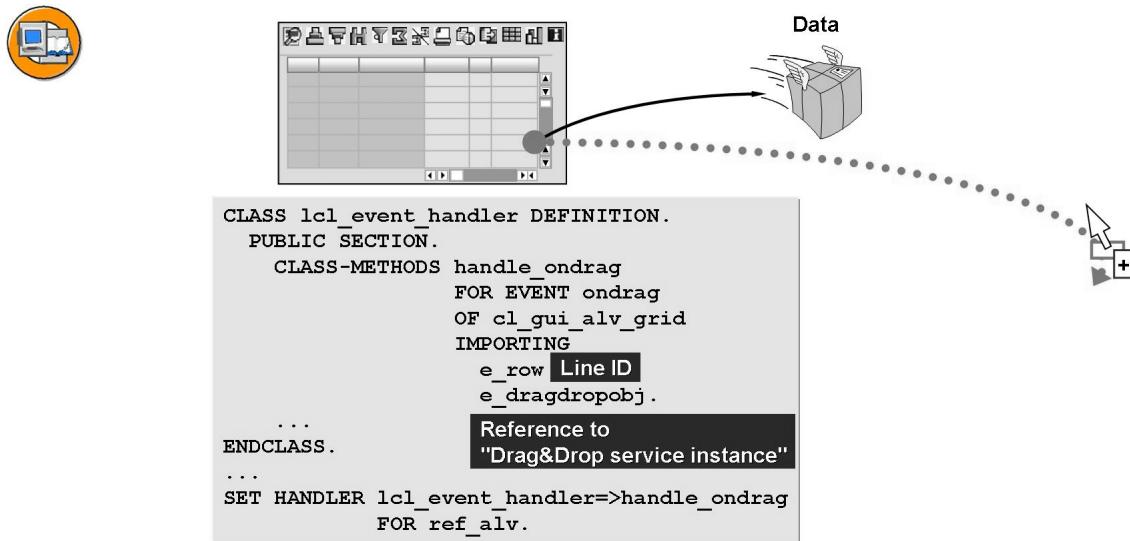
In a Drag&Drop procedure, you usually need to transport application data from the source object to the target object.

The system keeps a reference that can point to any object.

Accordingly, you must define a class in such a way that its instances can include the relevant application data. Within the public section of your class therefore, define the appropriate attributes (data objects). Since this data transport is program-specific, it is usually better to define this class **locally**.

Notes on the above example: The program will need to be able to transport both a customer data record (for the booking) and a booking key (for the cancellation). For this reason, we have defined two appropriately-typed structures in the local class **lcl\_booking**: **wa\_customer** for the customer data record and **wa\_node** for the tree node that in this case contains the booking key as part of the node ID.

## Syntax Example: Definition Part of the Handler Method for the Drag Event



**Figure 196: Syntax Example: Definition Part of the Handler Method for the Drag Event**

In the above example, you must define a handler method for the drag event of the SAP Grid Control.

In the implementation part, you will have to specify **which** customer data record the user wants to “drag”. The event will provide an appropriate parameter that must be imported.

The reference to the data transport instance must be passed to the “Drag&Drop service instance” that is generated by the system for every Drag&Drop procedure. The event will provide a reference to this “service instance” using an appropriate parameter that must be imported.

The “Drag&Drop service instance” also contains other attributes - such as the flavor used for the current Drag&Drop procedure.

The technical name of the drag event and the interface parameters are independent of the control class. Thus for this step in the implementation, it is **absolutely essential** to take the names from the online documentation or from the Class Builder.

## Syntax Example: Implementation Part of the Handler Method for the Drag Event



```
CLASS lcl_event_handler IMPLEMENTATION.
METHOD handle_ondrag.
DATA l_ref_booking TYPE REF TO lcl_booking.

CREATE OBJECT l_ref_booking.
READ TABLE it_scustom INTO l_ref_booking->wa_customer
INDEX e_row-index.

Data transport object
e_dragdropobj->object = l_ref_booking.
ENDMETHOD.

...
ENDCLASS.
```



**Figure 197: Syntax Example: Implementation Part of the Handler Method for the Drag Event**

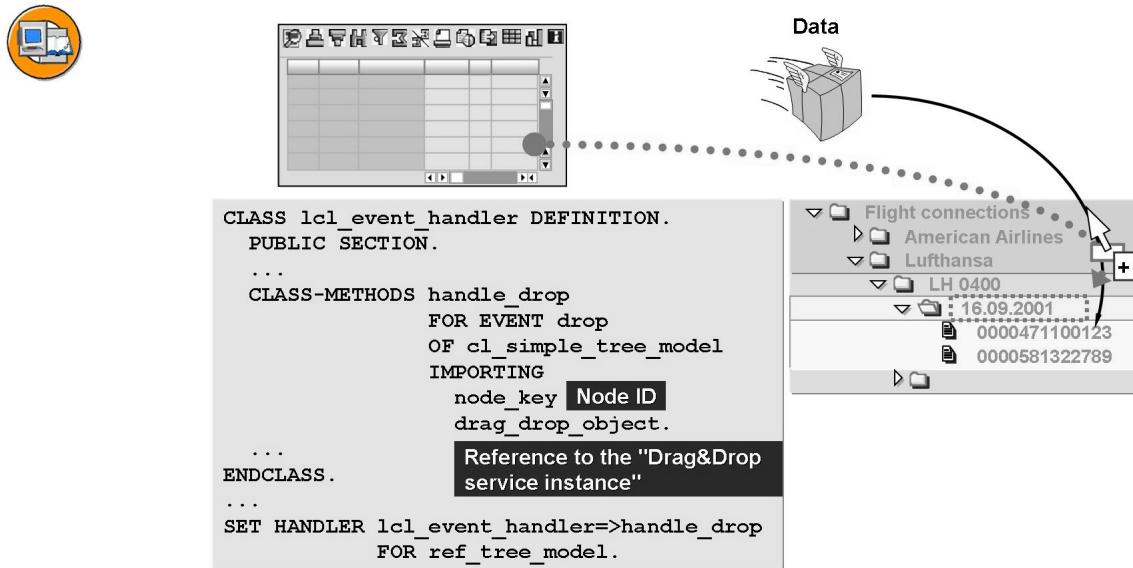
In the above example, the handler method for the drag event of the SAP Grid Control must perform only one task: Transport the application data of the “dragged” row in the grid into the data transport instance.

Define a local reference to an instance of your data transport class for this purpose (`l_ref_booking` in the example above).

Create this instance and fill the attribute for the customer data record with the application data that belong to the “dragged” row in the grid.

Now point the **object** reference of the “Drag&Drop service instance” to the data transport instance you have created. Even after your handler method has ended, the rest of the Drag&Drop procedure can read this data again using this instance.

## Syntax Example: Definition Part of the Handler Method for the Drop Event



**Figure 198: Syntax Example: Definition Part of the Handler Method for the Drop Event**

In the above example, you must define a handler method for the drop event of the simple tree model.

In the implementation part, you will have to specify **on which** flight connection node the user dropped the customer data record. The event will provide an appropriate parameter that must be imported.

The “Drag&Drop service instance” provides the reference to the data transport instance. The event will provide a reference to this “service instance” using an appropriate parameter that must be imported.

The technical name of the drop event and the interface parameters are independent of the control class. Thus for this step in the implementation, it is **absolutely essential** to take the names from the online documentation or from the Class Builder.

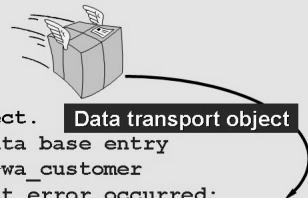
## Syntax Example: Implementation Part of the Handler Method for the Drop Event



```

CLASS lcl_event_handler IMPLEMENTATION.
...
METHOD handle_drop.
  DATA:
    l_ref_booking TYPE REF TO lcl_booking,
    l_wa_newnode  TYPE treemsnadt.

    IF drag_drop_object->flavor = 'BOOK'.
      TRY.
        l_ref_booking ?= drag_drop_object->object.
      *   ... create new booking node and modify data base entry
      *   ... from l_wa_newnode and l_ref_booking->wa_customer
        CATCH CX_SY_MOVE_CAST_ERROR. " move cast error occurred:
          CALL METHOD drag_drop_object->abort.
      ENDTRY.
    ELSE. " other flavor:
      ...
    ENDIF.
  ENDMETHOD.
...
ENDCLASS.
```



**Figure 199: Syntax Example: Implementation Part of the Handler Method for the Drop Event**

In the above example, the handler method for the drag event of the SAP Grid Control must perform the following task: If the Drag&Drop procedure in question is a “Booking” event, add an appropriate node to the flight booking tree. To do this, you must read the application data from the data transport instance.

Define a local reference (**l\_ref\_booking** in the above example) to an instance of your data transport class and a work area for the node entry (**l\_wa\_newnode**).

The “Drag&Drop service instance” provides the local reference to the **object** reference. Once again you can access the data transport instance that you created and filled in the first method and to other attributes - such as the flavor.

Use the application data to generate a new node in the flight bookings tree and if necessary to update the appropriate transparent table.

Use the ABORT method of the “Drag&Drop service instance” if you want to Drag&Drop procedure before it has been completed.

Here you need to make the assignment using a **widening cast** ("?="). This means that the static type check will be omitted. For this reason, you should enclose your assignment in a **CATCH...ENDCATCH** clause **on principle**. You can then catch the runtime error, **move\_cast\_error**, that may occur.

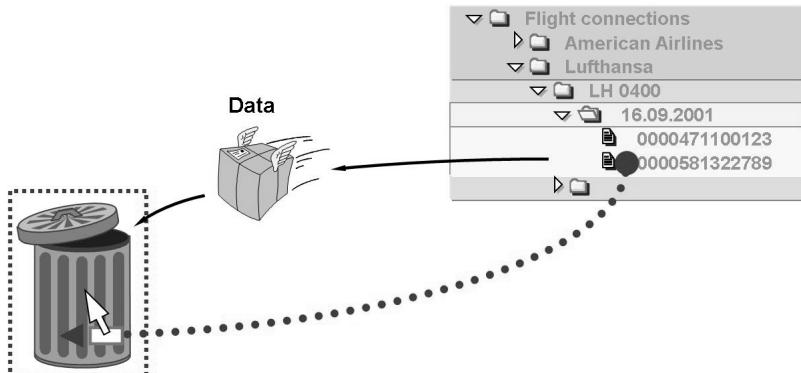
→ **Note:** From SAP R/3 Basis Release 6.10 onwards, use a **TRY-ENDTRY** block instead of a **CATCH-ENDCATCH**. For further information, refer to the keyword documentation for the TRY statement.

## Syntax Example: Definition Part of the Handler Method for the Drop-Complete Event



```
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
  ...
  CLASS-METHODS handle_drop_complete
    FOR EVENT drop_complete
    OF cl_simple_tree_model
    IMPORTING node_key drag_drop_object.
  ENDCLASS.
```

Node ID      Reference to the "Drag&Drop service instance"



**Figure 200: Syntax Example: Definition Part of the Handler Method for the Drop-Complete Event**

After the drop event of the target control the system raises the **drop-complete event** of the **source control**. This “Drag&Drop service instance” is also available for this event. Again, you can program functions in an appropriate handler method that you want to execute after the Drag&Drop procedure has been completed.

The above example shows a typical example: Cancellation. After the “Move” procedure, the source control must be updated. Thus, in this case, the booking node must be removed from the tree and the follow-up activities carried out after the node has been placed in the trash can.

To complete the above example, you simply define another handler method for the drop-complete event of the **simple tree model**. A handler method for the drop event of the picture control is not necessary in this example.

Implement a handler method for the drag event similar to that for the SAP Grid Control. Take the name for the event and interface parameters from the Class Builder or the online documentation.

In the implementation section of the handler method for the drop-complete event you must specify which booking node the user wants to “remove”. The event will provide an appropriate parameter that must be imported.

You must also get the flavor used from the “Drag&Drop service instance”. The event will provide a reference to this “service instance” using an appropriate parameter that must be imported.

## Syntax Example: Implementation Part of the Handler Method for the Drop-Complete Event



```
CLASS lcl_event_handler IMPLEMENTATION.  
  ...  
  METHOD handle_drop_complete.  
    IF drag_drop_object->flavor = 'CANC'.  
      * ... delete booking node with imported node key  
      * ... get application from imported node key and update data base  
      ELSE. " other flavor:  
        CALL METHOD drag_drop_object->abort.  
      ENDIF.  
    ENDMETHOD.  
  ENDCLASS.
```

You do not need to export the data transport object in this case

**Figure 201: Syntax Example: Implementation Part of the Handler Method for the Drop-Complete Event**

In the above example, the handler method for the drop-complete event of the simple tree model must perform the following task: If the Drag&Drop procedure in question is a “Cancellation” event, delete the appropriate node from the flight booking tree.

Again, you must export the application data from the “Drag&Drop service instance” using widening cast, similar to the handler method for the drop event of the simple tree model.

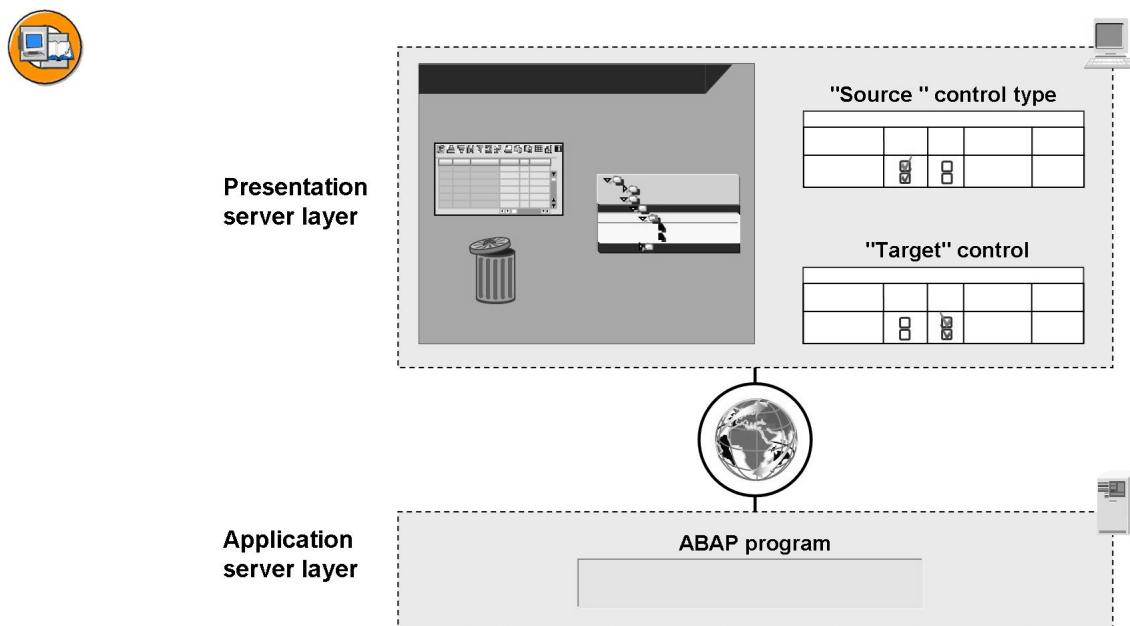
For the above example, you need only the node ID. You simply need to pass the pointer to an “empty” data transport instance in the handler method for the drag event of the simple tree model to the “Drag&Drop service instance”. This

will make the node ID of the “dragged” booking node available **as a parameter**. This makes implementation much easier in this case: Remove this node with the imported ID from the flight bookings tree.

 **Note:** To comply with the usual ergonomic rules, offer the user an Undo function for each Drag&Drop procedure, at least for Move procedures.

The example used in this unit is implemented in the program SAPBC412\_DNDD\_COPY\_DELETE in the package BC412.

## Drag&Drop in the WAN Environment: Solution Optimized for Performance



**Figure 202: Drag&Drop in the WAN Environment: Solution Optimized for Performance**

Since the control framework is generally suited to working in a **Wide Area Network** environment, you only need to take the following into account when implementing Drag&Drop functions.

The entire Drag&Drop procedure is executed on the presentation server – that is, the system checks there that the controls involved have been assigned to the identically-named flavors. To do this, the system must pass **all** control type tables to the presentation server.

Few tables with many flavors attached to each are passed more quickly than many tables, each with few flavors.

One possible alternative to the procedure shown in this unit might have been to define four control types, each with just one flavor. This solution might have been easier to understand and implement. There would have been no need to query the flavor used in the handler methods. You could have allowed more mouse operations for the same procedure. The procedures could have identified simply from the controls involved.

However, this conceivable alternative would not have been as good from the viewpoint described above. Thus, the example shown in this unit represents the solution for **optimal runtime performance**.

**Summary:** For each control (sub-)object that will behave in the same way, user the same instance of the class CL\_DRAGDROP (control type table).

# Exercise 18: Drag&Drop Functions

## Exercise Objectives

After completing this exercise, you will be able to:

- Implement a Drag&Drop Copy function in a tree control.

## Business Example

Allow the user to copy single flight nodes from a flight connection tree to a *Favorites* folder using Drag&Drop.

**Program:** ZBC412\_##\_DND\_EX1

**Template:** SAPBC412\_TRMS\_EXERCISE\_1

**Model solution:** SAPBC412\_DNDS\_EXERCISE\_1

where ## is the group number.

### Task 1:

Copy the template.

1. Copy your solution from the first exercise from the previous chapter (ZBC412\_##\_TRM\_EX1) or the appropriate model solution (SAPBC412\_TRMS\_EXERCISE\_1) assigning it the new name **ZBC412\_##\_DND\_EX3**. Get to know how your copy works.

### Task 2:

Create a node for the user's favorites.

1. On the root node level of the hierarchy, create a node for the user's favorites. Extend the add\_nodes subroutine accordingly.

### Task 3:

Create two behavior-type tables (control types) each of which containing one behavior type (Flavor) – one should contain the source flavor, the other should contain the target flavor.

1. Create another PBO module where you “initialize” the Drag&Drop (suggested name: init\_dragdrop\_processing\_0100) and call it from the flow logic of screen 0100.
2. Create reference variables for the two control type instances (We suggest you use the names ref\_flav\_src and ref\_flav\_trg).

*Continued on next page*

3. Create both instances in the `init_dragdrop_processing_0100` module.
4. Add a “source flavor” with the effect “Copy” and a “target flavor” with the effect “Copy”. Use the `add` method.

### Task 4:

Assign the two behavior type tables (control types) to the tree control nodes. Make sure that users can only drag flight nodes (but not drop on them), while they can only drop objects onto the *Favorites* node.

1. Create two global data objects of type `i` for the control type identifiers (we suggest you use the name `handle_flav_src` and `handle_flav_trg`).
2. Get the identifiers from the behavior type tables and store them in the two data objects. To do this, call the `get_handle` method in the `init_dragdrop_processing_0100` module.
3. Fill the `dragdropid` parameter with the identifiers in the appropriate method calls (in the `add_nodes` subroutine).

### Task 5:

Define a local class for the data transport.

1. Define a local class (suggested name: `lcl_flightdate`) for the data transport between the source and target nodes. In the public section of this class, define a work area as an instance attribute to contain the node entry (suggested name: `wa_node`, global type: `treemsnodi`).

### Task 6:

Implement a handler method for the event that reads the data from the “dragged” node from the global node table and copies it into the data transport instance.

1. To do this, create a local class (suggested name: `lcl_event_handler`).
2. In the public section of the class `lcl_event_handler`, define a static handler method (suggested name: `handle_drag`) for the drag event of the class `cl_simple_tree_model`. Find out about the interface of the event and import the necessary parameters.
3. In the implementation part of the method, define a local reference variable (suggested name: `l_ref_flightdate`) to your local class for the data transport.

*Continued on next page*

Create an instance of this class. Read the properties of the “dragged” node (import parameter `node_key`) by calling the `node_get_properties` method of the tree model instance that raised the event. To do this you must have imported the reference `sender` to this instance.



**Caution:** The method `node_get_properties` does **not** return the node ID. Therefore, you must also copy the node ID to the target structure.

Pass the contents of the target structure to the attribute of your transport instance.

Pass the reference to the transport instance to the `object` attribute of the “Drag&Drop service instance”. To do this, you must have imported the reference to the “Drag&Drop service instance”.

4. Register the handler method with the tree control object in the PBO module `init_tree_processing_0100`.

## Task 7:

Implement a handler method for the drop event, which reads the data of the “dragged” node from the data transport instance and, using this data, inserts another node in the *Favorites* folder.

1. Define a static handler method (suggested name: `handle_drag`) for the drop event of the class `cl_simple_tree_model`. Find out about the interface of the event and import the necessary parameters.
2. In the implementation part of the method, define a local reference (suggested name: `l_ref_flightdate`) to your local class for the data transport.

Also define a local work area for the node entry (suggested name: `l_wa_newnode`).

Pass the reference `object` of the “Drag&Drop service instance” to the local reference. To do this, you must have imported the reference to the “Drag&Drop service instance”. Enclose this assignment in a `CATCH...ENDCATCH` block and catch the possible runtime error, `move_cast_error`.

Fill the fields of the local work area for the new node entry with the contents of your transport object. Form the new node ID by combining the ID of the “dragged” node with the system date and time. This will ensure that your node IDs are unique. As your relationship node, choose the one on which the dragged node was dropped (import parameter `node_key`).



**Note:** For the purposes of this exercise, simply use the new node ID for the text entry.

*Continued on next page*

Add the new node to the tree control using the `add_node` method.

Finally, expand the *Favorites* node by calling the `expand_node` method of the tree model instance.

3. Register the tree control object with the handler method object in the PBO module `init_tree_processing_0100`.

## Solution 18: Drag&Drop Functions

### Task 1:

Copy the template.

1. Copy your solution from the first exercise from the previous chapter (ZBC412\_##\_TRM\_EX1) or the appropriate model solution (SAPBC412\_TRMS\_EXERCISE\_1) assigning it the new name **ZBC412\_##\_DND\_EX3**. Get to know how your copy works.

a) -

### Task 2:

Create a node for the user's favorites.

1. On the root node level of the hierarchy, create a node for the user's favorites.  
Extend the `add_nodes` subroutine accordingly.
- a)

```

CLEAR l_wa_node.
l_wa_node-text      = text-fav.
CALL METHOD l_ref_tree_model->add_node
EXPORTING
  node_key           = 'FAV'
  *     RELATIVE_NODE_KEY    =
  *     RELATIONSHIP        =
  isfolder           = 'X'
  text               = l_wa_node-text
  expander          = 'X'
  drag_drop_id      = handle_flav_trg
EXCEPTIONS
  OTHERS             = 5.
IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

```

*Continued on next page*

### Task 3:

Create two behavior-type tables (control types) each of which containing one behavior type (Flavor) – one should contain the source flavor, the other should contain the target flavor.

1. Create another PBO module where you “initialize” the Drag&Drop (suggested name: init\_dragdrop\_processing\_0100) and call it from the flow logic of screen 0100.

a)

```
PROCESS BEFORE OUTPUT.  
...  
MODULE init_dragdrop_processing_0100.  
...
```

2. Create reference variables for the two control type instances (We suggest you use the names ref\_flav\_src and ref\_flav\_trg).

a)

```
DATA:  
...  
ref_flav_src TYPE REF TO cl_dragdrop,  
ref_flav_trg TYPE REF TO cl_dragdrop.  
...
```

*Continued on next page*

3. Create both instances in the `init_dragdrop_processing_0100` module.

a)

```

MODULE init_dragdrop_processing_0100 OUTPUT.
  IF ref_flav_src IS INITIAL.
    CREATE OBJECT:
      ref_flav_src,
      ref_flav_trg.
    CALL METHOD ref_flav_src->add
      EXPORTING
        flavor          = 'FAV'
        dragsrc         = 'X'
        droptarget      = space
        effect          = cl_dragdrop=>copy
    EXCEPTIONS
      OTHERS          = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.
  CALL METHOD ref_flav_trg->add
    EXPORTING
      flavor          = 'FAV'
      dragsrc         = space
      droptarget      = 'X'
      effect          = cl_dragdrop=>copy
    EXCEPTIONS
      OTHERS          = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.
ENDIF.
ENDMODULE." init_dragdrop_processing_0100  OUTPUT

```

4. Add a “source flavor” with the effect “Copy” and a “target flavor” with the effect “Copy”. Use the `add` method.

a) –

*Continued on next page*

## Task 4:

Assign the two behavior type tables (control types) to the tree control nodes. Make sure that users can only drag flight nodes (but not drop on them), while they can only drop objects onto the *Favorites* node.

1. Create two global data objects of type i for the control type identifiers (we suggest you use the name handle\_flav\_src and handle\_flav\_trg).

a)

```
DATA:  
...  
    handle_flav_src TYPE i,  
    handle_flav_trg TYPE i.
```

2. Get the identifiers from the behavior type tables and store them in the two data objects. To do this, call the get\_handle method in the init\_dragdrop\_processing\_0100 module.

a)

```
MODULE init_dragdrop_processing_0100 OUTPUT.  
...  
    CALL METHOD ref_flav_src->get_handle  
        IMPORTING  
            handle      = handle_flav_src  
        EXCEPTIONS  
            OTHERS      = 1.  
        IF sy-subrc <> 0.  
            MESSAGE a012.  
        ENDIF.  
    CALL METHOD ref_flav_trg->get_handle  
        IMPORTING  
            handle      = handle_flav_trg  
        EXCEPTIONS  
            OTHERS      = 1.  
        IF sy-subrc <> 0.  
            MESSAGE a012.  
        ENDIF.  
...  
ENDMODULE.  " init_dragdrop_processing_0100  OUTPUT
```

*Continued on next page*

3. Fill the dragdropid parameter with the identifiers in the appropriate method calls (in the add\_nodes subroutine).

a)

```

FORM add_nodes USING l_ref_tree_model TYPE REF TO cl_simple_tree_model.
...
CLEAR l_wa_node.
l_wa_node-text      = text-fav.
CALL METHOD l_ref_tree_model->add_node
      EXPORTING
        node_key           = 'FAV'
      *     RELATIVE_NODE_KEY    =
      *     RELATIONSHIP       =
        isfolder            = 'X'
        text                = l_wa_node-text
        expander            = 'X'
        drag_drop_id        = handle_flav_trg
      EXCEPTIONS
        OTHERS              = 5.
      IF sy-subrc <> 0.
        MESSAGE a012.
      ENDIF.
...
ENDFORM.          " ADD_NODES

```

## Task 5:

Define a local class for the data transport.

1. Define a local class (suggested name: lcl\_flightdate) for the data transport between the source and target nodes. In the public section of this class, define a work area as an instance attribute to contain the node entry (suggested name: wa\_node, global type: treemsnadt).

a)

```

CLASS lcl_flightdate DEFINITION.
  PUBLIC SECTION.
    DATA wa_node TYPE treemsnadt.
  ENDCLASS.

```

*Continued on next page*

## Task 6:

Implement a handler method for the event that reads the data from the “dragged” node from the global node table and copies it into the data transport instance.

1. To do this, create a local class (suggested name: lcl\_event\_handler).
  - a)
2. In the public section of the class lcl\_event\_handler, define a static handler method (suggested name: handle\_drag) for the drag event of the class cl\_simple\_tree\_model. Find out about the interface of the event and import the necessary parameters.
  - a)

```
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS handle_drag FOR EVENT drag
      OF cl_simple_tree_model
      IMPORTING
        sender
        node_key
        drag_drop_object.
  ENDCLASS.
```

3. In the implementation part of the method, define a local reference variable (suggested name: l\_ref\_flightdate) to your local class for the data transport.

Create an instance of this class. Read the properties of the “dragged” node (import parameter node\_key) by calling the node\_get\_properties method of the tree model instance that raised the event. To do this you must have imported the reference sender to this instance.



**Caution:** The method node\_get\_properties does **not** return the node ID. Therefore, you must also copy the node ID to the target structure.

Pass the contents of the target structure to the attribute of your transport instance.

*Continued on next page*

Pass the reference to the transport instance to the `object` attribute of the “Drag&Drop service instance”. To do this, you must have imported the reference to the “Drag&Drop service instance”.

a)

```
CLASS lcl_event_handler IMPLEMENTATION.
METHOD handle_drag.
DATA l_ref_flightdate TYPE REF TO lcl_flightdate.
CREATE OBJECT l_ref_flightdate.
CALL METHOD sender->node_get_properties
      EXPORTING
          node_key      = node_key
      IMPORTING
          properties    = l_ref_flightdate->wa_node
      EXCEPTIONS
          OTHERS        = 2.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
l_ref_flightdate->wa_node-node_key = node_key.
drag_drop_object->object = l_ref_flightdate.
ENDMETHOD.
ENDCLASS.
```

4. Register the handler method with the tree control object in the PBO module `init_tree_processing_0100`.

a)

```
SET HANDLER:
  lcl_event_handler->handle_drag FOR ref_tree_model.
```

*Continued on next page*

## Task 7:

Implement a handler method for the drop event, which reads the data of the “dragged” node from the data transport instance and, using this data, inserts another node in the *Favorites* folder.

1. Define a static handler method (suggested name: `handle_drag`) for the drop event of the class `cl_simple_tree_model`. Find out about the interface of the event and import the necessary parameters.
  - a)

```
CLASS lcl_event_handler DEFINITION.  
  PUBLIC SECTION.  
    ...  
    CLASS-METHODS handle_drop FOR EVENT drop  
      OF cl_simple_tree_model  
      IMPORTING  
        node_key  
        drag_drop_object.  
    ENDCLASS.
```

2. In the implementation part of the method, define a local reference (suggested name: `l_ref_flightdate`) to your local class for the data transport. Also define a local work area for the node entry (suggested name: `l_wa_newnode`).

Pass the reference object of the “Drag&Drop service instance” to the local reference. To do this, you must have imported the reference to the “Drag&Drop service instance”. Enclose this assignment in a `CATCH...ENDCATCH` block and catch the possible runtime error, `move_cast_error`.

Fill the fields of the local work area for the new node entry with the contents of your transport object. Form the new node ID by combining the ID of the “dragged” node with the system date and time. This will ensure that your node IDs are unique. As your relationship node, choose the one on which the dragged node was dropped (import parameter `node_key`).

 **Note:** For the purposes of this exercise, simply use the new node ID for the text entry.

Add the new node to the tree control using the `add_node` method.

*Continued on next page*

Finally, expand the *Favorites* node by calling the `expand_node` method of the tree model instance.

a)

```

METHOD handle_drop.
DATA:
  l_ref_flightdate TYPE REF TO lcl_flightdate,
  l_wa_newnode TYPE treemsnodi.
CATCH SYSTEM-EXCEPTIONS move_cast_error = 1.
  l_ref_flightdate ?= drag_drop_object->object.
ENDCATCH.
IF sy-subrc = 0.
  l_wa_newnode = l_ref_flightdate->wa_node.
  CONCATENATE text-fli l_wa_newnode-node_key
    INTO l_wa_newnode-text
    SEPARATED BY space.
  CONCATENATE sy-datum sy-uzeit
    l_wa_newnode-node_key
    INTO l_wa_newnode-node_key
    SEPARATED BY space.
  CALL METHOD ref_tree_model->add_node
    EXPORTING
      node_key          = l_wa_newnode-node_key
      relative_node_key = node_key
      relationship      = cl_simple_tree_model
                          ->relat_last_child
      isfolder          = space
      text              = l_wa_newnode-text
    EXCEPTIONS
      OTHERS            = 1.
  IF sy-subrc <> 0.
    CALL METHOD drag_drop_object->abort.
  ENDIF.
  CALL METHOD ref_tree_model->expand_node
    EXPORTING
      node_key = 'FAV'.
ELSE.
  CALL METHOD drag_drop_object->abort.
ENDIF.
ENDMETHOD.
ENDCLASS.

```

3. Register the tree control object with the handler method object in the PBO module `init_tree_processing_0100`.

*Continued on next page*

a)

```
SET HANDLER:
...
lcl_event_handler=>handle_drop FOR ref_tree_model.
```

## Result

The complete source code for the model solution is shown below.

### Screen flow logic

#### Screen 100

```
PROCESS BEFORE OUTPUT.
MODULE status_0100.
MODULE init_container_processing_0100.
MODULE init_dragdrop_processing_0100.
MODULE init_tree_processing_0100.

PROCESS AFTER INPUT.
MODULE exit_command_0100 AT EXIT-COMMAND.
MODULE copy_ok_code.
MODULE user_command_0100.
```

### ABAP program

#### *Data Declarations*

```
REPORT sapbc412_dnds_exercise_1 MESSAGE-ID bc412.

DATA:
* screen-specific:
ok_code TYPE sy-ucomm,
copy_ok LIKE ok_code,

* application data:
it_scarr TYPE SORTED TABLE OF scarr
    WITH UNIQUE KEY carrid,
it_spfli TYPE SORTED TABLE OF spfli
    WITH UNIQUE KEY carrid connid,
it_sflight TYPE SORTED TABLE OF sflight
    WITH UNIQUE KEY carrid connid fldate,
wa_scarr LIKE LINE OF it_scarr,
wa_spfli LIKE LINE OF it_spfli,
```

*Continued on next page*

```

        wa_sflight LIKE LINE OF it_sflight,

* container:
ref_cont_left    TYPE REF TO cl_gui_docking_container,

* content:
ref_tree_model TYPE REF TO cl_simple_tree_model,

* drag&drop-specific:
ref_flav_src TYPE REF TO cl_dragdrop,
ref_flav_trg TYPE REF TO cl_dragdrop,

handle_flav_src TYPE i,
handle_flav_trg TYPE i.

```

### *Local Classes*

```

*-----*
*      CLASS lcl_flightdate DEFINITION
*-----*
*      contains only instance attributes for data transportation*
*      between drag&drop objects
*-----*
CLASS lcl_flightdate DEFINITION.
PUBLIC SECTION.
DATA wa_node TYPE treemsnodi.
ENDCLASS.

*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
*      contains event handler methods for drag&drop processing*
*-----*
CLASS lcl_event_handler DEFINITION.
PUBLIC SECTION.

CLASS-METHODS handle_drag FOR EVENT drag
          OF cl_simple_tree_model
IMPORTING
          sender
          node_key
          drag_drop_object.

```

*Continued on next page*

```

CLASS-METHODS handle_drop FOR EVENT drop
    OF cl_simple_tree_model
    IMPORTING
        node_key
        drag_drop_object.
ENDCLASS.

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

METHOD handle_drag.
    DATA l_ref_flightdate TYPE REF TO lcl_flightdate.

    CREATE OBJECT l_ref_flightdate.
    CALL METHOD sender->node_get_properties
        EXPORTING
            node_key      = node_key
        IMPORTING
            properties   = l_ref_flightdate->wa_node
    EXCEPTIONS
        OTHERS        = 2

    IF sy-subrc <> 0.
        MESSAGE a012.
    ENDIF.
    l_ref_flightdate->wa_node-node_key = node_key.
    drag_drop_object->object = l_ref_flightdate.
ENDMETHOD.

*-----*

METHOD handle_drop.
    DATA:
        l_ref_flightdate TYPE REF TO lcl_flightdate,
        l_wa_newnode TYPE treemsnodi.

    CATCH SYSTEM-EXCEPTIONS move_cast_error = 1.
        l_ref_flightdate ?= drag_drop_object->object.
    ENDCATCH.
    IF sy-subrc = 0.
        l_wa_newnode = l_ref_flightdate->wa_node.

```

*Continued on next page*

```

CONCATENATE text-fli l_wa_newnode-node_key
    INTO l_wa_newnode-text
    SEPARATED BY space.
CONCATENATE sy-datum sy-uzzeit
    l_wa_newnode-node_key
    INTO l_wa_newnode-node_key
    SEPARATED BY space.

CALL METHOD ref_tree_model->add_node
    EXPORTING
        node_key          = l_wa_newnode-node_key
        relative_node_key = node_key
        relationship      = cl_simple_tree_model
                            =>relat_last_child
        isfolder          = space
        text              = l_wa_newnode-text
    EXCEPTIONS
        OTHERS             = 1.
    IF sy-subrc <> 0.
        CALL METHOD drag_drop_object->abort.
    ENDIF.

CALL METHOD ref_tree_model->expand_node
    EXPORTING
        node_key = 'FAV'.
ELSE.
    CALL METHOD drag_drop_object->abort.
ENDIF.
ENDMETHOD.

ENDCLASS.

```

### *ABAP Program: Event Blocks*

```

START-OF-SELECTION.
* get application data:
SELECT * FROM scarr
    INTO TABLE it_scarr.
IF sy-subrc <> 0.
    MESSAGE a060.
ENDIF.

SELECT * FROM spfli
    INTO TABLE it_spfli.

```

*Continued on next page*

```

IF sy-subrc <> 0.
MESSAGE a060.
ENDIF.

SELECT * FROM sflight
INTO TABLE it_sflight.
IF sy-subrc <> 0.
MESSAGE a060.
ENDIF.

CALL SCREEN 100.

```

### ***Modules***

```

*&-----*
*&      Module status_0100 OUTPUT
*&-----*
*      Set GUI title and GUI status for screen 100.
*-----*
MODULE status_0100 OUTPUT.
SET PF-STATUS 'NORM_0100'.
SET TITLEBAR 'TITLE_1'.
ENDMODULE.                                " status_0100 OUTPUT

*&-----*
*&      Module init_container_processing_0100 OUTPUT
*&-----*
*      Start control handling,
*      create container object
*-----*
MODULE init_container_processing_0100 OUTPUT.
IF ref_cont_left IS INITIAL.

CREATE OBJECT ref_cont_left
EXPORTING
ratio          = 35
EXCEPTIONS
others         = 1.
IF sy-subrc NE 0.
MESSAGE a010.
ENDIF.

ENDIF.

```

*Continued on next page*

```

ENDMODULE.          " init_container_processing_0100 OUTPUT

*&-----*
*&      Module  init_dragdrop_processing_0100  OUTPUT
*&-----*
*      build up flavor tables and get handles to them
*-----*
MODULE init_dragdrop_processing_0100 OUTPUT.
IF ref_flav_src IS INITIAL.

CREATE OBJECT:
  ref_flav_src,
  ref_flav_trg.

CALL METHOD ref_flav_src->add
  EXPORTING
    flavor      = 'FAV'
    dragsrc     = 'X'
    droptarget   = space
    effect       = cl_dragdrop=>copy
  EXCEPTIONS
    OTHERS      = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

CALL METHOD ref_flav_trg->add
  EXPORTING
    flavor      = 'FAV'
    dragsrc     = space
    droptarget   = 'X'
    effect       = cl_dragdrop=>copy
  EXCEPTIONS
    OTHERS      = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

CALL METHOD ref_flav_src->get_handle
  IMPORTING
    handle      = handle_flav_src
  EXCEPTIONS
    OTHERS      = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

```

*Continued on next page*

```

ENDIF.

CALL METHOD ref_flav_trg->get_handle
  IMPORTING
    handle      = handle_flav_trg
  EXCEPTIONS
    OTHERS      = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

ENDIF.
ENDMODULE.      " init_dragdrop_processing_0100  OUTPUT

*&-----*
*&      Module  init_tree_processing_0100  OUTPUT
*&-----*
*   create tree object, link to container and fill with data
*-----*
MODULE init_tree_processing_0100 OUTPUT.
  IF ref_tree_model IS INITIAL.

    CREATE OBJECT ref_tree_model
      EXPORTING
        node_selection_mode = cl_simple_tree_model
                               =>node_sel_mode_single
      EXCEPTIONS
        others            = 1.
  IF sy-subrc <> 0.
    MESSAGE a043.
  ENDIF.

    CALL METHOD ref_tree_model->create_tree_control
      EXPORTING
        parent           = ref_cont_left
      EXCEPTIONS
        OTHERS          = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.

    PERFORM add_nodes USING ref_tree_model.

    SET HANDLER:
      lcl_event_handler=>handle_drag FOR ref_tree_model,

```

*Continued on next page*

```

lcl_event_handler=>handle_drop FOR ref_tree_model.

ENDIF.

ENDMODULE.          " init_tree_processing_0100  OUTPUT

*&-----*
*&      Module  EXIT_COMMAND_0100  INPUT
*&-----*
* Implementation of user commands of type 'E' for screen 100.
*-----*
MODULE exit_command_0100 INPUT.

CASE ok_code.
  WHEN 'CANCEL'.           " Cancel screen processing
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
  WHEN 'EXIT'.              " Exit program
    PERFORM free_control_ressources.
    LEAVE PROGRAM.
  WHEN OTHERS.
ENDCASE.

ENDMODULE.          " EXIT_COMMAND_0100  INPUT

*&-----*
*&      Module  COPY_OK_CODE  INPUT
*&-----*
*      Save the current user command in order to
*      prevent unintended field transport for the screen
*      field ok_code for next screen processing (ENTER)
*-----*
MODULE copy_ok_code INPUT.
  copy_ok = ok_code.
  CLEAR ok_code.
ENDMODULE.          " COPY_OK_CODE  INPUT

*&-----*
*&      Module  USER_COMMAND_0100  INPUT
*&-----*
* Implementation of user commands of type ' ' for screen 100.
*-----*
MODULE user_command_0100 INPUT.

CASE copy_ok.
  WHEN 'BACK'.             " Go back to program
    PERFORM free_control_ressources.
    LEAVE TO SCREEN 0.
  WHEN OTHERS.

```

*Continued on next page*

```

ENDCASE.
ENDMODULE.          " USER_COMMAND_0100  INPUT

```

### *Subroutines*

```

*&-----*
*&      Form  FREE_CONTROL_RESSOURCES
*&-----*
*      free control ressources on the presentation server
*      free all reference variables (ABAP object)
*      -> garbage collector
*-----*
*      no interface
*-----*
FORM free_control_ressources.
  CALL METHOD:
    ref_cont_left->free.
  FREE:
    ref_tree_model,
    ref_cont_left,
    ref_flav_src,
    ref_flav_trg.
ENDFORM.          " FREE_CONTROL_RESSOURCES

*-----*
*      FORM ADD_NODES
*-----*
*      build up a hierarchy consisting of
*      carriers, connections and flight dates
*-----*
*  --> L_REF_TREE_MODEL
*-----*
FORM add_nodes USING l_ref_tree_model
  TYPE REF TO cl_simple_tree_model.

DATA:
  l_wa_node TYPE treemsnodi,
  date_text(10) TYPE c.

  l_wa_node-text = text-car.
  CALL METHOD l_ref_tree_model->add_node
    EXPORTING
      node_key           = 'ROOT'
*      RELATIVE_NODE_KEY = 

```

*Continued on next page*

```

*      RELATIONSHIP          =
isfolder           = 'X'
text               = l_wa_node-text
expander           = 'X'
EXCEPTIONS
OTHERS             = 5

.

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

* this node should be a drop-target
CLEAR l_wa_node.
l_wa_node-text     = text-fav.
CALL METHOD l_ref_tree_model->add_node
EXPORTING
node_key           = 'FAV'
*      RELATIVE_NODE_KEY    =
*      RELATIONSHIP         =
isfolder           = 'X'
text               = l_wa_node-text
expander           = 'X'
drag_drop_id       = handle_flav_trg
EXCEPTIONS
OTHERS             = 5

.

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

* scarr-nodes:
LOOP AT it_scarr INTO wa_scarr.
CLEAR l_wa_node.
l_wa_node-node_key = wa_scarr-carrid.
l_wa_node-text     = wa_scarr-carrname.
CALL METHOD l_ref_tree_model->add_node
EXPORTING
node_key           = l_wa_node-node_key
relative_node_key = 'ROOT'
relationship        = cl_simple_tree_model
                   =>relat_last_child
isfolder           = 'X'
text               = l_wa_node-text
expander           = 'X'
EXCEPTIONS

```

*Continued on next page*

```

OTHERS           = 5

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
ENDLOOP.

* spfli-nodes:
LOOP AT it_spfli INTO wa_spfli.
CLEAR l_wa_node.
CONCATENATE wa_spfli-carrid
            wa_spfli-connid
            INTO l_wa_node-node_key
            SEPARATED BY space.
CONCATENATE wa_spfli-carrid
            wa_spfli-connid
            ':'
            wa_spfli-cityfrom
            '->'
            wa_spfli-cityto
            INTO l_wa_node-text
            SEPARATED BY space.
l_wa_node-relatkey = wa_spfli-carrid.
CALL METHOD l_ref_tree_model->add_node
      EXPORTING
        node_key          = l_wa_node-node_key
        relative_node_key = l_wa_node-relatkey
        relationship      = cl_simple_tree_model
                            =>relat_last_child
        isfolder          = 'X'
        text              = l_wa_node-text
        expander          = 'X'
      EXCEPTIONS
        OTHERS           = 5

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
ENDLOOP.

* sflight-nodes:
LOOP AT it_sflight INTO wa_sflight.
CLEAR l_wa_node.
CONCATENATE wa_sflight-carrid
            wa_sflight-connid

```

*Continued on next page*

```
        wa_sflight-fldate
        INTO l_wa_node-node_key
        SEPARATED BY space.

CONCATENATE wa_sflight-carrid
        wa_sflight-connid
        INTO l_wa_node-relatkey
        SEPARATED BY space.

WRITE wa_sflight-fldate TO date_text.
l_wa_node-text = date_text.

* flight date nodes should be able to be dragged:
CALL METHOD l_ref_tree_model->add_node
    EXPORTING
        node_key          = l_wa_node-node_key
        relative_node_key = l_wa_node-relatkey
        relationship      = cl_simple_tree_model
                            =>relat_last_child
        isfolder          = space
        text              = l_wa_node-text
* EXPANDER          =
        drag_drop_id      = handle_flav_src
    EXCEPTIONS
        OTHERS            = 5

.

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

ENDLOOP.

ENDFORM.          " ADD_NODES
```



## Lesson Summary

You should now be able to:

- Name the uses of Drag&Drop functions
- Describe the technical sequence of a Drag&Drop procedure
- Implement copying procedures between different control objects or sub-objects
- Implement move procedures between different control objects or sub-objects



## **Unit Summary**

You should now be able to:

- Name the uses of Drag&Drop functions
- Describe the technical sequence of a Drag&Drop procedure
- Implement copying procedures between different control objects or sub-objects
- Implement move procedures between different control objects or sub-objects



# Unit 9

## Including Different Controls in Complex User Dialogs

### Unit Overview

This unit combines the usage of multiple screen elements and EnjoySAP Controls.



### Unit Objectives

After completing this unit, you will be able to:

- Name the benefits and uses of some classic screen elements
- Name the benefits and uses of some control-based screen elements
- Choose and implement the right classic or control-based dialog programming technique for the task on hand
- Combine classical and control-based screen elements in the same program - that is, implement the interaction between them

### Unit Contents

Lesson: Including Different Controls in Complex User Dialogs.....	496
Exercise 19: Creating a One-Screen Transaction for Booking Flights (OPTIONAL).....	507

## Lesson: Including Different Controls in Complex User Dialogs

### Lesson Overview

This lesson combines the usage of multiple screen elements and EnjoySAP Controls.



### Lesson Objectives

After completing this lesson, you will be able to:

- Name the benefits and uses of some classic screen elements
- Name the benefits and uses of some control-based screen elements
- Choose and implement the right classic or control-based dialog programming technique for the task on hand
- Combine classical and control-based screen elements in the same program - that is, implement the interaction between them

### Business Example

The user wants a single-screen transaction with the following functions:

- Display a set of customer data records in a table. Restrict the set by customer number.
- Book any number of flights for any number of customers from this set
- Undo the last booking.
- Format the additional data for any booking to create an itinerary (booking confirmation).

## An Idea for Implementing Dialog Forms

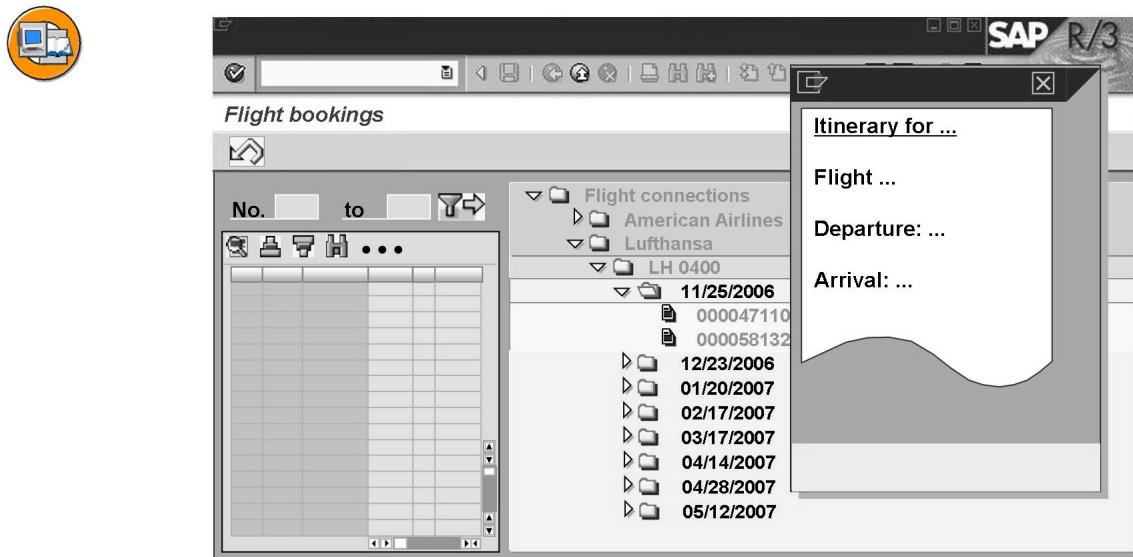


Figure 203: An Idea for Implementing Dialog Forms

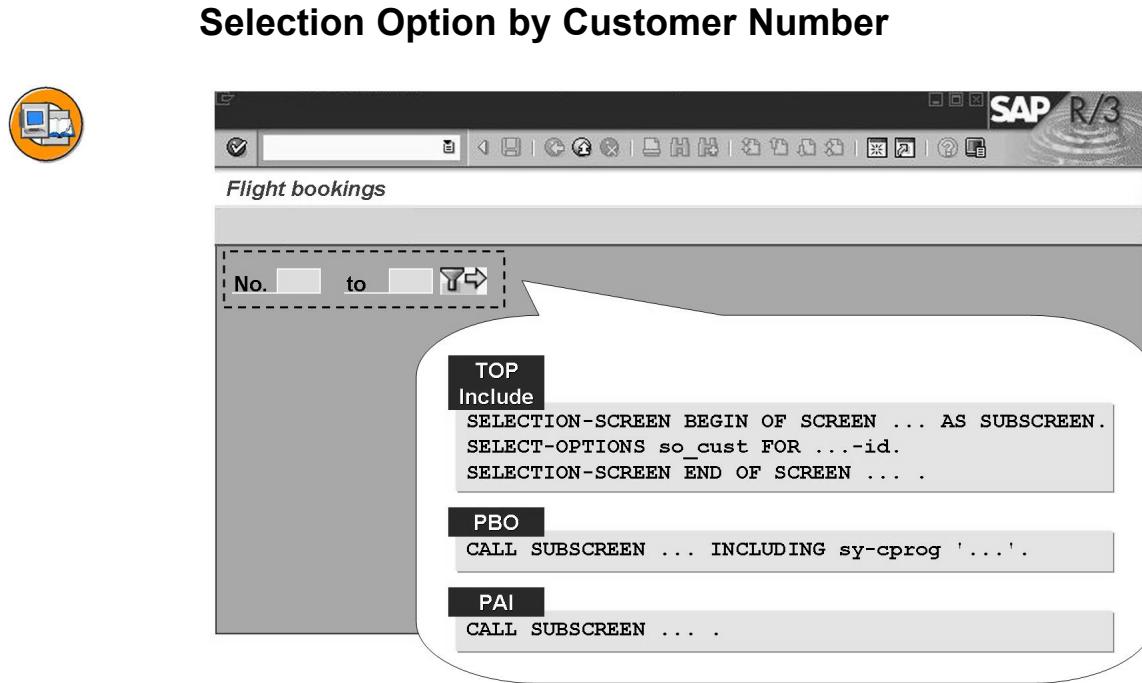
First you must decide what program type to use: We are not talking about a typical “report” here. This means that we do not need the predefined sequence of ABAP events in an executable program. The **module type** suggests itself here. There will be substantially more source code than in previous exercise and demonstration programs. We should split the source code into several Include programs, to make the program easier to maintain and clarify its structure. The system can take over this task, if we comply with the naming convention for customer programs – **SAPMY...** and **SAPMZ....**

It seems appropriate to use two EnjoySAP Controls:

- The **SAP Grid Control** to display the customer data records in a table
- The **tree control** to display all the available flight connections in a hierarchical structure

A classical **modal dialog box** is an appropriate way of outputting the itinerary. We could then use all the formatting options for ABAP lists and the standard list GUI status. The print functions are also automatically made available.

We can implement booking the flights and creating the itinerary using mouse operations, for which the EnjoySAP Controls have appropriate events.



**Figure 204: Selection Option by Customer Number**

Since the EnjoySAP Controls always need to be included in a “Container” screen for technical reasons, it is a good idea to implement the latter functionally.

Apart from using the associated user interface for menus and pushbuttons, we may wish to implement the selection option for customer numbers classically. The ABAP statement SELECT-OPTIONS allows you to implement this requirement in a very simple way. The system automatically processes additional functions like search helps.

Moreover, since EnjoySAP Controls have been developed primarily for **displaying** data, they do not naturally suggest themselves as a solution in this case.

For technical reasons, we can display a selection option of limiting records only by using the “detour” of a Selection Screen as a Subscreen. Thus, we need to declare the top left corner as a subscreen area, define a selection screen as a subscreen and display this selection screen in the subscreen area of the screen.

We can supply default values for the selection option dynamically in the LOAD-OF-PROGRAM event.

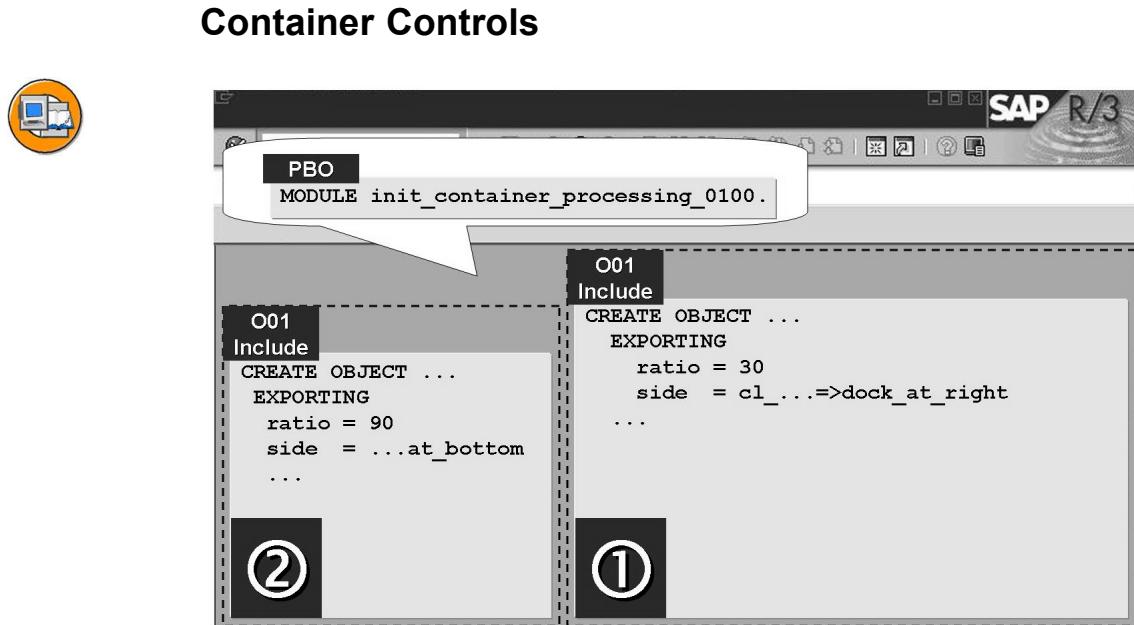


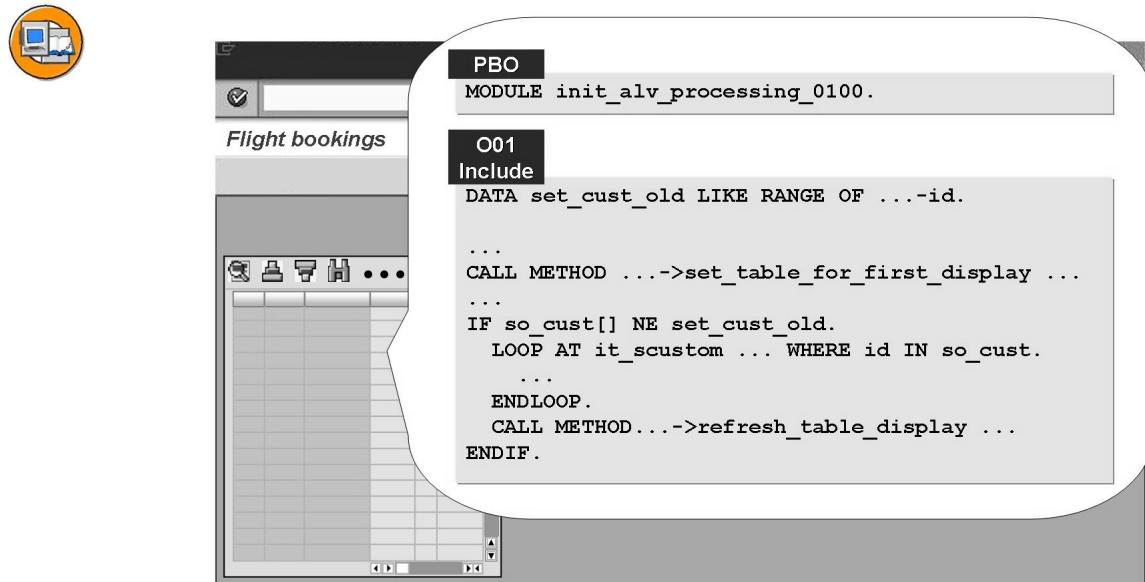
Figure 205: Container Controls

All EnjoySAP Controls require a container control.

It is a good idea to use docking containers here so that users can adjust the frames of each screen area to suit themselves.

We need to choose sensible default settings for the size of each screen area.

## The SAP Grid Control for the Customer Data Records



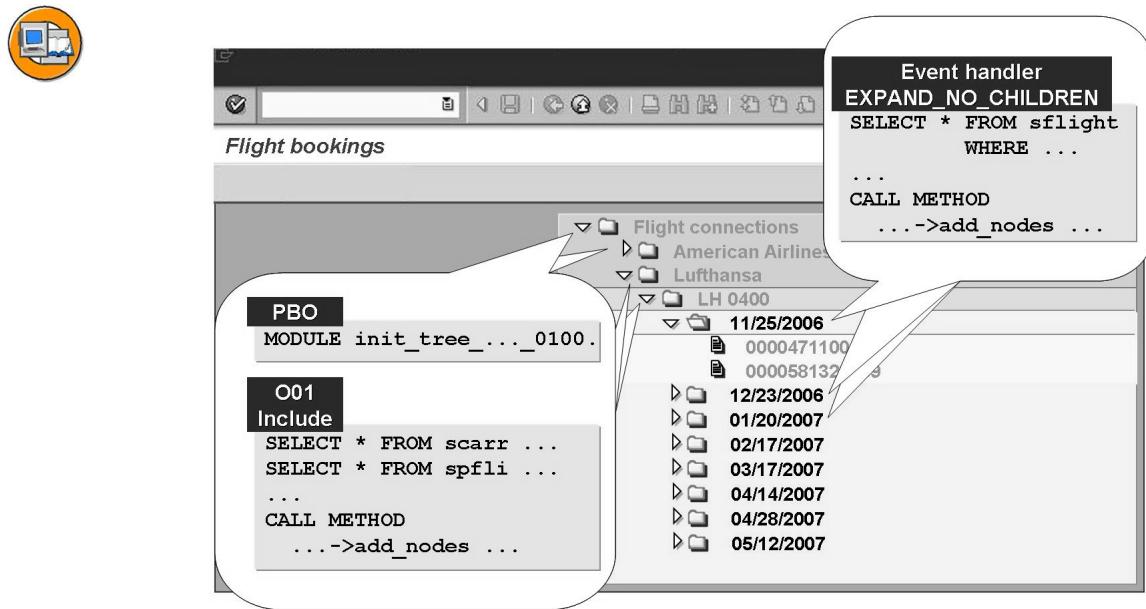
**Figure 206: The SAP Grid Control for the Customer Data Records**

We need to pay special attention to the resources needed at runtime when transporting the data to the SAP Grid Control.

We should not refresh the dataset unless the user has really created a new set records by customer number at runtime.

For comparison purposes, we can create a data object for such a set using the `DATA ... LIKE RANGE OF ...` statement.

## The Simple Tree Model for the Flights



**Figure 207: The Simple Tree Model for the Flights**

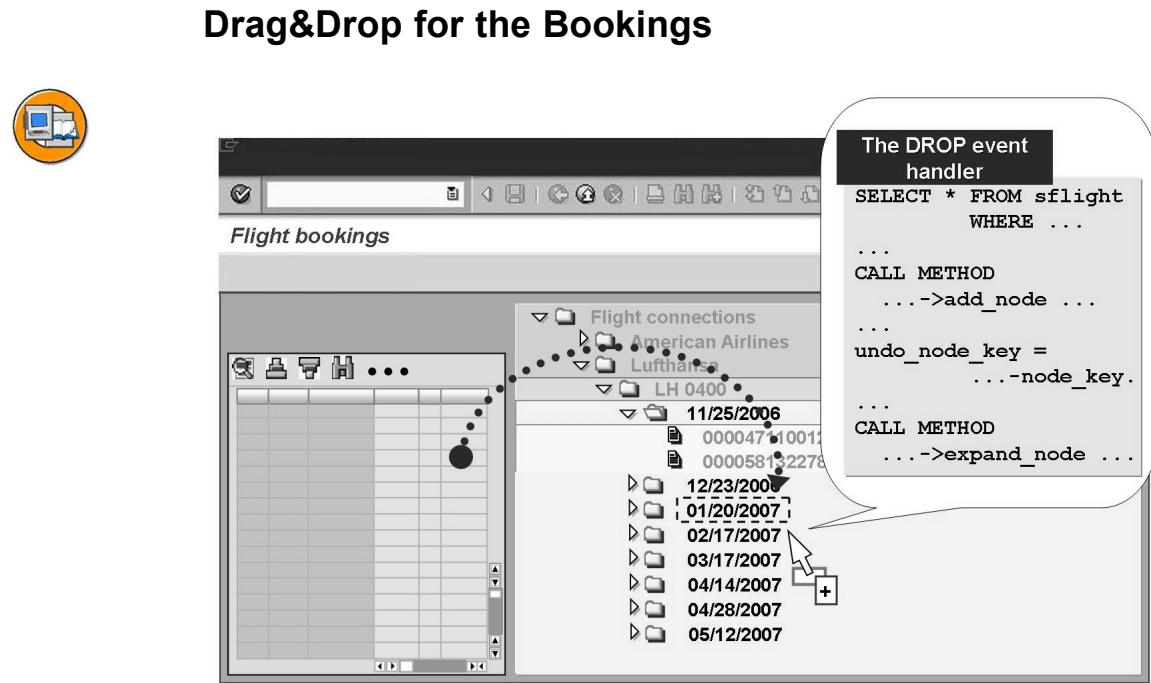
We should also pay special attention to runtime needs when implementing the tree control.

In this case, it is not transporting the data to the presentation server that is runtime-intensive, but rather reading it from the database.

Initially therefore, we should only read the data for a maximum of two hierarchy levels from the database. Again, the LOAD-OF-PROGRAM event block presents itself. Only if the user is interested in a particular flight connection at runtime should we read and buffer the data locally in the program.

We then need to pass the requested data to the tree model instance using events. This instance in turn uses the best (that is, least runtime-intensive) method for passing the data to the presentation server.

When we create the unique node IDs, we should bear in mind that the application keys will later be needed to identify the selected node.



**Figure 208: Drag&Drop for the Bookings**

To implement a booking procedure, we could use the Drag&Drop functions of the EnjoySAP Controls.

One row of the SAP Grid Control can provide the source of a copying procedure, while a flight **connection** node of the tree control provides the target.

As a reaction to the “drop” event, we need to create a booking node and insert it under the appropriate flight connection node.

In doing so, we need to ensure that the ID of this new node is buffered globally in the program, so that we can implement the Undo function later.

In this course, we want to focus on user dialogs. This means that we will not be “generating” unique booking numbers or updating the relevant database tables.

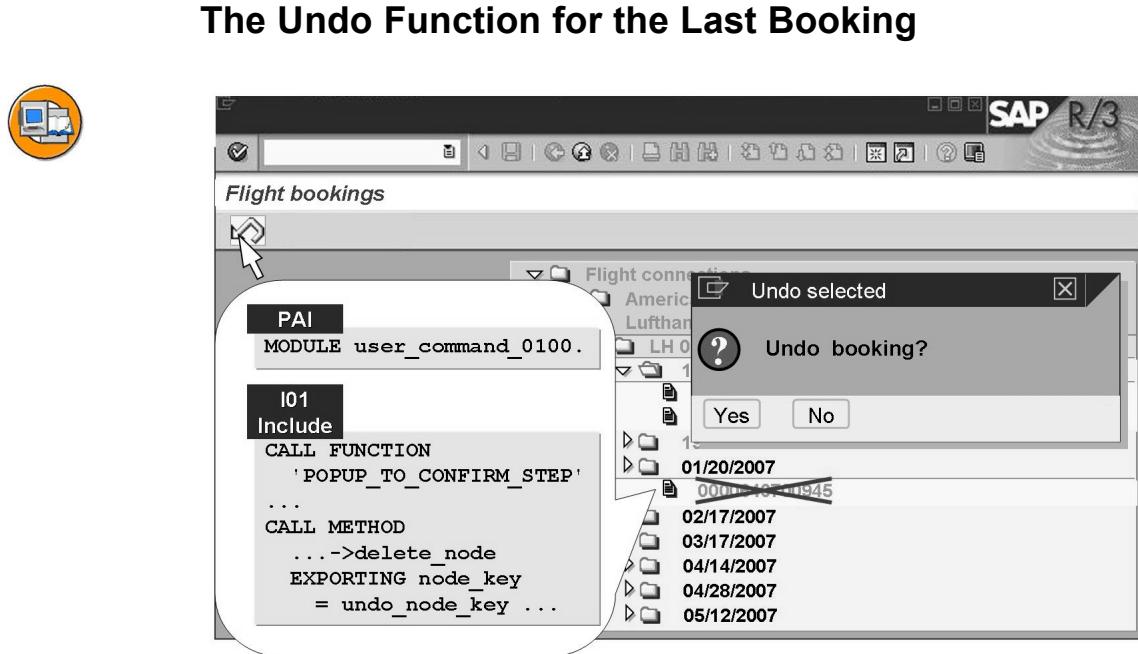


Figure 209: The Undo Function for the Last Booking

We may wish to put the Undo button on the program interface, since we will be activating the Save pushbutton here later.

Alternatively, we could create a toolbar control instance.

Either way we should display a confirmation prompt before the user undoes the booking by removing the node again from the tree.

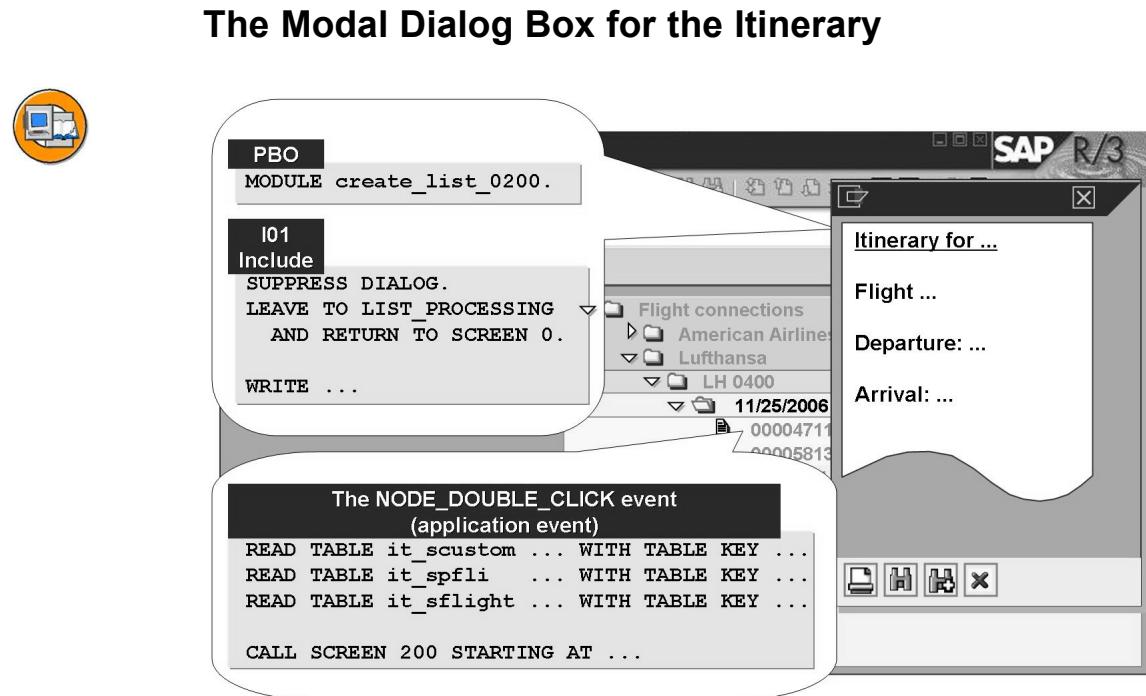


Figure 210: The Modal Dialog Box for the Itinerary

To format the itinerary, we need additional information from the buffer tables. To read these, we need to extract the application key from the ID of the node that the user has double-clicked.

The modal dialog box should **only** appear if the user double-clicks a **booking** node.

To process the ABAP list in the modal dialog box, use the SUPPRESS DIALOG and LEAVE TO LIST-PROCESSING ABAP statements. Set the standard list status including the Print function using SET PF-STATUS space.

## Other Options for Extending the Program

We could add these functions:



- Inserting a new customer record
- Show and hide the Undo function dynamically
- Search for all of a customer's bookings in the tree and print a complete confirmation
- Display the complete confirmation as the next screen (fullscreen) after the user has saved
- Allow the user to delete any booking using Drag&Drop

For the last three of these functions, we need to buffer the inserted bookings in another internal table, defined globally in the program.

We can then make and undo bookings using this table, until the user is happy with the result.

The Save function should then trigger an update of the database table (or tables), typically by **asynchronous update**.

Note that you need a profound knowledge of SAP's bundling, lock, and authorization concepts, to update the database safely and consistently. For more information on these topics, refer to the online documentation or attend the course **BC414 (Database Updates)**.



# Exercise 19: Creating a One-Screen Transaction for Booking Flights (OPTIONAL)

## Exercise Objectives

After completing this exercise, you will be able to:

- Show a selection option in a classical screen
- Attach several docking containers to different edges of the screen
- Fill and update a SAP Grid Control with optimal runtime performance, using dialogs
- Fill a tree control, update it, and have it react to double-clicking events
- Implement Drag&Drop procedures between the SAP Grid and tree controls including the Undo function
- Display data as an ABAP list in a modal dialog box

## Business Example

Develop a program that performs the following functions:

- Display a set of customer data records in a table. Restrict the set by customer number
- Book any number of flights for any number of customers from this set
- Undo the last booking
- Format the additional data for any booking to create an "itinerary" (booking confirmation)

**Program:** SAPMZBC412\_##\_UDC\_EX1

**Template:** SAPBC412\_UDCT\_EXERCISE\_1

**Model solution:** SAPBC412\_UDCS\_EXERCISE\_1

where ## is the group number.

## Task:

Perform the following tasks:

1. Copy the template SAPBC412\_UDCT\_EXERCISE\_1 – **including all its sub-objects** – to a new program with the name **SAPMZBC412\_##\_UDC\_EX1**. Create the transaction code **ZBC412\_##\_UDC1** and get to know the features of this copied program

*Continued on next page*

2. Define a selection option for customer numbers (suggested name: `so_cust`) on a selection screen (suggested screen number: 1100), used as a subscreen. The customer data records are stored in the transparent table SCUSTOM. Show the selection screen in a subscreen area (suggested name: `subarea_cust`) of screen 100.
3. Create two docking containers (suggested names: `ref_cont_left` and `ref_cont_right`). Give them suitable dimensions and link them to screen 100 so that the screen is split sensibly.
4. Place a SAP Grid Control instance (suggested name: `ref_alv`) in the left screen area and use it to display the customer data records from the transparent table SCUSTOM, for the selected customers.

Update the content only when the user changes the selection option criteria.

5. Place a tree control instance of the **simple tree model** (suggested name: `ref_tree_model`) in the right screen area and use it to display the flight connection data from the flight data model (that is, from the transparent tables SCARR, SPFLI, and SFLIGHT).

#### Optional:

Optimize this step for runtime performance. Use the possibility you have of reading some of the data afterwards, by reacting to a suitable event

6. Implement a booking procedure as a Drag&Drop function between the SAP Grid Control and the flight connection node of the tree control.
7. Extend the status `STATUS_NORM_0100` to include an *Undo* function (suggested name: '`UNDO`'). Allow the user to remove the booking node inserted most recently from the tree control
8. Implement the following function, to be triggered when the user double-clicks an inserted booking node:

Format some additional data for the relevant flight booking as an “itinerary” and booking confirmation, in the form of an ABAP list. Display this as a modal dialog box.

Then allow the user to print this list.

## Solution 19: Creating a One-Screen Transaction for Booking Flights (OPTIONAL)

### Task:

Perform the following tasks:

1. Copy the template SAPBC412\_UDCT\_EXERCISE\_1 – **including all its sub-objects** – to a new program with the name **SAPMZBC412##UDC\_EX1**. Create the transaction code **ZBC412##UDC1** and get to know the features of this copied program
  - a)
2. Define a selection option for customer numbers (suggested name: **so\_cust**) on a selection screen (suggested screen number: 1100), used as a subscreen. The customer data records are stored in the transparent table SCUSTOM. Show the selection screen in a subscreen area (suggested name: **subarea\_cust**) of screen 100.
  - a) See the source text excerpt from the sample solution in the TOP include **BC412\_UDCS\_EXERCISE1\_TOP**.
3. Create two docking containers (suggested names: **ref\_cont\_left** and **ref\_cont\_right**). Give them suitable dimensions and link them to screen 100 so that the screen is split sensibly.
  - a) See the source text excerpt from the sample solution in the PBO Module **init\_container\_processing\_0100**.
4. Place a SAP Grid Control instance (suggested name: **ref\_alv**) in the left screen area and use it to display the customer data records from the transparent table SCUSTOM, for the selected customers.

Update the content only when the user changes the selection option criteria.

  - a) See the source text excerpt from the sample solution in the and PBO Module **init\_alv\_processing\_0100**.
5. Place a tree control instance of the **simple tree model** (suggested name: **ref\_tree\_model1**) in the right screen area and use it to display the flight connection data from the flight data model (that is, from the transparent tables SCARR, SPFLI, and SFLIGHT).

### Optional:

*Continued on next page*

Optimize this step for runtime performance. Use the possibility you have of reading some of the data afterwards, by reacting to a suitable event

- a) See the source text excerpt from the sample solution in the PBO Module **init\_tree\_processing\_0100**
- 6. Implement a booking procedure as a Drag&Drop function between the SAP Grid Control and the flight connection node of the tree control.
  - a) See the source text excerpt from the sample solution in the PBO Module **init\_dragdrop\_processing\_0100**.
- 7. Extend the status STATUS\_NORM\_0100 to include an *Undo* function (suggested name: 'UNDO'). Allow the user to remove the booking node inserted most recently from the tree control
  - a)
- 8. Implement the following function, to be triggered when the user double-clicks an inserted booking node:

Format some additional data for the relevant flight booking as an “itinerary” and booking confirmation, in the form of an ABAP list. Display this as a modal dialog box.

Then allow the user to print this list.

- a) See the source text excerpt from the sample solution in the PBO Module **create\_list\_0200**.

## Result

The complete source code for the model solution is shown below.

### Screen flow logic

#### SCREEN 100

```
PROCESS BEFORE OUTPUT.
MODULE status_0100.
MODULE init_container_processing_0100.
MODULE init_dragdrop_processing_0100.
MODULE init_alv_processing_0100.
MODULE init_tree_processing_0100.
CALL SUBSCREEN subarea_cust INCLUDING sy-cprog '1100'.
```

```
PROCESS AFTER INPUT.
CALL SUBSCREEN subarea_cust.
MODULE exit_command_0100 AT EXIT-COMMAND.
MODULE user_command_0100.
```

*Continued on next page*

## SCREEN 200

```

PROCESS BEFORE OUTPUT.
MODULE create_list_0200.
MODULE status_0200.

PROCESS AFTER INPUT.
MODULE user_command_0200.
```

### ABAP program

#### Main Program

```

INCLUDE bc412_udcs_exercise_1top.
INCLUDE bc412_udcs_exercise_1c01.
INCLUDE bc412_udcs_exercise_1o01.
INCLUDE bc412_udcs_exercise_1i01.
INCLUDE bc412_udcs_exercise_1f01.
INCLUDE bc412_udcs_exercise_1e01.
```

#### Declarations

```

*&-----*
*& Include BC412_UDCS_EXERCISE1_TOP *
*&-----*
PROGRAM sapbc412_udct_exercisel MESSAGE-ID bc412
          NO STANDARD PAGE HEADING LINE-SIZE 80.
DATA:
* container:
  ref_cont_left  TYPE REF TO cl_gui_docking_container,
  ref_cont_right TYPE REF TO cl_gui_docking_container,

* content:
  ref_alv         TYPE REF TO cl_gui_alv_grid,
  ref_tree_model  TYPE REF TO cl_simple_tree_model,

* node buffers for tree processing:
  it_nodes        TYPE treemsnota,
  undo_node_key   TYPE tm_nodekey,

* for event registration:
  it_events       TYPE cntl_simple_events,
  wa_event        LIKE LINE OF it_events,

* drag&drop-specific:
  wa_layout       TYPE lvc_s_layo,
```

*Continued on next page*

```

ref_flav_src TYPE REF TO cl_dragdrop,
ref_flav_trg TYPE REF TO cl_dragdrop,

handle_src TYPE i,
handle_trg TYPE i,

* application-data:
it_scarr TYPE SORTED TABLE OF scarr
    WITH UNIQUE KEY carrid,
it_spfli TYPE SORTED TABLE OF spfli
    WITH UNIQUE KEY carrid connid,
it_sflight TYPE SORTED TABLE OF sflight
    WITH UNIQUE KEY carrid connid fldate,
it_scustom TYPE STANDARD TABLE OF scustom
    WITH NON-UNIQUE KEY id,

wa_scarr LIKE LINE OF it_scarr,
wa_spfli LIKE LINE OF it_spfli,
wa_sflight LIKE LINE OF it_sflight,
wa_scustom LIKE LINE OF it_scustom,

* for itenary list:
wa_sbook TYPE sbook,
arr_date TYPE sflight-fldate.

* subscreen content for screen 100:
SELECTION-SCREEN BEGIN OF SCREEN 1100 AS SUBSCREEN.
SELECT-OPTIONS so_cust FOR wa_scustom-id.
SELECTION-SCREEN END OF SCREEN 1100.

* auxiliary for screen 100:
DATA:
ok_code      TYPE sy-ucomm,
copy_ok_code LIKE ok_code,
l_answer     TYPE c,
set_cust_old LIKE RANGE OF wa_scustom-id.

```

### Local Classes

```

*-----*
*   INCLUDE BC412UDCS_EXERCISE_1C01
*-----*
*-----*

```

*Continued on next page*

```

*      CLASS lcl_booking DEFINITION
*-----*
CLASS lcl_booking DEFINITION.
  PUBLIC SECTION.
    DATA wa_customer TYPE scustom.
  ENDCLASS.

*-----*
*      CLASS lcl_event_handler DEFINITION
*-----*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS handle_expand_no_children
      FOR EVENT expand_no_children
      OF cl_simple_tree_model
      IMPORTING node_key.

    CLASS-METHODS handle_ondrag FOR EVENT ondrag
      OF cl_gui_alv_grid
      IMPORTING
        e_row
        e_dragdropobj.

    CLASS-METHODS handle_drop FOR EVENT drop
      OF cl_simple_tree_model
      IMPORTING
        node_key
        drag_drop_object.

    CLASS-METHODS handle_node_double_click
      FOR EVENT node_double_click
      OF cl_simple_tree_model
      IMPORTING
        node_key.
  ENDCLASS.

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

METHOD handle_expand_no_children.
  DATA:
    BEGIN OF l_wa_key_sflight_c,

```

*Continued on next page*

```

        carrid(3) TYPE c,
        connid(4) TYPE n,
        fldate(8) TYPE n,
END OF l_wa_key_sflight_c,

l_it_sflight LIKE it_sflight,
l_it_nodes TYPE treemsnota,
l_wa_node  TYPE treemsnodi,
date_text(10) TYPE c.

*   get application keys:
CLEAR wa_sflight.

SPLIT node_key AT c_sep INTO
    l_wa_key_sflight_c-carrid
    l_wa_key_sflight_c-connid
    l_wa_key_sflight_c-fldate.
IF l_wa_key_sflight_c-connid CO space.
    CLEAR l_wa_key_sflight_c-connid.
ENDIF.
IF l_wa_key_sflight_c-fldate CO space.
    CLEAR l_wa_key_sflight_c-fldate.
ENDIF.

MOVE-CORRESPONDING l_wa_key_sflight_c TO wa_sflight.

*   create only sflight-notes:
IF ( NOT wa_sflight-connid IS INITIAL
AND wa_sflight-fldate IS INITIAL ).

*   maybe data for sflight-nodes is already buffered:
LOOP AT it_sflight INTO wa_sflight
    WHERE carrid = wa_sflight-carrid
          AND connid = wa_sflight-connid.
    INSERT wa_sflight INTO TABLE l_it_sflight.
ENDLOOP.
IF sy-subrc <> 0.
*   otherwise get more sflight-data ...
    SELECT * FROM sflight
        INTO TABLE l_it_sflight
        WHERE carrid = wa_sflight-carrid
              AND connid = wa_sflight-connid.
    IF sy-subrc <> 0.
        MESSAGE a060.
    ENDIF.

```

*Continued on next page*

```

*      buffer additional data:
      INSERT LINES OF l_it_sflight INTO TABLE it_sflight.
      ENDIF.
*      and create nodes:
      LOOP AT l_it_sflight INTO wa_sflight.
      CLEAR l_wa_node.
      CONCATENATE wa_sflight-carrid
                  wa_sflight-connid
                  wa_sflight-fldate
                  INTO l_wa_node-node_key
                  SEPARATED BY ','.
      CONCATENATE wa_sflight-carrid
                  wa_sflight-connid
                  INTO l_wa_node-relatkey
                  SEPARATED BY ','.
      l_wa_node-relatship = cl_simple_tree_model->relat_last_child.
      l_wa_node-isfolder = 'X'.
      l_wa_node-expander = 'X'.
*      drop booking should only be possible for flight date nodes:
      l_wa_node-dragdropid = handle_trg.
      WRITE wa_sflight-fldate TO date_text.
      l_wa_node-text = date_text.
      INSERT l_wa_node INTO TABLE l_it_nodes.
      ENDLOOP.

      CALL METHOD ref_tree_model->add_nodes
          EXPORTING
              node_table      = l_it_nodes
          EXCEPTIONS
              OTHERS          = 1.
      IF sy-subrc <> 0.
      MESSAGE a012.
      ENDIF.

*      expand the node:
      CALL METHOD ref_tree_model->expand_node
          EXPORTING
              node_key = node_key.

      ENDIF.

      ENDMETHOD.

*-----*
METHOD handle_ondrag.
DATA l_ref_booking TYPE REF TO lcl_booking.
```

*Continued on next page*

```

CREATE OBJECT l_ref_booking.
READ TABLE it_scustom INTO l_ref_booking->wa_customer
INDEX e_row-index.
e_dragdropobj->object = l_ref_booking.
ENDMETHOD.

*-----
METHOD handle_drop.
DATA:
l_ref_booking TYPE REF TO lcl_booking,
l_wa_newnode TYPE treemsnode.

CATCH SYSTEM-EXCEPTIONS move_cast_error = 1.
l_ref_booking ?= drag_drop_object->object.
ENDCATCH.

IF sy-subrc = 0.
*      create new booking node,
*      due to didactical reasons just take sy-datum and sy-uzzeit
*      as node key here, begin with them to make separation easier
*
*      correct solution would be:
*      get number from number range intervall (->BC414)
CONCATENATE sy-datum sy-uzzeit
node_key          "imported!
l_ref_booking->wa_customer-id
INTO l_wa_newnode-node_key
SEPARATED BY ','.

CALL METHOD ref_tree_model->add_node
EXPORTING
node_key          = l_wa_newnode-node_key
relative_node_key = node_key      "imported!
relationship      = cl_simple_tree_model->relat_last_child
isfolder          = space
text              = l_wa_newnode-node_key
EXCEPTIONS
OTHERS            = 1.

IF sy-subrc <> 0.
CALL METHOD drag_drop_object->abort.
ENDIF.

*      buffer new node for undo functionality:
undo_node_key = l_wa_newnode-node_key.

```

*Continued on next page*

```

*      of course new booking would have to be stored in table SBOOK now
*      left out here due to didactical reasons
*      ...

*      expand the parent node:
CALL METHOD ref_tree_model->expand_node
EXPORTING
    node_key = node_key.

ELSE.
    CALL METHOD drag_drop_object->abort.
ENDIF.
ENDMETHOD.

*-----
METHOD handle_node_double_click.

DATA:
    BEGIN OF l_wa_key_sbook_c,
        carrid(3)    TYPE c,
        connid(4)    TYPE n,
        fldate(8)    TYPE n,
        *      bookid(8)    TYPE n, left out for didactical reasons
        customid(8)  TYPE n,
    END OF l_wa_key_sbook_c.
    CLEAR wa_sbook.

* separate sy-datum and sy-uzeit first:
DO 2 TIMES.
    SEARCH node_key FOR ','.
    SHIFT node_key BY sy-fdpos PLACES.
    SHIFT node_key.
ENDDO.

SPLIT node_key AT c_sep INTO
    l_wa_key_sbook_c-carrid
    l_wa_key_sbook_c-connid
    l_wa_key_sbook_c-fldate
    l_wa_key_sbook_c-customid.
IF l_wa_key_sbook_c-connid CO space.
    CLEAR l_wa_key_sbook_c-connid.
ENDIF.
IF l_wa_key_sbook_c-fldate CO space.
    CLEAR l_wa_key_sbook_c-fldate.
ENDIF.
IF l_wa_key_sbook_c-customid CO space.
    CLEAR l_wa_key_sbook_c-customid.
ENDIF.

```

*Continued on next page*

```

MOVE-CORRESPONDING l_wa_key_sbook_c TO wa_sbook.

*   display only booking data:
IF NOT wa_sbook-customid IS INITIAL.
  READ TABLE it_scustom INTO wa_scustom
    WITH TABLE KEY id = wa_sbook-customid.
  IF sy-subrc = 0.
    CALL SCREEN 200 STARTING AT 50 5
      ENDING   AT 130 30.
  ELSE.
    EXIT.
  ENDIF.
ELSE.
  EXIT.
ENDIF.
ENDMETHOD.
ENDCLASS.
```

### Event blocks

```

*-----*
*   INCLUDE BC412_UDCS_EXERCISE_1E01
*-----*
LOAD-OF-PROGRAM.

* default values for select-option:
so_cust-low      = 1.
so_cust-high     = 20.
so_cust-sign     = 'I'.
so_cust-option   = 'BT'.
APPEND so_cust.

so_cust-low      = 40.
so_cust-high     = 60.
so_cust-sign     = 'I'.
so_cust-option   = 'BT'.
APPEND so_cust.

* get initial application data:
SELECT * FROM scarr
  INTO TABLE it_scarr.
IF sy-subrc <> 0.
  MESSAGE a060.
ENDIF.
SELECT * FROM spfli
  INTO TABLE it_spfli.
```

*Continued on next page*

```
IF sy-subrc <> 0.
MESSAGE a060.
ENDIF.
```

## Modules

```
-----*
***INCLUDE BC412_UDCS_EXERCISE_1001 .
*-----*

*&-----*
*&      Module  status_0100  OUTPUT
*&-----*
*      set GUI for screen 0100
*-----*
MODULE status_0100 OUTPUT.
  SET PF-STATUS 'STATUS_NORM_0100'.
  SET TITLEBAR 'TITLE_NORM_0100'.
ENDMODULE.          " status_0100  OUTPUT

*&-----*
*&      Module  init_container_processing_0100  OUTPUT
*&-----*
*      create screen "frames"
*-----*
MODULE init_container_processing_0100 OUTPUT.
  IF ref_cont_left IS INITIAL.

    CREATE OBJECT ref_cont_right
    EXPORTING
      ratio      = 50
      side       = cl_gui_docking_container=>dock_at_right
    EXCEPTIONS
      others           = 1.
  IF sy-subrc NE 0.
    MESSAGE a010.
  ENDIF.

    CREATE OBJECT ref_cont_left
    EXPORTING
      ratio      = 90
      side       = cl_gui_docking_container=>dock_at_bottom
    EXCEPTIONS
      others           = 1.
  IF sy-subrc NE 0.
```

*Continued on next page*

```

MESSAGE a010.
ENDIF.

ENDIF.
ENDMODULE.           " init_container_processing_0100  OUTPUT

*&-----*
*&     Module  init_dragdrop_processing_0100  OUTPUT
*&-----*
*      creates flavor tables and retrieves handles to them
*-----*
MODULE init_dragdrop_processing_0100 OUTPUT.
IF ref_flav_src IS INITIAL.

CREATE OBJECT:
ref_flav_src,
ref_flav_trg.

CALL METHOD ref_flav_src->add
EXPORTING
flavor          = 'BOOK'
dragsrc         = 'X'
droptarget      = space
effect          = cl_dragdrop=>copy
effect_in_ctrl = cl_dragdrop=>none
EXCEPTIONS
OTHERS          = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

CALL METHOD ref_flav_trg->add
EXPORTING
flavor          = 'BOOK'
dragsrc         = space
droptarget      = 'X'
effect          = cl_dragdrop=>copy
effect_in_ctrl = cl_dragdrop=>none
EXCEPTIONS
OTHERS          = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

CALL METHOD ref_flav_src->get_handle

```

*Continued on next page*

```

IMPORTING
    handle      = handle_src
EXCEPTIONS
    OTHERS      = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

CALL METHOD ref_flav_trg->get_handle
IMPORTING
    handle      = handle_trg
EXCEPTIONS
    OTHERS      = 1.
IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

ENDIF.

ENDMODULE.          " init_dragdrop_processing_0100  OUTPUT

*&-----*
*&     Module  init_alv_processing_0100  OUTPUT
*&-----*
*     create alv object, link to container and fill with data
*-----*
MODULE init_alv_processing_0100 OUTPUT.
IF ref_alv IS INITIAL. " first screen processing

CREATE OBJECT ref_alv
EXPORTING
    i_parent      = ref_cont_left
EXCEPTIONS
    others        = 5
.

IF sy-subrc <> 0.
MESSAGE a045.
ENDIF.

SET HANDLER lcl_event_handler=>handle_ondrag FOR ref_alv.

*   assign d&d-handle:
wa_layout-s_dragdrop-row_ddid = handle_src.

*   get content for first display:

```

*Continued on next page*

```

SELECT * FROM scustom
      INTO TABLE it_scust
      WHERE id IN so_cust.
IF sy-subrc <> 0.
MESSAGE a060.
ENDIF.

CALL METHOD ref_alv->set_table_for_first_display
      EXPORTING
          i_structure_name      = 'SCUSTOM'
          is_layout              = wa_layout
      CHANGING
          it_outtab             = it_scust
      EXCEPTIONS
          OTHERS                = 4

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.

ELSE.
*   refresh grid if new customers are selected:
IF so_cust[] <> set_cust_old.
SELECT * FROM scustom
      INTO TABLE it_scust
      WHERE id IN so_cust.
IF sy-subrc = 0.
MOVE so_cust[] TO set_cust_old.
ELSE.
MESSAGE i064.
ENDIF.

CALL METHOD ref_alv->refresh_table_display
      EXCEPTIONS
          OTHERS                = 2

IF sy-subrc <> 0.
MESSAGE a012.
ENDIF.
ENDIF.
ENDIF.

ENDMODULE.           " init_alv_processing_0100  OUTPUT

*&-----*
*&     Module  init_tree_processing_0100  OUTPUT
*&-----*

```

*Continued on next page*

```

*      create tree object, link to container and fill with data
*-----*
MODULE init_tree_processing_0100 OUTPUT.
  IF ref_tree_model IS INITIAL.

    CREATE OBJECT ref_tree_model
      EXPORTING
        node_selection_mode = cl_simple_tree_model->node_sel_mode_single
      EXCEPTIONS
        others                  = 1.
      IF sy-subrc <> 0.
        MESSAGE a043.
      ENDIF.

    CALL METHOD ref_tree_model->create_tree_control
      EXPORTING
        parent                 = ref_cont_right
      EXCEPTIONS
        OTHERS                  = 1.
      IF sy-subrc <> 0.
        MESSAGE a012.
      ENDIF.

* other implementation here than in unit "Tree Control":
PERFORM create_node_table CHANGING it_nodes.

CALL METHOD ref_tree_model->add_nodes
  EXPORTING
    node_table              = it_nodes
  EXCEPTIONS
    OTHERS                  = 1.
  IF sy-subrc <> 0.
    MESSAGE a012.
  ENDIF.
CLEAR it_nodes. "not needed any more!"

SET HANDLER:
  lcl_event_handler=>handle_expand_no_children FOR ref_tree_model,
  lcl_event_handler=>handle_drop FOR ref_tree_model,
  lcl_event_handler=>handle_node_double_click FOR ref_tree_model.

wa_event-eventid = cl_simple_tree_model->eventid_node_double_click.
INSERT wa_event INTO TABLE it_events.
* Tree Model Instance sets filter for event EXPAND_NO_CHILDREN!
CALL METHOD ref_tree_model->set_registered_events

```

*Continued on next page*

```

EXPORTING
  events          = it_events
EXCEPTIONS
  OTHERS          = 1.
IF sy-subrc <> 0.
  MESSAGE a012.
ENDIF.

ENDIF.
ENDMODULE.           " init_tree_processing_0100  OUTPUT

*&-----*
*&      Module  STATUS_0200  OUTPUT
*&-----*
*      for the ABAP list
*-----*
MODULE status_0200 OUTPUT.
  SET PF-STATUS space.
  SET TITLEBAR 'LIST'.
ENDMODULE.           " STATUS_0200  OUTPUT

*&-----*
*&      Module  create_list_0200  OUTPUT
*&-----*
*      creates the itenary
*-----*
MODULE create_list_0200 OUTPUT.
  SUPPRESS DIALOG.
  LEAVE TO LIST-PROCESSING AND RETURN TO SCREEN 0.

* customer name:
  WRITE: /
    text-hdr COLOR COL_HEADING,
    wa_scustom-form COLOR COL_HEADING,
    wa_scustom-name COLOR COL_HEADING.
  ULINE.
  SKIP.

* connection data:
  READ TABLE it_spfli INTO wa_spfli
    WITH TABLE KEY carrid = wa_sbook-carrid
         connid = wa_sbook-connid.
IF sy-subrc = 0.
  WRITE: /
    text-fli,

```

*Continued on next page*

```

wa_spfli-carrid,
wa_spfli-connid.

SKIP.

READ TABLE it_sflight INTO wa_sflight
    WITH TABLE KEY carrid = wa_sbook-carrid
        connid = wa_sbook-connid
        fldate = wa_sbook-fldate.

IF sy-subrc = 0.
    WRITE: /
        text-dep,
        wa_sflight-fldate,
        wa_spfli-deptime,
        wa_spfli-airpfrom.

    arr_date = wa_sflight-fldate + wa_spfli-period.

    SKIP.
    WRITE: /
        text-arr,
        arr_date,
        wa_spfli-arrrtime,
        wa_spfli-airpto.

ELSE.
    EXIT.
ENDIF.
ELSE.
    EXIT.
ENDIF.

ENDMODULE.          " create_list_0200  OUTPUT

*-----*
***INCLUDE BC412_UDCS_EXERCISE_1I01 .
*-----*

*&-----*
*&     Module exit_command_0100  INPUT
*&-----*
*     Implementation of user commands of type 'E'.
*-----*

MODULE exit_command_0100 INPUT.
CASE ok_code.


```

*Continued on next page*

```

WHEN 'CANCEL'.           " cancel current screen processing
CLEAR l_answer.
CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
    EXPORTING
        *
        DEFAULTOPTION = 'Y'
        textline1      = text-004
        textline2      = text-005
        titel         = text-006
        cancel_display = ''
    IMPORTING
        answer        = l_answer.
CASE l_answer.
    WHEN 'J'.
        PERFORM free_control_ressources.
        LEAVE TO SCREEN 0.
    WHEN 'N'.
        CLEAR ok_code.
        SET SCREEN sy-dynnr.
ENDCASE.

WHEN 'EXIT'.             " leave program
CLEAR l_answer.
CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
    EXPORTING
        *
        DEFAULTOPTION = 'Y'
        textline1      = text-001
        textline2      = text-002
        titel         = text-003
        cancel_display = 'X'
    IMPORTING
        answer        = l_answer.
CASE l_answer.
    WHEN 'J' OR 'N'.          " no data to update
        PERFORM free_control_ressources.
        LEAVE PROGRAM.
    WHEN 'A'.
        CLEAR ok_code.
        SET SCREEN sy-dynnr.
ENDCASE.

ENDCASE.
ENDMODULE.               " exit_command_0100  INPUT

```

\*&-----\*

\*& Module user\_command\_0100 INPUT

\*&-----\*

*Continued on next page*

```

*      Implementation of user commands of type ' ':
*      - push buttons on the screen
*      - GUI functions
*-----
MODULE user_command_0100 INPUT.
copy_ok_code = ok_code.
CLEAR ok_code.

CASE copy_ok_code.
WHEN 'UNDO'.
    IF NOT undo_node_key IS INITIAL.
        CLEAR l_answer.
        CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
            EXPORTING
*             DEFAULTOPTION = 'Y'
*             textline1      = text-008
*             TEXTLINE2      =
*             titel          = text-009
*             cancel_display = ' '
            IMPORTING
*             answer         = l_answer.
        IF l_answer = 'J'.
            CALL METHOD ref_tree_model->delete_node
                EXPORTING
*                 node_key       = undo_node_key
                EXCEPTIONS
*                 node_not_found = 1
*                 OTHERS          = 2.
            IF sy-subrc <> 0.
                MESSAGE a012.
            ENDIF.
            CLEAR undo_node_key.

        ENDIF.
    ENDIF.

WHEN 'BACK'.
    CLEAR l_answer.
    CALL FUNCTION 'POPUP_TO_CONFIRM_STEP'
        EXPORTING
*             DEFAULTOPTION = 'Y'
*             textline1      = text-004
*             textline2      = text-005
*             titel          = text-007
*             cancel_display = ' '

```

*Continued on next page*

```

IMPORTING
    answer      = l_answer.

CASE l_answer.
    WHEN 'J'.
        PERFORM free_control_ressources.
        LEAVE TO SCREEN 0.
    WHEN 'N'.
        SET SCREEN sy-dynnr.
ENDCASE.

WHEN OTHERS.
ENDCASE.

ENDMODULE.          " user_command_0100  INPUT

*&-----*
*&     Module  USER_COMMAND_0200  INPUT
*&-----*
*     back to screen 100
*-----*
MODULE user_command_0200 INPUT.
    copy_ok_code = ok_code.
    CLEAR ok_code.
ENDMODULE.          " USER_COMMAND_0200  INPUT

```

### Subroutines

```

*-----*
***INCLUDE BC412_UDCS_EXERCISE_1F01 .
*-----*

*&-----*
*&     Form  free_control_ressources
*&-----*
*     free all control related ressources
*-----*
FORM free_control_ressources.
    CALL METHOD:
        ref_cont_left->free,
        ref_cont_right->free.

    FREE:
        ref_alv,
        ref_tree_model,

```

*Continued on next page*

```

        ref_cont_left,
        ref_cont_right.
ENDFORM.                                     " free_control_ressources

*-----*
*      FORM create_node_table                  *
*-----*
*      build up a hierarchy consisting of      *
*      carriers, connections and flight dates   *
*-----*
*  -->  P_IT_NODES                         *
*-----*
FORM create_node_table CHANGING p_it_nodes LIKE it_nodes.

DATA:
l_wa_node LIKE LINE OF p_it_nodes,
date_text(10) TYPE c.

l_wa_node-node_key    = 'ROOT'.
l_wa_node-isfolder    = 'X'.
l_wa_node-expander    = 'X'.
l_wa_node-text         = text-car.
INSERT l_wa_node INTO TABLE p_it_nodes.

* scarr-nodes:
CLEAR l_wa_node.
LOOP AT it_scarr INTO wa_scarr.
  CLEAR l_wa_node.
  l_wa_node-node_key    = wa_scarr-carrid.
  l_wa_node-relatkey    = 'ROOT'.
  l_wa_node-relationship = cl_simple_tree_model=>relat_last_child.
  l_wa_node-isfolder    = 'X'.
  l_wa_node-expander    = 'X'.
  l_wa_node-text         = wa_scarr-carrname.
  INSERT l_wa_node INTO TABLE p_it_nodes.
ENDLOOP.

* spfli-nodes:
CLEAR l_wa_node.
LOOP AT it_spfli INTO wa_spfli.
  CLEAR l_wa_node.
  CONCATENATE wa_spfli-carrid
            wa_spfli-connid
            INTO l_wa_node-node_key
            SEPARATED BY ';'.

```

*Continued on next page*

```
l_wa_node-relatkey    = wa_spfli-carrid.  
l_wa_node-relatship   = cl_simple_tree_model=>relat_last_child.  
l_wa_node-isfolder    = 'X'.  
l_wa_node-expander    = 'X'.  
CONCATENATE wa_spfli-carrid  
               wa_spfli-connid  
               ':'  
               wa_spfli-cityfrom  
               ' -> '  
               wa_spfli-cityto  
               INTO l_wa_node-text  
               SEPARATED BY space.  
INSERT l_wa_node INTO TABLE p_it_nodes.  
ENDLOOP.  
  
ENDFORM.                                     " CREATE_NODE_TABLE
```



## Lesson Summary

You should now be able to:

- Name the benefits and uses of some classic screen elements
- Name the benefits and uses of some control-based screen elements
- Choose and implement the right classic or control-based dialog programming technique for the task on hand
- Combine classical and control-based screen elements in the same program - that is, implement the interaction between them



## Unit Summary

You should now be able to:

- Name the benefits and uses of some classic screen elements
- Name the benefits and uses of some control-based screen elements
- Choose and implement the right classic or control-based dialog programming technique for the task on hand
- Combine classical and control-based screen elements in the same program - that is, implement the interaction between them



## Course Summary

You should now be able to:

- Use the SAP Control Framework
- Use SAP container controls
- Use selected EnjoySAP Controls (Picture, HTML Viewer, Text Edit, SAP List Viewer, and Tree)
- Use special – control-based – mouse operations (context menus and Drag&Drop)



# Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.