Controls Tutorial (BC-CI)

HELP.BCCICTUT

Release 4.6C

Copyright

Icons

Icon	Meaning
⚠	Caution
	Example
	Note
	Recommendation
	Syntax

Contents

Controls Tutorial (BC-CI)

What are Controls?

Controls are independent binary software components that can be reused. Developers can build controls into their applications and use the functions they provide. Typically, controls are used for designing the user interface. However, using controls is not restricted to visible components.

Who Has to Deal With Controls?

Users...

...do not see the individual components that make up an application. This is why the integration of controls is transparent to them. They see a single application with which they work.

Application Developers...

...use controls in their applications. They are not interested in technical details. When programming their application, they access a logical wrapper class that helps them with the integration of the control.

Developers of the Wrapper Class...

...program the logical wrapper of individual controls. Therefore, they must have extensive technical know-how of controls and SAP's *Control Framework*.

Control Developers...

...create controls themselves. Reusing controls requires that a standardized interface is in place with which control developers must comply.

Whom Helps this Tutorial?

This documentation is designed for *application developers* who want to build controls into their ABAP programs.

BC - Component Integration: Controls Tutorial

The *BC - Component Integration: Controls Tutorial* contains introductory information for developers who want to use wrapped controls in their ABAP programs. However, the tutorial only covers custom controls that developers can use to integrate modern interface elements into their applications.

The Notes to the User [Page 6] section lists the technical requirements and tells you which programming know-how you need in order to be able to work through the tutorial. In addition, this section gives you a short overview of how controls are integrated with the R/3 System.

Each exercise of the tutorial builds on information learned in previous exercises. The exercises are grouped together in lessons that each cover a specific subject:

For each subject covered, there is an introductory section that provides you with background information.

Notes to the User

Read the following notes before working through the tutorial:

Prerequisites

The tutorial is designed to acquaint you step by step with the concepts of control programming. You should have practical knowledge of ABAP programming and be familiar with the tools contained in the ABAP Workbench. For information on this, see:

BC - ABAP Workbench Tutorial [Ext.]

BC - ABAP Workbench Tools [Ext.]

Controls are accessed through classes of *ABAP Objects* – which is the object-oriented extension of ABAP. However, a general introduction to object-oriented programming would have been beyond the scope of this tutorial. Nevertheless, the **Basics** sections of several lessons explain those ABAP Objects concepts that are necessary to understand controls programming. If you are familiar with these concepts, you can skip the corresponding sections.

For introductory information on object-oriented ABAP programming, see ABAP Objects [Ext.] in BC - ABAP Programming [Ext.].

System Configuration

As of Release 4.0A, controls programming is supported through function modules. Starting with Release 4.5A, *ABAP Objects* is available for object-oriented programming with ABAP. An object-oriented version of the *Control Framework* is available as of Release 4.6A. The programming examples provided in this tutorial require that you have installed this release.

There are two other factors which are of importance:

- The operating system used on the frontend computer
- The control version installed on the frontend computer

Frontend Platform

Controls are addressed through a standardized interface. The system supports two of the common standards that are available on the market: *ActiveX* and *JavaBeans*.

Until Release 4.5B, you must use function modules in your ABAP program to check if the frontend system supports the relevant standard. As of Release 4.6A, the *Control Framework* performs this check automatically and triggers an exception if an error occurs. You use the function modules to respond individually to such an error, for example, if there is no JavaBean implementation of the control yet.

Interface Standards Supported

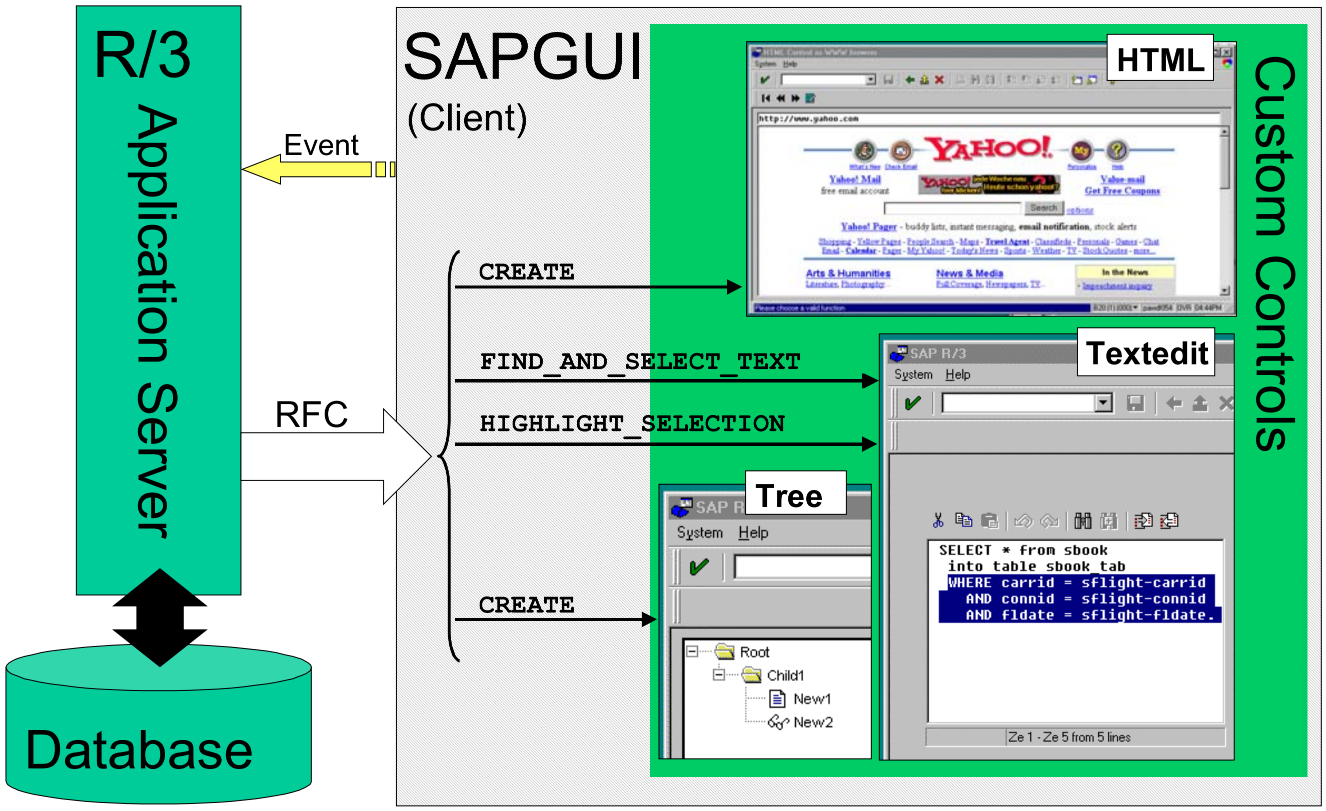
	Windows95 (32 Bit) **WindowsNT**	**Other platforms**
Standard used	ActiveX	JavaBeans
Function module that checks if the standard is available on the frontend (only required until Release 4.5B)	HAS_GUI_ACTIVEX	HAS_GUI_JAVABEAN

Controls Version

When you install a local SAPGUI on your machine, the system also installs the appropriate controls. If you install a new local SAPGUI, you should first uninstall the old version to ensure consistency.

Architecture

R/3 Architecture



```
SELECT * from sbook
 into table sbook_tab
 WHERE carrid = sflight-carrid
   AND connid = sflight-connid
   AND fldate = sflight-fldate.
```

Explanation

When you program controls, interaction between the application server (backend) and the SAPGUI (frontend) is of central importance. By calling ABAP methods, you can create the controls installed on the frontend, modify the control properties, and affect the behavior of these controls. The application server transfers these methods to the frontend by means of a Remote Function Call (RFC) and executes them.

The user of an application program triggers control events by performing specific actions (for example, double-clicking a node in a tree structure). You can decide which events you want to catch in your ABAP program and then register these events for the relevant control. A triggered event is transferred from the SAPGUI to the application server. In response to this event, you can then affect the behavior of the control with additional method calls.

Lesson 1: Creating a Custom Control

Task

In this lesson, you first of all create the prerequisites for integrating a control into an ABAP program. Next, you create a control for editing texts whose functions you will enhance step by step later on.

Overview: Objectives

In this lesson, you learn:

- How to integrate a visible control into a screen
- How to create a control
- Which functions are automatically supported by the textedit control

Basics

Controls are wrapped through ABAP Objects classes. Creating a control is consequently equivalent to instantiating an object, that is, generating an instance.

Basically, you generate objects using a *constructor* that determines the initial properties of the object. Each class that allows you to generate objects has such a constructor.

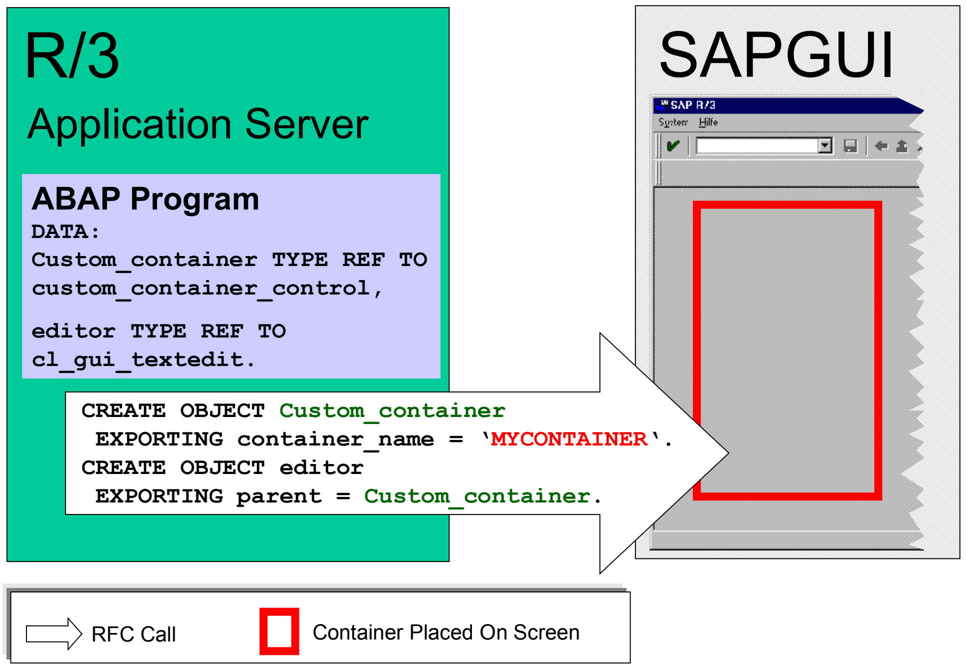
When you generate an object, you do not explicitly call the constructor. Instead, you create an instance using the **`CREATE OBJECT`** statement that implicitly executes the constructor.

To Continue

Lesson 2: Event Handling [Page 19] describes how to catch and further process events of a control.

Introduction

R/3 Architecture



```
DATA:
Custom_container TYPE REF TO
custom_container_control,

editor TYPE REF TO
cl_gui_textedit.
```

```
CREATE OBJECT Custom_container
 EXPORTING container_name = 'MYCONTAINER'.
CREATE OBJECT editor
 EXPORTING parent = Custom_container.
```

Explanation

You must assign each control that you want to display on a screen to a *container control*. The graphic above shows how a textedit control is integrated into a screen by means of a *custom container control*. Using a container created in the *Screen Painter*, you must determine an area that the custom container control is to fill.

A container control organizes the representation of one or more controls. Depending on the type of container control, you can:

- Place a control on the screen at a fixed position, for example, on a subscreen (*custom container control*)
- Embed several controls into a container control that is divided into areas (*splitter container control* and *easy splitter container control*)
- Represent a control in a modeless dialog box that can be moved freely (*dialog box container control*)
- Dock a control with a selected screen margin (*docking container control*).

For a detailed description on the individual container controls, see SAP Container [Ext.].

It is also possible to nest container controls. For example, you can specify a container control for the `parent` parameter of a container control. This way, you can arrange controls on the screen in many different ways.

Once you have created a container control, you can also display it on another screen. To do this, you must reassign the control, in other words, link it to another screen. All controls that have specified this container control as their `parent`, are also reassigned. If you have a nested container control, you can only reassign that control which contains all other controls.

Each screen is assigned to a popup level of 0 to 9. During their lifetime, container controls cannot be assigned to any other level than the one that was specified when they were created. If you want to display a control at a different popup level, you must create a new container control for this level.

Conclusions

Based on the architecture, we can derive the following steps for creating a control:

1. To display a control in a specific screen area, you create a container in the *Screen Painter*.
2. Using the `CREATE OBJECT` statement, you create a container control at PBO time that you assign to the screen.
3. Using the `CREATE OBJECT` statement, you create a control instance at PBO time that you assign to the container control.

Exercise 1: Reserving an Area for a Control

Prerequisites

You have created an executable program as a local object.

Procedure

1. Add the following lines to the source code of your main program:

   ```
   START-OF-SELECTION.
   SET SCREEN '100'.
   ```

   Then create the screen with the following attributes:

   – *Screen type*: Normal

- *Next screen*: 100

2. Edit the *layout* of your screen in the *Screen Painter* and mark an area for your custom control.

   If you use the alphanumeric mode of the *Screen Painter*, follow these steps:

   a. Place your cursor on a line of the screen and choose *Edit → Create element → Custom control*.

   b. To select the bottom right corner of the area, you can either double-click the corresponding line and column, or choose the *Mark end of ctrl* pushbutton. The system displays a dialog box for the element attributes.

   c. Enter the name **MYCONTAINER1** and *copy* your settings.

   If you use the graphical mode of the *Screen Painter*, follow these steps:

   a. Choose the custom control icon (which you can recognize by the letter 'C') and mark an area on the screen.

   b. Enter the name **MYCONTAINER1** in the *Name* field. You can also define the name on the attributes dialog box that is displayed when you double-click the element you have created.

3. To be able to exit the screen after the program has started, add a pushbutton to your screen and determine the function code in the PAI module:

   a. In the *Layout* editor of the *Screen Painter*, define a pushbutton with function code **EXIT** and place it on the screen.

   b. In the element list, enter the name **OK_CODE** for the element of type **OK**.

   c. Create PAI module **user_command_0100** and use it to determine the function code of the pushbutton:

```
MODULE USER_COMMAND_0100 INPUT.
    CASE OK_CODE.
       WHEN 'EXIT'.
           LEAVE TO SCREEN 0.
    ENDCASE.
ENDMODULE.                   " USER_COMMAND_0100 INPUT
```

   d. Declare **OK_CODE** as a global data field by double-clicking it and adding the following line to your main program:

```
DATA ok_code LIKE sy-ucomm.
```

4. Save your settings, and activate all your objects.

Check Your Work

When you start your program, the system displays your screen with one pushbutton. The container for the custom control is not displayed. If you click the pushbutton, the system returns you to your main program.

Exercise 2: Creating a Control and its Container

Prerequisites

In Exercise 1, you created a screen for your main program on which you placed an area for a *custom container control*. In the next step, you create this control and link it to a Textedit Control [Ext.] [Ext.].

Procedure

1. Declare the following variables in your main program:

- a reference variable to the class of the custom container control
- a reference variable to the class of the textedit control, and
- a variable that contains the ID of the current report:

```
DATA: custom_container TYPE REF TO cl_gui_custom_container,
      editor TYPE REF TO cl_gui_textedit,
      repid LIKE sy-repid.
```

2. Create constant `line_length` that determines the length of lines in the text window of the textedit control:

```
CONSTANTS: line_length TYPE I VALUE 256.
```

3. Create PBO module `status_100` in the flow logic of screen 100.

4. In the PBO module, instantiate the custom container control and link it to the **`MYCONTAINER1`** custom control area:

```
IF EDITOR IS INITIAL.
   repid = sy-repid.
   CREATE OBJECT custom_container
      EXPORTING
         CONTAINER_NAME = 'MYCONTAINER1'
      EXCEPTIONS
         CNTL_ERROR = 1
         CNTL_SYSTEM_ERROR = 2
         CREATE_ERROR = 3
         LIFETIME_ERROR = 4
         LIFETIME_DYNPRO_DYNPRO_LINK = 5.
```

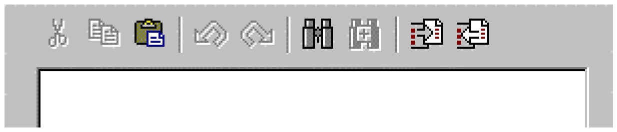
5. In the PBO module, instantiate the textedit control and specify the container control you have just created as the `parent` parameter:

```
   CREATE OBJECT editor
      EXPORTING
         parent = custom_container
         WORDWRAP_MODE = CL_GUI_TEXTEDIT=>WORDWRAP_AT_FIXED_POSITION
         WORDWRAP_POSITION = line_length
         WORDWRAP_TO_LINEBREAK_MODE = CL_GUI_TEXTEDIT=>TRUE.
ENDIF.                "editor is initial
```

6. Activate your main program.

Check Your Work

Start your program. You can now see a text editor at the screen position where you created the container. Nine icons are displayed above the input field of the editor:



If you do not see all these icons on your screen, you must enlarge the container in the *Screen Painter*.

The textedit control provides basic functions for editing text. You can:

- Cut, copy and paste text
- Undo actions
- Search and replace text
- Load and save local files

You can access these functions either via the tool bar or via the context menu (by clicking the right mouse button). Drag and drop functions are also available. You can:

- Load texts by dragging text file icons (no shortcuts) into the text window
- Move text selected in the text window with the right mouse button.

Discussion

Attributes

The **CREATE OBJECT** statement calls the *constructor* of class **cl_gui_textedit**. The constructor determines some initial properties of the instantiated object when it is created. In this exercise, you set two attributes using constants of the class. The **WORDWRAP_MODE** attribute determines if and how words are wrapped. The **WORDWRAP_TO_LINEBREAK_MODE** attribute determines if the system converts automatic wrapping into an actual line break when the text is saved in the R/3 System.

You can display methods (including constructors), attributes, and events of a class in the Class Builder [Ext.] (by using transaction **se24** or choosing *Development → Class Builder* from within the initial screen of the ABAP Workbench).

Exercise 2: Creating a Control and its Container

Time of Creation

All methods that operate on a control must be transferred to the frontend by means of a Remote Function Call (RFC). By calling method `CL_GUI_CFW=>FLUSH`, you can explicitly determine when this takes place.

At the end of each PBO module, this method is **implicitly** called after the screen fields are transported. This ensures that the methods are transferred to the frontend before the screen is set up (see also Synchronization of the Automation Queue [Ext.] in the Control Framework documentation).

The actual parameters of control methods must be valid at the time of transfer. The values of global system variables change frequently since they are also used in methods and function modules called. Consequently, they are not suitable as actual parameters.

This is why the `repid` variable rather than the `sy-repid` system variable was passed in the constructor method after it was assigned the current value of the system field. The `repid` variable is only visible in the main program and thus remains stable.

Lifetime

All statements are included in an `IF` block to ensure that the control is instantiated only once and not each time the PBO module is called.

The lifetime of a control depends on the mode that you can specify in the constructor using parameter `LIFETIME`.

In the default setting (*IMode* mode*)*, controls that you create persist until:

- The internal mode is destroyed in which it was created.
- The container control is destroyed in which it is embedded.
- You call the `FREE` method of the instance created.

For more information on the lifetime of controls, see Lifetime Management [Ext.] in the Control Framework documentation.

Exercise 3: Calling Methods of the Control

Usage

Once you have created an instance of a control, you affect its behavior by means of methods. These methods help you to use the functions of the control and determine its properties. In order to execute control methods, you must normally transfer them to the frontend.

In this exercise, you implement a function that loads a text from an internal table into the text window.

Procedure

1. To generate a text, declare an internal table and a work area globally in your main program. Declare also global flag **g_loaded**:

```
TYPES: begin of mytable_line,
          line(line_length) type C,
          end of mytable_line.
DATA: MYTABLE TYPE TABLE OF MYTABLE_LINE,
      TEXTSTRUCT TYPE MYTABLE_LINE,
      g_loaded TYPE C.
```

The line length of the internal table was intentionally set to the position of the line break in the textedit control (see Exercise 2: Creating a Control and its Container [Page 14]).

2. Define a text element and have your internal table filled with text in the PBO module. As the system should do this only once, include your statements in the **IF** block for creating the control:

```
DO 20 TIMES.
   WRITE TEXT-001 TO TEXTSTRUCT-LINE.
   APPEND TEXTSTRUCT TO MYTABLE.
ENDDO.
```

3. To make the new function accessible, define another pushbutton on screen 100 that has the following properties:
   - Function code: **IMP**
   - Text: **Import**
   - Icon: **ICON_IMPORT**

4. In the PAI module, determine the function code in the existing **CASE** statement, and create subroutine **LOAD_TAB**:

```
WHEN 'IMP'.
      PERFORM LOAD_TAB.
```

5. To load the text of the internal table into the editor, call method **set_text_as_r3table** in your subroutine. Flag **g_loaded** tells you that the text has been loaded:

```
FORM LOAD_TAB.
   call method editor->set_text_as_r3table
              exporting table = mytable
      exceptions
          others = 1.
   if sy-subrc ne 0.
        CALL FUNCTION 'POPUP_TO_INFORM'
             EXPORTING
                  TITEL = repid
                  TXT2 = ' '
                  TXT1  = 'Error in set_text_as_r3table'(600).
   else.
      g_loaded = 'X'.
   endif.
   CALL METHOD CL_GUI_CFW=>FLUSH.
   IF SY-SUBRC NE 0.
      CALL FUNCTION 'POPUP_TO_INFORM'
           EXPORTING
                TITEL = REPID
                TXT2 = sy-subrc
                TXT1  = 'Form LOAD_TAB: Error in FLUSH'(601).
   ENDIF.
ENDFORM.                    " LOAD_TAB
```

6. Activate your new objects, and start your program.

Check Your Work

If you click the **`Import`** pushbutton you created, twenty lines appear that display the text of text element **`TEXT-001`**. If not, you probably have forgotten to create the text element.

In this exercise, you called a control method within a subroutine. For a discussion of this subject, see Exercise 3: Control Methods in Subroutines [Page 39].

Summary

In this lesson, you learned that you use a specific class of controls - referred to as *container* controls - to integrate controls on a screen. Container controls manage one or more control instances and represent the link between the control and the screen.

Depending on the control container type, you must define an area in the *Screen Painter* where you want to place the container. To instantiate an instance of a control, you must use method **`CREATE OBJECT`** to instantiate an object of the class of the control.

All methods that operate on a control are transferred to the frontend by means of a Remote Function Call (RFC). You use method **`FLUSH`** to determine when this is to be done. In this context, you must ensure that all actual parameters are still valid at flush time. System fields must not be passed since they often change at runtime.

Lesson 2: Event Handling

Task

You previously added a function to your editor that allows you to load the contents of an internal table into the text window. Now, you implement a function that converts a text line into a comment line when you double-click the line.

Overview: Objectives

In this lesson, you learn how to:

- Register events for a control that you want to catch
- Assign events caught to event handler methods in the ABAP program
- Write event hander methods and declare them to the control.

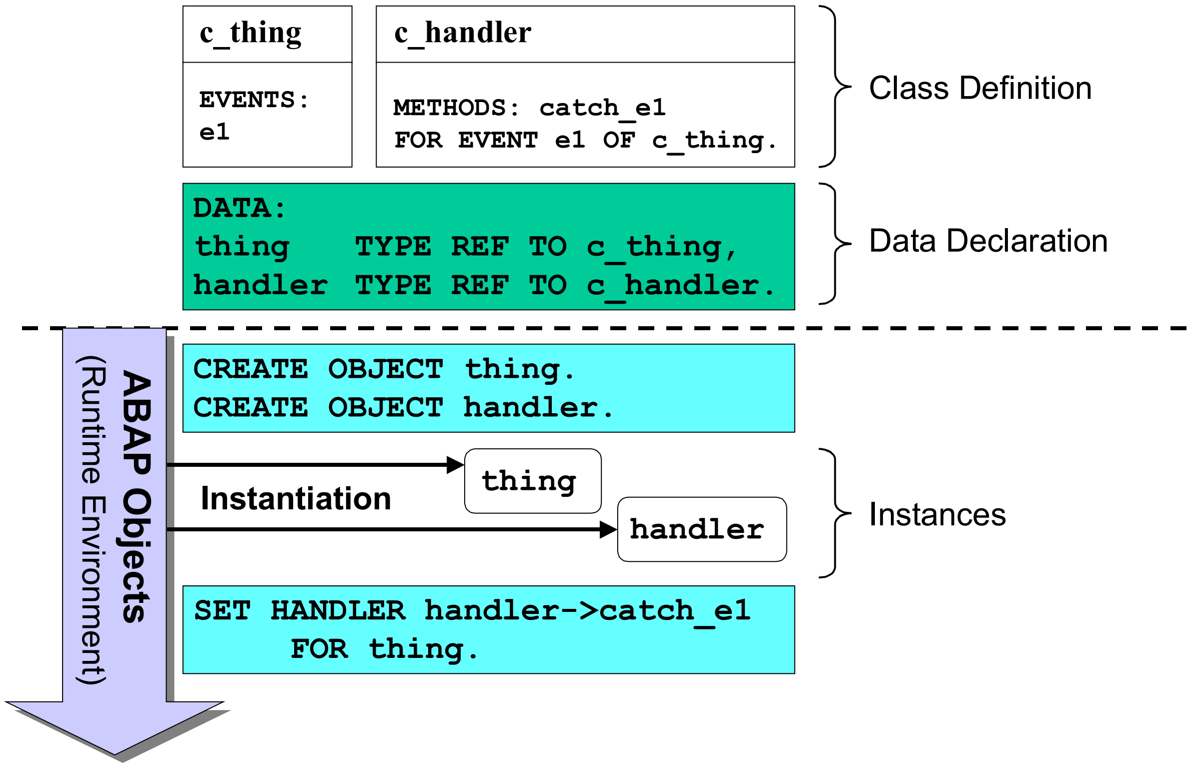
Basics

The event handling of controls is based on that of *ABAP Objects*. Generally, you use events in ABAP Objects to inform objects that a specific state has occurred.

For example, an object may use an event to indicate that:

- The user has clicked an image.
- The user has navigated to a new page in a Web browser.
- The object is ready to output a table on the screen.

The following graphic shows how you can use events in your programs:


c_thing
EVENTS:
e1
c_handler
METHODS: catch_e1
FOR EVENT e1 OF c_thing.
Class Definition
DATA:
thing TYPE REF TO c_thing,
handler TYPE REF TO c_handler.
Data Declaration
CREATE OBJECT thing.
CREATE OBJECT handler.
Instantiation
thing
handler
Instances
SET HANDLER handler->catch_e1
FOR thing.
ABAP Objects
(Runtime Environment)

An event is a component of a class or an interface. Within the methods of the class, you can trigger defined events using **`RAISE EVENT`**.

To be able to respond to an event, you must define a method and link it to the event. It is then sufficient to trigger the event in order to handle it. These methods are also called *event handler methods*. By defining a second class, you can separate these methods from the object that triggers the event. This way, you can also group together handler methods of a specific type in a class.

In the above graphic, the event handler method is defined as an instance method. In this case, you must first instantiate an object of the class to enable the system to call the method. You can also define the method as a static method of the class. In this case, you do not need to generate the **`handler`** object.

Once the objects have been instantiated, you use the **`SET HANDLER`** statement to link the event handling method to the object in which the event is defined. The runtime environment then ensures that the method implemented is automatically called at this event.

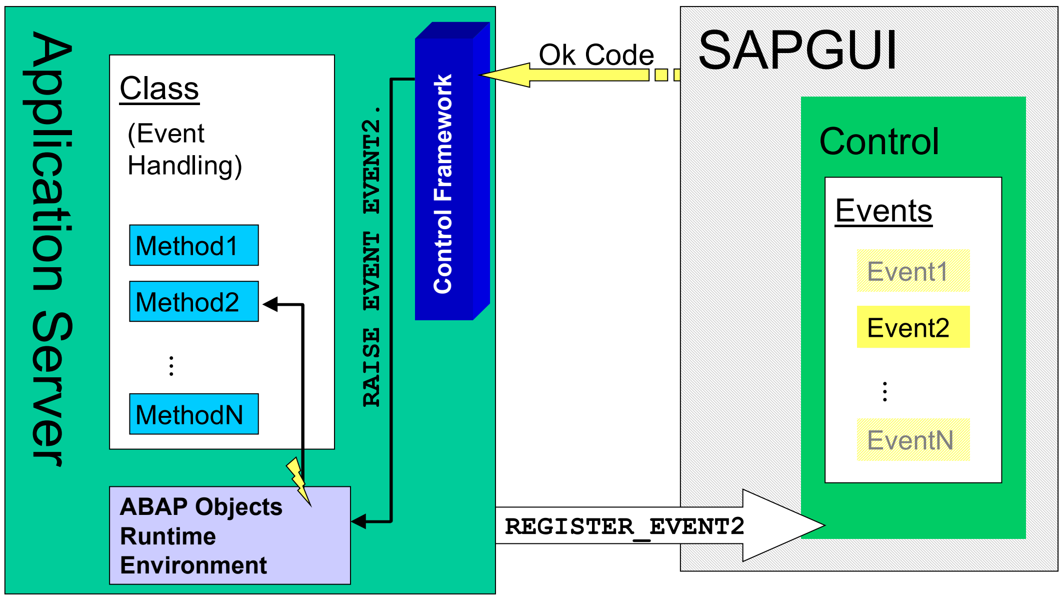
For more information on event handling, see ABAP Objects [Ext.].

To Continue

Lesson 3: Flush Optimization [Page 30] describes how you use flush calls and how methods are buffered before they are transferred to the frontend.

Introduction

R/3 Architecture



Explanation

When the control is set to its initial state, the SAPGUI filters all events triggered by the control on account of user actions. This prevents the connection to the application server from being excessively strained by permanent events such as mouse movements. Events can only be handled if you *register* selected events using methods of the instance created. The standard method used is **`SET_REGISTERED_EVENTS`**.

For information on which events you can catch on the frontend, see the class definition of the relevant control in the Class Builder [Ext.].

Once an event is triggered, the SAPGUI encodes this event in the OK code which is transferred by the frontend to the application server. An event triggered on the frontend first of all has nothing to do with ABAP Objects events. In order to initiate event handling for this event, the Control Framework must assign the event to the relevant instance and trigger the event.

There are two ways of processing the triggered event in the ABAP program:

- The Control Framework triggers the event irrespective of the screen flow logic, that is, without processing the PAI and PBO events. These events are called *system events*. This type of event processing is the default setting.
- The Control Framework processes the PAI module after an event. In this case, you must call method **`CL_GUI_CFW=>DISPATCH`** to initiate the event handling of ABAP Objects. These events are called *application events*.

You must determine if you need system events or application events for your application. See Exercise 2: Registering an Application Event [Page 24] and Exercise 3: Registering a System Event [Page 26] for a discussion of the advantages and disadvantages.

The method that you linked to the event is then called automatically.

Conclusions

To process events triggered by a control, you must:

- Register the events for the control instance to ensure they are not withheld by the SAPGUI.
- Call static method **`DISPATCH`** in the PAI module for application events. This identifies the events transferred for processing and calls the appropriate event handler methods, if they exist.
- Define a class with event handler methods. In the methods, you can get additional event information and affect control behavior.
- Link the events of your control instance to the relevant event handler method. To do this, you use method **`SET HANDLER`**.

Exercise 1: Writing an Event Handler Method

Usage

Control events triggered by the user on the frontend are processed on the backend using ABAP Objects event processing. For each event to which you want to respond you must implement an event handler method.

Procedure

1. Define a local class before the **START-OF-SELECTION** event. Implement a method for this class that is called at the **DBLCLICK** event. This method only displays the event type in a text element:

```
DATA: EVENT_TYPE(20) TYPE C.
CLASS lcl_event_handler DEFINITION.
 PUBLIC SECTION.
   CLASS-METHODS: CATCH_DBLCLICK FOR EVENT DBLCLICK OF CL_GUI_TEXTEDIT.
                             IMPORTING SENDER.
ENDCLASS.

DATA: event_handler TYPE REF TO lcl_event_handler.

CLASS lcl_event_handler IMPLEMENTATION.
   METHOD CATCH_DBLCLICK.
      event_type = text-002.
   ENDMETHOD.
ENDCLASS.
```

Do not forget to define a text such as **doubleclick** for text element **text-002**.

2. Create an input/output field on screen 100 in which the event type can be output:
   - Name: **EVENT_TYPE**
   - Element type: **Input/output field**

3. Link the method to the event. To do this, insert the following line of code in the **IF** block of the PBO module (the *editor* object must already have been created):

```
SET HANDLER lcl_event_handler=>catch_dblclick FOR editor.
```

4. Activate and start your program.

Check Your Work

The input/output field is not updated when you double-click somewhere in the text window. This means that the ABAP Objects event is not yet triggered.

Discussion

The steps in this exercise only represent event handling in ABAP Objects and therefore describe event handling on the backend. Since when the control is set to its initial state all events are filtered by the SAPGUI, the ABAP event for the double-click cannot be triggered on the backend. The reason is that the event has not yet been registered on the frontend (see Exercise 2: Registering an Application Event [Page 24]).

Event Parameter sender

In the definition of the event handler method, event parameter *sender* is imported. You can access this parameter at any ABAP Objects event although it is not listed as an event parameter in the Class Builder. This parameter imports a reference to the instance that has triggered the event. This reference can then be used directly to call methods of this instance.

The Event Handler Method

Defining the Handler Method as a Static Method

In this exercise, you used the **`CLASS`** addition to define the event handler method as a static method of class **`CL_GUI_TEXTEDIT`**. This means that you do not need to instantiate an object of class **`lcl_event_handler`** to be able to call the method.

You use statement **`SET HANDLER`** to link static method **`lcl_event_handler=>catch_dblclick`** to event **`DBLCLICK`** for instance *editor*. You can create additional instances of the textedit control and link them to this method at this event. All instances registered with **`SET HANDLER`** then use the same method that exists only once at runtime.

Event parameter *sender* contains a reference to the triggering instance that you can use to call methods of this instance.

Defining the event handler method as a static method makes sense if all instances of a class should behave identically at an event.

Defining the Handler Method as an Instance Method

Instance methods offer the advantage that you can create more than one object of your local class. This is necessary if you want to retain a state (for example, for a static variable) within the event handler method. If the method is used by several control instances, conflicts may arise if only one instance of the event handler class exists. In such a case, you should use different instance methods for different control instances of the same class.

Exercise 2: Registering an Application Event

Usage

So that you can respond to events, you must first register the events for the control instance and consequently on the frontend. When the control is set to its initial state, the SAPGUI filters all events.

In this exercise, you register event `DBLCLICK` in the textedit control as an application event. Initially, you only want to display a text for the event triggered on the screen.

Procedure

1. Declare the following variables in your main program: a) An internal table for the events you want to register, and b) a structure for one line of this table:

```
DATA events TYPE cntl_simple_events.
DATA wa_events TYPE cntl_simple_event.
```

Insert the following lines of code into the `IF` block of the PBO module after the control is created.

2. Assign the static attribute for event **`DBLCLICK`** to field **`eventid`**:

```
wa_events-eventid = cl_gui_textedit=>event_double_click.
```

3. Using field **`appl_event`**, define the event as an application event:

```
wa_events-appl_event = 'X'.
```

4. Append this line to your internal table events, and pass this table to the textedit control using method **`set_registered_events`**:

```
APPEND wa_events TO events.
CALL METHOD editor->set_registered_events
     EXPORTING events = events.
```

This method is provided by all controls. Depending on the control, there may be additional methods that you can use to register individual events.

5. In the PAI module, call method `DISPATCH` in the last query of the `CASE` statement:

```
WHEN OTHERS.
   CALL METHOD CL_GUI_CFW=>DISPATCH.
```

6. Activate and start your program.

Check Your Work

If you double-click in the text window of the editor, the text of text element **`text-002`** is displayed on the screen.

Discussion

Registering event `DBLCLICK` means that the frontend transfers double-clicks of the user to the backend by means of the OK code. Since you registered this event as an application event, the Control Framework processes the PAI module of the screen concerned. You use method

DISPATCH to determine when the event is triggered by ABAP Objects. The runtime environment then executes method **CATCH_DBLCLICK** and continues processing after the DISPATCH call.

The DISPATCH call is globally valid for all control instances of the main program created.

Advantages

- You determine yourself when the event handler method is to be called. To do this, you use method **DISPATCH**.
- Since the system processes the PAI module, the field transport between the screen and the application server has already taken place. This means you can access screen fields in the event handler method.

Disadvantages

When you use application events, you may encounter problems when you check entries using the **FIELD** or **CHAIN** statement. If entries are not correct, the user may be forced to re-enter the data. However, in doing this, the user may trigger other events in the control. This way, events could be lost.

Using method **DISPATCH** several times does not provide a solution to this problem since the event handler methods are designed to be called only once.

Exercise 3: Registering a System Event

Usage

System events are passed on irrespective of the flow logic of the relevant screen.

In the previous exercise, you have registered event **DLBCLICK** as an application event. You now register this event as a system event.

Procedure

1. Change the value of field **wa_events-appl_event** in the PBO module:

```
wa_events-appl_event = SPACE.
```

2. Comment out the call of method **DISPATCH** in the PAI module.
3. Activate and start your program.

Check Your Work

Double-click in the text window. No text is displayed since the flow logic of the screen and the field transport are ignored.

Discussion

For system events, the PAI/PBO modules are not processed. This has the following consequences:

Advantages

- Method **DISPATCH** is called automatically.
- Events cannot be lost, since the flow logic of the screen is not processed. Field checks and any re-entries that may be required are eliminated and do not conflict with event handling.

Disadvantages

No field transport takes place when the event is triggered. This has the following consequences for accessing screen fields in event handler methods:

- During read access, obsolete values are accessed. To be more precise: The values of the last field transport are accessed.
- Write access to input/output fields is lost completely. During the subsequent field transport, the system overwrites input with the content of the screen field.

This problem does not occur for output-only fields.

This means that as long as you do not access screen fields in read or write mode in an event handler method, this disadvantage has no effect.

Solution

The Control Framework provides method **SET_NEW_OK_CODE**. With this method, you can set an application-specific OK_CODE in an event handler method. After the event handler method, the Control Framework processes the PAI event in which you can access current screen fields in read and write mode.

Modify your program as follows:

1. In the event handler method, comment out the line `event_type = text-002.`.
2. In the event handler method, set application-specific OK code **SHOW**:

```
CALL METHOD cl_gui_cfw=>set_new_ok_code
     exporting new_code = 'SHOW'.
```

3. Read the OK code in the PAI module. To do this, add the following lines to your **CASE** query:

```
when 'SHOW'.
     event_type = 'Doubleclick' (555).
```

4. Activate and start your program.

The PAI event is now processed after the event handler method, and the text is displayed again.

Exercise 4: Responding to an Event

Usage

In the previous exercise, you have registered an event. You have used the **`SET HANDLER`** statement to link the event handler method to the event. To respond to this event, you only need to extend the event handler method.

In this exercise, you implement a function that converts a line into a comment line when the line is double-clicked. If the line is a comment line already, the comment marker at the beginning of the line is deleted. Implement this function by retrieving additional control information in the event handler method and using methods to trigger control actions.

Procedure

Add the following code blocks to method **`catch_dblclick`** before **`set_new_ok_code`** is called:

1. Read the position of the double-click, and define corresponding local variables in advance:

```
DATA: FROM_LINE TYPE I,
        FROM_POS TYPE I,
        TO_LINE TYPE I,
        TO_POS TYPE I.
CALL METHOD SENDER->GET_SELECTION_POS
     IMPORTING
            FROM_LINE = FROM_LINE
            FROM_POS = FROM_POS
            TO_LINE = TO_LINE
            TO_POS = TO_POS.
```

2. Since the text in the text buffer of the editor may have changed, first reload it into your internal table **`mytable`**. Using flag **`g_loaded`**, you can determine if any text at all has been loaded into the text buffer:

```
IF NOT g_loaded IS INITIAL.
   CALL METHOD SENDER->GET_TEXT_AS_R3TABLE
     importing
```

```
            TABLE = MYTABLE.
ENDIF.
```

3. Read the line of the internal table that was clicked. Depending on whether the line is marked as a comment line or not, insert an asterisk or remove it.

```
READ TABLE MYTABLE INDEX FROM_LINE INTO TEXTSTRUCT.
IF SY-SUBRC = 0.
    IF TEXTSTRUCT+0(1) CS '*'.
      SHIFT TEXTSTRUCT.
    ELSEIF TEXTSTRUCT+0(1) NS '*'.
      SHIFT TEXTSTRUCT RIGHT.
      TEXTSTRUCT+0(1) = '*'.
    ENDIF.
```

4. Reload the text of the internal table into the editor.

```
MODIFY MYTABLE FROM TEXTSTRUCT INDEX FROM_LINE.
    CALL METHOD SENDER->SET_TEXT_AS_R3TABLE
       EXPORTING TABLE = MYTABLE
    EXCEPTIONS
       OTHERS = 1.
ENDIF.
```

5. Transfer the methods to the frontend using command **`FLUSH`**.

```
CALL METHOD cl_gui_cfw=>FLUSH.
IF SY-SUBRC NE 0.
      CALL FUNCTION 'POPUP_TO_INFORM'
           EXPORTING
                TITEL = REPID
                TXT2 = sy-subrc
                TXT1  = 'Method CATCH_DBLCLICK: Error in Flush!'(602).
ENDIF.
```

6. Activate and start your program.

Check Your Work

To display the generated text of the internal table in the text window, click on pushbutton `Import`. If you now double-click a line, an asterisk ('`*`') appears at the beginning of that line, and the text is indented to the right. If you double-click the line again, the asterisk is removed.

Discussion

Method **`get_text_as_r3table`** loads the text in the text window of the control into internal table **`mytable`**. In the next step, this table is modified. However, from a chronological point of view, the method should not be executed before flush time. This would mean that the modification is carried out prematurely. Why does the program behave correctly nevertheless?

The reason is that another control is used when large volumes of data are transferred by control methods. This control is the Data Provider. Currently, the Data Provider implicitly triggers another flush to transfer the data.

You can detect any implicit flush in the trace mode. This mode is described in lesson 4 in Exercise 1: Activating the Trace Mode [Page 46].

However, it is not to your disadvantage to define another flush after method **`get_text_as_r3table`**. If a flush is called and no control methods are called before, the Control Framework does not perform an RFC.

SAP recommends you to use an additional flush call and not to rely on an implicit flush. This ensures that your programs are more robust. If the implicit flush is eliminated in later versions of the Data Provider, this will not effect your existing programs.

The next lesson describes in detail how flush calls are used correctly.

Summary

In lesson 2, you learned how to catch and respond to events of a control. To do this, you must:

- Register the event for the SAPGUI. To do this, you use method **`set_registered_events`**.
- Determine if you want to register the event as a *system event* or an *application event*.
- Define an event handler method. To do this, you can define a local class in your program, for example.
- Link the event to the event handler method. To do this, you use statement **`SET HANDLER`**.
- Call static method **`cl_gui_cfw=>DISPATCH`** in the PAI module for application events.

Lesson 3: Flush Optimization

Task

In this lesson, you add a function to the text editor that allows you to write-protect text lines in the editor window.

Overview: Objectives

Optimizing the number of flush calls simply means that you should build as few flush calls as possible into your programs. However, this implies that you must know at which points in your program synchronization is absolutely essential.

In the exercises, we therefore present the 'simple' solution first, discuss it with regard to the flush issue and make suggestions for improvement. If you are aware of the consequences of applying the "as few flush calls as possible" rule, then you can reduce their number.

In this lesson, you learn:

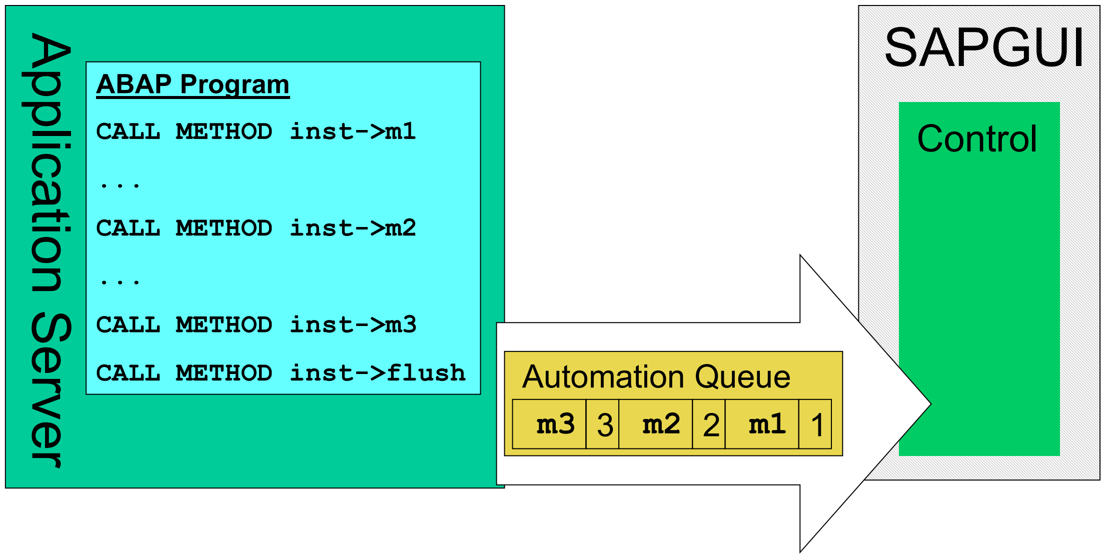
- In which cases errors might occur if flush calls are used wrongly
- How to reduce the number of flush calls
- How the actual parameters of control methods are handled implicitly.

To Continue

Lesson 4: Testing and Error Handling [Page 44] describes what you must consider when troubleshooting control programs.

Introduction

R/3 Architecture



ABAP Program

```
CALL METHOD inst->m1
...
CALL METHOD inst->m2
...
CALL METHOD inst->m3
CALL METHOD inst->flush
```

Explanation

At its interface, the control wrapper class provides methods for operating on the control. However, calling a method does not imply that the method is automatically executed at runtime. Initially, the system buffers methods in a queue, referred to as the *Automation Queue*, when they are called in the ABAP program. The execution sequence of the methods therefore remains unchanged. However, the methods are only executed if they are transferred to the frontend via Remote Function Call (RFC) using method **`FLUSH`**. This means that the Automation Queue is used to reduce the number of RFC calls required.

Due to the buffering of control methods in the Automation Queue, the method calls and the remaining ABAP code are executed at different times. Method **`FLUSH`** thus determines a synchronization point.

For reasons of optimization, the Control Framework may not transfer the Automation Queue although you have called method **`FLUSH`**. This makes sense, for example, if the queue is empty or contains only methods with **`EXPORTING`** parameters. For more information, see Using Controls in the WAN [Ext.] in the Control Framework documentation.

Type of Parameter Passing

Robust control programming depends on the Automation Queue concept. If a method is added to the queue, there are two ways how the Control Framework can save the method parameters:

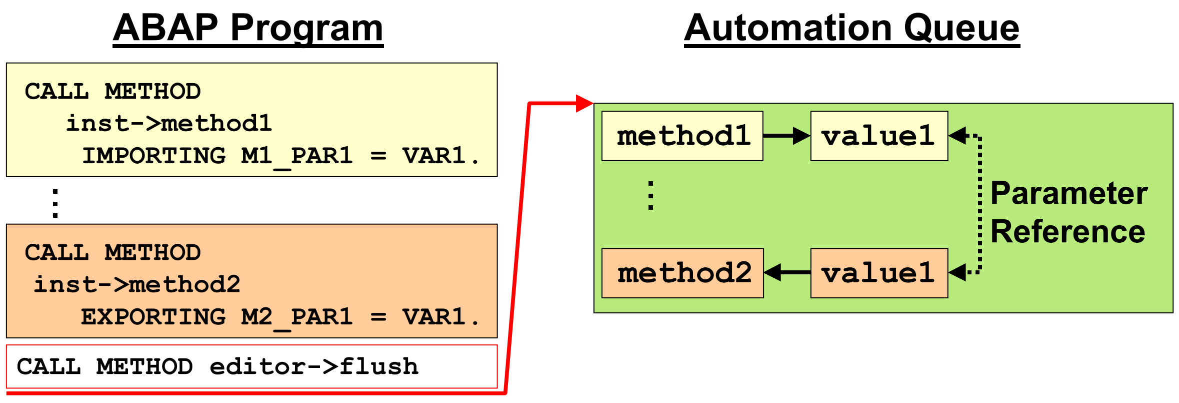
- The values of the actual parameters are added to the queue (call by value).
- The references to the actual parameters are added to the queue (call by reference).

You cannot influence how actual parameters are buffered. In most cases, you pass actual parameters of control methods by reference. The Control Framework then decides on the buffering type using optimization aspects as a basis. Actual parameters declared as value parameters at the method interface are generally passed by value.

Call by Reference

The Framework stores the reference to an actual parameter in the queue if the variable was imported during the method call, that is, returned to the program. If more methods with the same parameter are added to the queue, they are also stored by reference.

The Framework recognizes identical references in the queue and links them by a *parameter reference*:



```
CALL METHOD
   inst->method1
    IMPORTING M1_PAR1 = VAR1.
:
CALL METHOD
 inst->method2
    EXPORTING M2_PAR1 = VAR1.
CALL METHOD editor->flush
```

This means: Values of variables imported in the program during a method call can be exported within the queue during subsequent method calls. It is not necessary to return control to the backend. Due to this mechanism, the number of RFC calls can be reduced, improving performance.

Call by Value

In all other cases, the Control Framework stores **the values** of the actual parameters in the Automation Queue. The value that is passed is determined before the flush call.

Conclusions

An RFC call represents a bottleneck in the connection to the frontend. Consequently, you should keep the number of flush calls in your programs to a minimum.

Since the method calls and the remaining chunks of source code are executed at different times, wrong synchronization might have the following consequences:

- The values of imported variables are not current (which affects conditional queries) or are modified prematurely (so that any changes are overwritten). This is the case if the Automation Queue is processed too late by a flush call.
- A change made to a variable becomes ineffective between two method calls if the first method imports and the second method exports the change. The reason is that the variable is passed within the Automation Queue.

- Actual parameters passed by reference are no longer valid. This is the case if the method was called in a subroutine and the **FLUSH** method was only called after the end of the subroutine.

The exercises in this lesson serve as a basis for discussing the flush issue.

Exercise 1: Using Imported Values

Usage

Before you respond to user actions, you usually check the current status of the control. To do this, you use control methods that pass values to your program. If you change or read these values, you must be sure to use the current values after the method call. This depends on the correct flush time.

In this exercise, you implement a function that allows the user to protect selected lines. To do this, you determine the selected area using method **GET_SELECTION_POS** and then protect this area using method **PROTECT_LINES**.

Procedure

1. Create a pushbutton on your screen with the following properties:

- Function code: **PROTECT**
- Icon name: **ICON_LOCKED**

In the next steps, you add more lines to the **CASE** block in the **PAI** module:

2. Declare global variables for the selection area, and a run index:

```
WHEN 'PROTECT'.
      DATA: FROM_IDX TYPE I,
            TO_IDX TYPE I,
            INDEX TYPE I.
```

3. Determine the area selected by the user with the mouse:

```
CALL METHOD EDITOR->GET_SELECTION_POS
        IMPORTING
          FROM_LINE = FROM_IDX
          TO_LINE = TO_IDX
```

```
exceptions
          ERROR_CNTL_CALL_METHOD = 1.
```

4. Synchronize execution in the control with your ABAP program:

```
CALL METHOD cl_gui_cfw=>FLUSH.
IF SY-SUBRC NE 0.
      CALL FUNCTION 'POPUP_TO_INFORM'
           EXPORTING
                TITEL = REPID
                TXT2 = sy-subrc
                TXT1  = 'PAI USER_COMMAND_100(1): Error in
Flush!'(603).
ENDIF.
```

Without this synchronization, variables **FROM_IDX** and **TO_IDX** have obsolete values in the following **IF** query (initial value of both is 0).

5. Protect the lines selected by means of a loop. To do this, add one method call for each line to the Automation Queue and flush them afterwards:

```
LOOP AT MYTABLE INTO TEXTSTRUCT.
      IF ( SY-TABIX >= FROM_IDX AND SY-TABIX <= TO_IDX ).
        INDEX = SY-TABIX.
        CALL METHOD EDITOR->PROTECT_LINES
            EXPORTING
               FROM_LINE = INDEX
               TO_LINE   = INDEX
                 ENABLE_EDITING_PROTECTED_TEXT = cl_gui_textedit=>true.
        endif.
ENDLOOP.
CALL METHOD cl_gui_cfw=>FLUSH.
IF SY-SUBRC NE 0.
      CALL FUNCTION 'POPUP_TO_INFORM'
           EXPORTING
                TITEL = REPID
                TXT2 = sy-subrc
                TXT1  = 'PAI USER_COMMAND_100(2): Error in
Flush!'(604).
ENDIF.
```

Using method **PROTECT_LINES** is rather naive. It would be easier to just use a method call where you pass **from_idx** and **to_idx**. The example is designed to make you aware of the special effects involved in control programming.

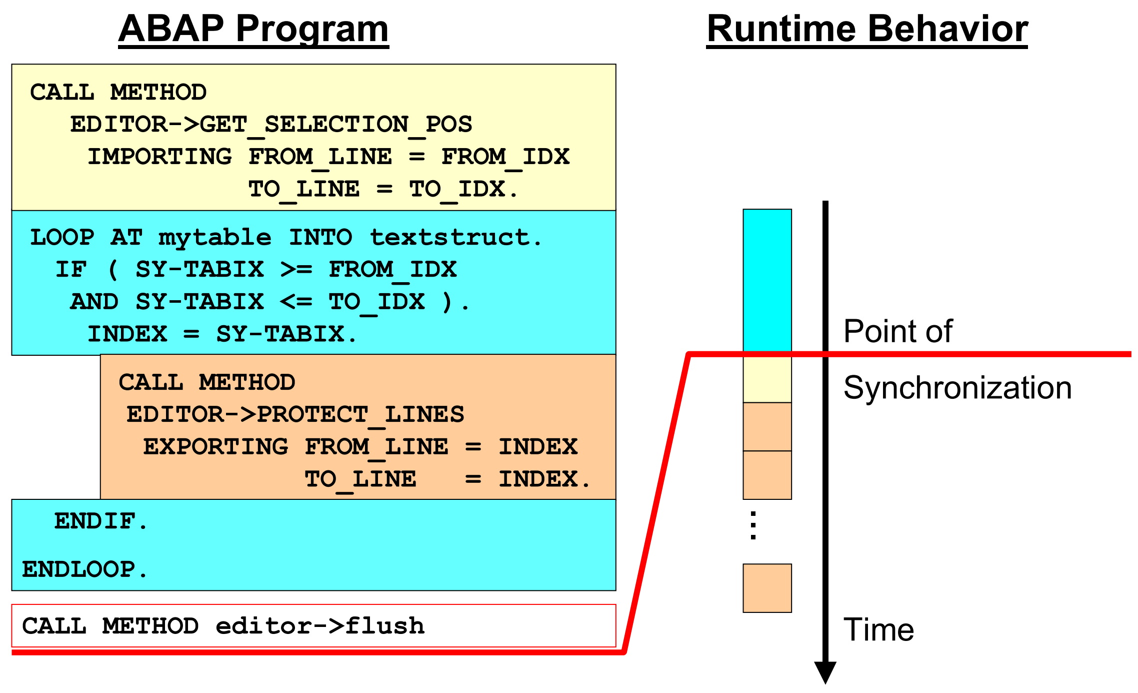
6. Activate your new objects, and start your program.

Check Your Work

Import your internal table by choosing the pushbutton you created in a previous exercise. If you place your cursor on a line and click your new pushbutton, the line is highlighted in gray and can no longer be modified. Using the left mouse button, you can select an entire area to write-protect multiple lines.

Discussion

Synchronization after method call **`GET_SELECTION_POS`** is absolutely essential for this solution. As a test, comment out the **`FLUSH`** statement and check the values of the **`from_idx`** and **`to_idx`** variables in the debugger. The runtime behavior is as follows (due to space restrictions, the graphic does not show parameter **`ENABLE_EDITING_PROTECTED_TEXT`** of method **`PROTECT_LINES`**):



```
CALL METHOD
   EDITOR->GET_SELECTION_POS
    IMPORTING FROM_LINE = FROM_IDX
                TO_LINE = TO_IDX.
LOOP AT mytable INTO textstruct.
  IF ( SY-TABIX >= FROM_IDX
   AND SY-TABIX <= TO_IDX ).
    INDEX = SY-TABIX.
      CALL METHOD
       EDITOR->PROTECT_LINES
        EXPORTING FROM_LINE = INDEX
                    TO_LINE    = INDEX.
  ENDIF.
ENDLOOP.
CALL METHOD editor->flush
```

If there is no **FLUSH** after **GET_SELECTION_POS** is called, the values of **from_idx** and **to_idx** are checked prematurely. When the PAI module is processed for the first time, both variables are initial. Due to the loop condition, no **PROTECT_LINES** method call is added to the Automation Queue. If the function is called again afterwards, the loop condition is always checked against obsolete values. In this case, although lines are protected, these are only those of the most recent call.

You can eliminate the second flush in this exercise, if you use method **PROTECT_SELECTION**. This way, you do not need to read the line numbers of the selected area.

Values in the Automation Queue

There is no method in this example that is called before **PROTECT_LINES** in the same queue and imports actual parameter **INDEX**. This is why the Control Framework buffers the actual parameters of the methods as values in the queue. The next exercise provides an example where the Framework stores actual parameter references in the Automation Queue.

Exercise 2: Value Passing Between Methods

Usage

Method parameters that are passed by reference (call by reference) can be passed on in the Automation Queue. Thus methods can use return values of preceding methods in the queue.

For this exercise, we have designed an example in which this option is **not** desired. You implement a query that disallows users to protect the first visible line in the text window. If they attempt to do this, users should get an error message.

Procedure

In the PAI module, insert the following lines of code before the **LOOP AT mytable INTO textstruct** line, and create the text element with an appropriate error message:

```
CALL METHOD EDITOR->GET_FIRST_VISIBLE_LINE
         IMPORTING
              LINE = INDEX
         EXCEPTIONS
```

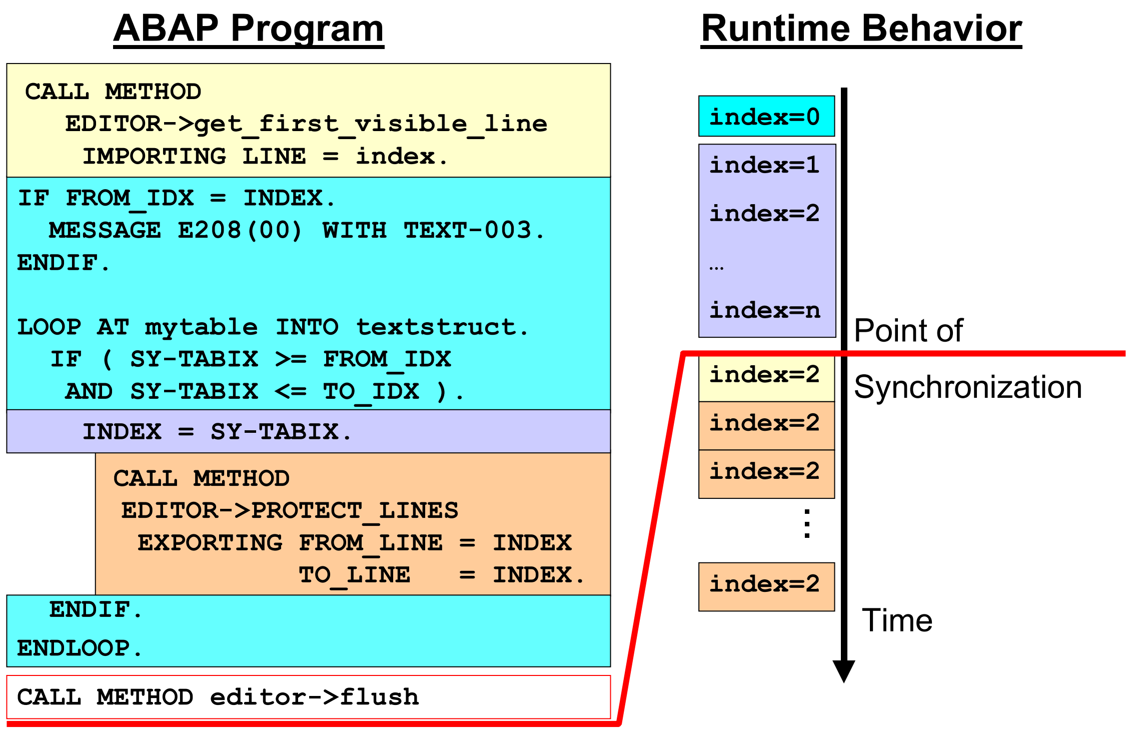
```
              ERROR_CNTL_CALL_METHOD = 1.
IF FROM_IDX = INDEX.
     MESSAGE i208(00) WITH TEXT-003.
     EXIT.
ENDIF.
```

Check Your Work

Start your program, import the internal table and select any area you like on the text window. However, rather than displaying an error message, the system protects the first visible line.

Discussion

There are two reasons for this wrong behavior: First, the **INDEX** variable is used multiple times, and second, the methods are executed at different times (due to space restrictions, the graphic does not show parameter **ENABLE_EDITING_PROTECTED_TEXT** of method **PROTECT_LINES**):



ABAP Program

```
CALL METHOD
   EDITOR->get_first_visible_line
    IMPORTING LINE = index.
IF FROM_IDX = INDEX.
  MESSAGE E208(00) WITH TEXT-003.
ENDIF.

LOOP AT mytable INTO textstruct.
  IF ( SY-TABIX >= FROM_IDX
   AND SY-TABIX <= TO_IDX ).
    INDEX = SY-TABIX.
      CALL METHOD
       EDITOR->PROTECT_LINES
        EXPORTING FROM_LINE = INDEX
                  TO_LINE   = INDEX.
  ENDIF.
ENDLOOP.
CALL METHOD editor->flush
```

Reading the Imported Value

The fact that the error message fails to appear has the following reason: In the **IF** query, **INDEX** still has its initial value of 0 since the **get_first_visible_line** method is only executed at flush time. This is why the error message is not displayed. Any time the user retries to protect lines, **INDEX** always has the value of the most recent **CASE** branch processing run. This effect has been described in the previous exercise.

Passing Parameters Within the Automation Queue

The fact that the wrong lines are protected has the following reason: In the loop, one or more **PROTECT_LINES** method calls are added to the Automation Queue, depending on the values of the **FROM_IDX** and **TO_IDX** variables. The actual parameters are stored as references since the **get_first_visible_line** method has previously imported the same **INDEX** parameter. At runtime, **INDEX** is then passed on to the actual parameters of the **PROTECT_LINES** method.

The Automation Queue does not adopt the reference parameter values until flush time. This is why the **INDEX = SY-TABIX** assignment does not have any practical effect. In addition, the actual parameters of method **PROTECT_LINES** are overwritten after **get_first_visible_line** returns the value of **LINE**.

Generally, the capability of the Automation Queue to pass on reference parameters reduces the number of flush calls. We recommend that you use this feature as comprehensively as possible to enhance performance.

Solution

If you do not want to pass on the return value of a method within the Automation Queue, you can:

- Introduce another variable, or
- Use another flush.

The **get_selection_pos** and **get_first_visible_line** method calls are independent of each other. Therefore, add both to the Automation Queue. To do this, move the flush call that comes after the **GET_SELECTION_POS** method before the **IF** block for displaying the message.

Then, the selected lines are protected, and the error message created is displayed as a dialog box for the first visible line.

Exercise 3: Control Methods in Subroutines

Usage

You can also call the methods of a control in a subroutine or in a method. This way, you can isolate a specific function and improve the program structure.

In this exercise, you combine method calls for checking the control status in a subroutine.

Procedure

1. In the PAI module, call subroutine **`get_lines`** in branch **`CASE`** in which you determine function code **`PROTECT`**. Pass parameters **`from_idx`**, **`to_idx`** and **`index`** by reference:

```
WHEN 'PROTECT'.
   DATA: FROM_IDX TYPE I,
         TO_IDX TYPE I,
         INDEX TYPE I.
   PERFORM GET_LINES CHANGING FROM_IDX TO_IDX INDEX.
```

2. Create the subroutine, and move the **`GET_SELECTION_POS`** and **`GET_FIRST_VISIBLE_LINE`** method calls into this subroutine.

   Ensure that the formal parameters of **`get_lines`** are identical to the actual parameters of the methods copied.

Check Your Work

The behavior of your program does not change, but the code structure of your program has improved.

Discussion

Methods **`GET_SELECTION_POS`** and **`GET_FIRST_VISIBLE_LINE`** import values into actual parameters **`from_idx`**, **`to_idx`** and **`index`**. The Control Framework stores references to these parameters when it buffers the methods in the Automation Queue.

Reference Parameters of the Subroutine

The values are assigned to the actual parameters at flush time. In our example, this does not take place within the subroutine. This is rather astonishing since the actual parameters of the methods must be valid at flush time. However, the reference of a parameter in the subroutine turns invalid after the program has ended. Nevertheless, no runtime error occurs.

The reason is that the reference parameters of subroutines are passed on to the Automation Queue (if **`USING`** and **`CHANGING`** are used). This means that the references are still known when the Automation Queue is flushed.

Using global variables as actual parameters of control methods is another feasible solution. However, you are not recommended to do this since programs with global variables are more complex.

Value Parameters of Subroutines

Actual parameters **from_idx**, **to_idx** and **index** are reused in the PAI module after the subroutine has been processed. In our example, we could therefore not pass the subroutine parameters by call by value - in this case, the return values of methods **GET_SELECTION_POS** and **GET_FIRST_VISIBLE_LINE** would be lost.

As a test, we will nevertheless pass one of these parameters by value. To do this, change the form definition as follows:

```
FORM GET_LINES CHANGING VALUE(FROM_IDX) TO_IDX INDEX.
```

Start your program and try to protect a line. The program terminates with a runtime error. The reason is that method **GET_SELECTION_POS** uses local variable **FROM_IDX** in the form which is not valid at flush time - that is, outside the form.

This problem occurs only if the actual parameters of the control methods are stored as references in the Automation Queue.

This means: If you use use local variables in a subroutine that are held as references in the Automation Queue, a flush call is required in the subroutine.

Exercise 4: Using Multiple Controls

Usage

You can create multiple controls of the same type or of a different type in your program. In this exercise, you create a second editor window that is designed to be used as a clipboard for short texts.

For a detailed description on creating controls, see Exercise 2: Creating a Control and its Container [Page 14] in the first lesson.

Procedure

1. Create a new reference variable `scratch` to the class of the textedit control, and reference variable `custom_container2` for the custom container control in your main program:

```
DATA: scratch TYPE REF TO cl_gui_textedit,
     custom_container2 TYPE REF TO cl_gui_custom_container.
```

2. Create a new container `MYCONTAINER2` for a custom control on your screen.
3. Create the control (including the custom container control) at PBO time and hide the status bar. To do this, insert the following lines of code into `IF` block `IF EDITOR IS INITIAL`:

```
   CREATE OBJECT custom_container2
      EXPORTING
         CONTAINER_NAME = 'MYCONTAINER2'
      EXCEPTIONS
         CNTL_ERROR = 1
         CNTL_SYSTEM_ERROR = 2
         CREATE_ERROR = 3
         LIFETIME_ERROR = 4
         LIFETIME_DYNPRO_DYNPRO_LINK = 5.

   CREATE OBJECT SCRATCH
      EXPORTING
         parent = custom_container2
         WORDWRAP_MODE = CL_GUI_TEXTEDIT=>WORDWRAP_AT_WINDOWBORDER
         WORDWRAP_TO_LINEBREAK_MODE = CL_GUI_TEXTEDIT=>TRUE.
   CALL METHOD SCRATCH->SET_STATUSBAR_MODE
      exporting
         STATUSBAR_MODE = CL_GUI_TEXTEDIT=>FALSE.
```

4. Activate your new objects, and start your program.

Check Your Work

In the second container, the system displays a text editor without a status bar. Using the *Copy* function of the first editor and the *Paste* function of the second editor, you can copy text into the second window.

Discussion

In the PBO module, you create multiple controls and call additional control methods. Transferring methods to the frontend only requires synchronization at PBO end. This means the methods of two different instances use the same Automation Queue.

Basically, there is one Automation Queue for each internal mode. In this queue, the system buffers method calls of all control instances that were created in the same internal mode.

Theoretically, you can buffer as many method calls in the queue as you like. However, the system limits the runtime for RFC calls. If this limit is exceeded, you must introduce additional points of synchronization, since otherwise the backend terminates the connection. Generally, this is not due to the number of RFC calls, but to the data volume to be transferred in such a call (long texts, for example).

Strategy

Especially if you use multiple controls, you can apply a certain strategy to reduce the number of flush calls. In many cases, you must retrieve the control properties with **`GET`** methods before you can determine control behavior with **`SET`** methods. Since you need the current values in your ABAP program after a **`GET`** method, a flush is indispensable (see also Exercise 1: Using Imported Values [Page 34]). In case of several **`GET`** calls independent of each other, you best proceed as follows:

1. Bundle all **`GET`** calls in your source code.
2. Call method **`FLUSH`** to transfer the methods to the frontend, and import the values of the **`GET`** calls.
3. Read these values, and call all **`SET`** methods in a second block.

Grouping calls thus allows you to optimize performance.

Summary

In this lesson, you have learned that the control methods are buffered in an Automation Queue. Basically, the queue has the following properties:

- The Control Framework transfers the queue to the frontend if you call method **`CL_GUI_CFW=>FLUSH`**.
- There is one Automation Queue for each internal mode. This means that you can buffer methods of different control instances in a queue.
- The Control Framework stores the actual parameters as references or values in the Automation Queue.
- Actual parameters are stored as references in the queue if the method has return values or if the actual parameter was returned before in a previous call of the queue. In all other cases, the Framework stores the actual parameters as values in the queue.

The last two points are important for determining the correct flush time and reusing variables that are used as actual parameters:

- If you have used variables for the return values of control methods, you should only reuse these as actual parameters for the succeeding control methods.
- Do not use system variables as actual parameters.
- If you determine control values in subroutines, you must absolutely perform a flush when you use local variables. The following table gives you a short summary:

Essential Synchronization Before Exiting Subroutines

Type of storage in Automation Queue:	**Local variable in subroutine** **or** **Pass by value()**	**Global variable in subroutine** **or** **Pass by reference**
As value	Flush not absolutely required.	Flush not absolutely required.
As reference	Flush absolutely required before subroutine is exited and before EXIT.	Flush not absolutely required.

Lesson 4: Testing and Error Handling

Task

Display the methods that are transferred in the Automation Queue.

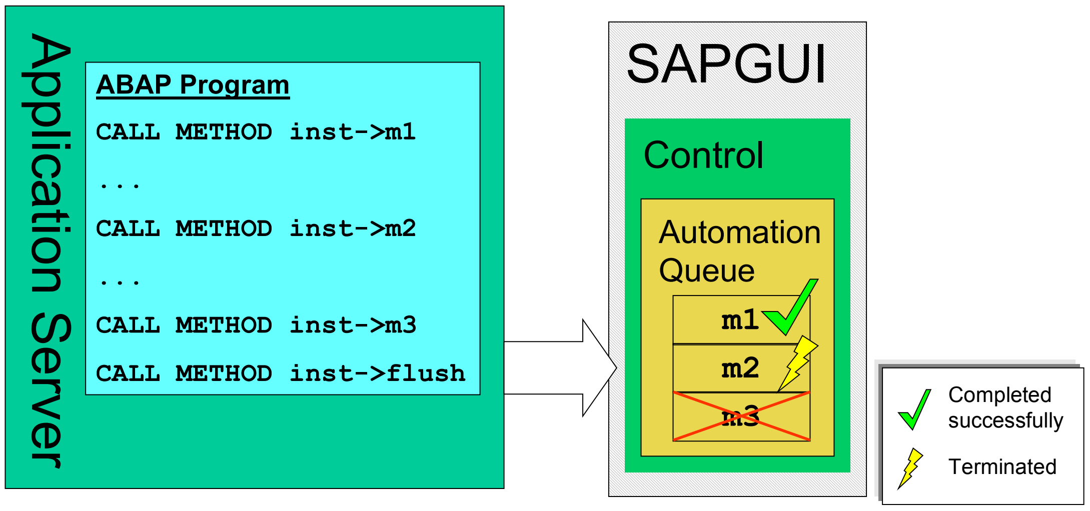
Overview: Objectives

In this lesson you learn what you must be aware of when debugging programs that use controls, and which features and options the ABAP Workbench provides for troubleshooting.

See also Automation Queue Services [Ext.] in the Control Framework documentation.

Introduction

R/3 Architecture



Explanation

After the Control Framework has transferred the methods via Remote Function Call (RFC) to the SAPGUI, they are executed one after the other. If a method causes a runtime error on the frontend, method processing terminates. This means that both the method that caused the error and all subsequent methods stop executing.

From the backend point of view, an error does not occur until the **FLUSH** method is called since the control methods are only buffered but not executed before.

The section on Error Handling in Case of Synchronization [Ext.] in the Control Framework documentation provides more detailed information on error handling in connection with **implicit** flush calls.

Conclusion

If a flush call triggers an exception, it is not yet obvious which method in the Automation Queue caused the error. This is why you cannot catch these runtime errors dynamically and respond to them. At flush time, you only know that an error has occurred, but not where it occurred.

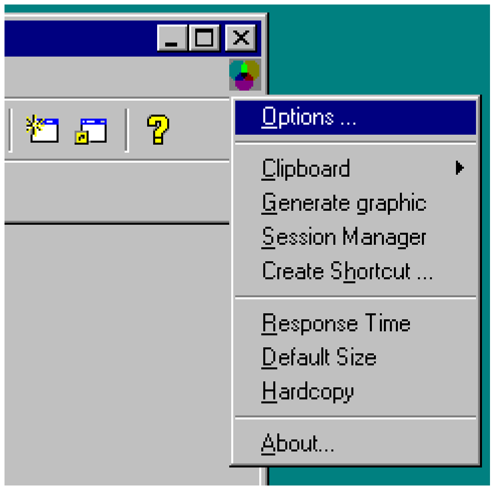
Exercise 1: Activating the Trace Mode

Usage

To keep track of the method calls added to the Automation Queue, you can use the *trace mode*. All Automation calls are written to a file with their parameters and result values and displayed on a dialog box. For example, this helps you analyze which queue method called has caused an error.

Procedure

1. Choose *Options* in the configuration menu:



The system displays a screen with tabs for the general options.

2. Choose the *Trace* tab.
3. To set the trace mode, choose *Disable* first.
4. Select the *Automation* checkbox, and choose a display level using one of the *Trace Level* radio buttons.

   If you choose level 1, the system displays all Automation calls including parameters and result values. If you choose level 2, the tables of the Automation Queue are additionally displayed.

5. Enter a name for the trace file where you want to save the information.

   The file is created on your local PC in directory `C:\sappcadm`.

6. To track trace information during program execution, select *Show Window*.
7. Choose *Enable*.

The system opens an additional window to display Automation calls.

8. Save your data by choosing *OK*.

Check Your Work

Start your program. In the trace window, you see how your method calls are passed on via the Automation Queue to the Automation Controller. If you have specified a file name, you can find the same text there.

Transfer of the methods listed is indicated by a horizontal line in the trace run. This means that a flush has been triggered for these methods. You can also use the trace run for flush optimization.

Exercise 2: Synchronous Method Processing

Usage

As a result of the delay between control method call and control method execution, any error on the frontend only occurs at flush time. However, if synchronization takes place after each control method, you can determine which method terminates processing.

The debugger settings allow you to enable automatic synchronization after each control method. In debugging mode, the system then executes a flush call after each control call. If a runtime error occurs, the call is displayed that caused the error.

Yet the debugger setting for synchronous method processing is not only relevant to determining the termination point. If you track the flow of your program in the debugger without synchronous processing, the control methods are only buffered, but not yet executed. As a result, you lose Sy-Subrc. What is more, variables imported through the methods do not yet have current values.

In some cases, your program may not run correctly with synchronous processing. Frequently, the reason is a missing explicit flush call.

Procedure

1. In the ABAP Debugger, choose *Settings*.
2. In the debugger settings section, select *Automation Controller: Always process requests synchronously*.
3. Activate the trace mode (see Exercise 1: Activating the Trace Mode [Page 46]).

Check Your Work

In the trace run, you see that a flush is performed after each control method.

Summary

The ABAP Workbench provides the following capabilities for debugging programs that use controls:

- The trace mode allows you to keep track of the control methods transferred in the Automation Queue, and check if they were executed successfully. The synchronization points are identified by a horizontal line.
- By setting an indicator in the debugger, you can process Automation calls synchronously. If a runtime error occurs, the relevant control method is specified as the termination point.

Additional Sources of Information

General documentation:

- SAP Control Framework [Ext.]
- SAP Container [Ext.]

Class documentation for Basis controls:

- SAP HTML Viewer Control [Ext.]
- SAP Picture Control [Ext.]
- SAP TextEdit Control [Ext.]
- SAP Tree Control [Ext.]