Oracle Developer: Build Forms I

Volume 2 • Student Guide

43112GC10
Production 1.0
April 1999
M08602

ORACLE®

Authors

Fergus Griffin
Ellen Gravina

Technical Contributors and Reviewers

Grant Anderson
David Ball
Christian Bauwens
Ruth Delaney
Brian Fry
Tushar Gadhia
Danae Hadjioannou
Daniel Maas
Jayne Marlow
Stella Misiulis
Mark Sullivan

Publisher

Tommy Cheung

Copyright © Oracle Corporation, 1999. All rights reserved.

This documentation contains proprietary information of Oracle Corporation. It is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited. If this documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions for commercial computer software and shall be deemed to be Restricted Rights software under Federal law, as set forth in subparagraph (c) (1) (ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

This material or any portion of it may not be copied in any form or by any means without the express prior written permission of the Worldwide Education Services group of Oracle Corporation. Any other copying is a violation of copyright law and may result in civil and/or criminal penalties.

If this documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with “Restricted Rights,” as defined in FAR 52.227-14, Rights in Data-General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them in writing to Education Products, Oracle Corporation, 500 Oracle Parkway, Box 659806, Redwood Shores, CA 94065. Oracle Corporation does not warrant that this document is error-free.

Oracle Developer, Oracle Server, and PL/SQL are trademarks or registered trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Appendix A: Practice Solutions

Appendix B: Table Descriptions and Data

Appendix G: Using the Layout Editor in Oracle Developer

10

Creating Windows and Content Canvases

Objectives

After completing this lesson, you should be able to do the following:

- **Describe windows and content canvases**
- **Describe the relationship between windows and content canvases**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Objectives

- **Identify window and content canvas properties**
- **Display a form module in multiple windows**
- **Display a form module on multiple layouts**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Introduction

Overview

With Oracle Developer you can take advantage of the GUI environment by displaying a form module across several canvases and in multiple windows. This lesson familiarizes you with the window object and the default canvas type, the content canvas.

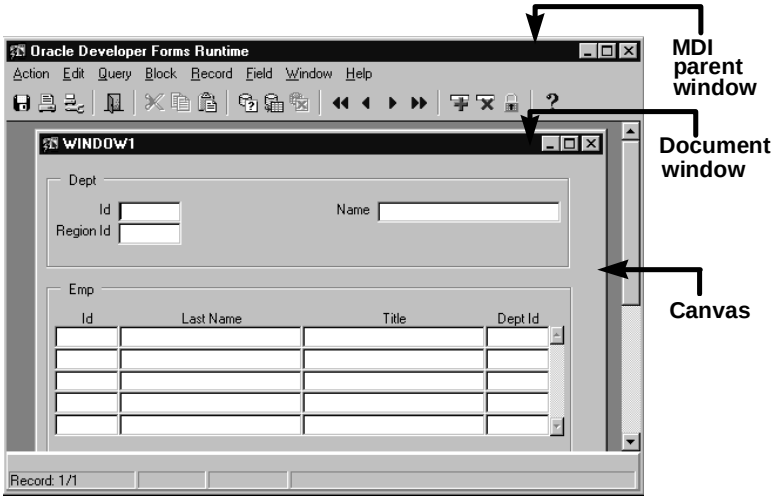
Windows and Canvases

- **Window: Container for Form Builder visual objects**
- **Canvas: Surface on which you "paint" visual objects**

  To see a canvas and its objects, display the canvas in a window.

Copyright © Oracle Corporation, 1999. All rights reserved.

Window, Canvas, and Viewport



Copyright © Oracle Corporation, 1999. All rights reserved.

Windows and Content Canvases

With Form Builder you can display an application in multiple windows by using its display objects—windows and canvases.

What Is a Window?

A *window* is a container for all visual objects that make up a Form Builder application. It is similar to an empty picture frame. The window manager provides the controls for the window that enable such functionality as scrolling, moving, and resizing. You can minimize a window.

A single form may include several windows.

What Is a Canvas?

A *canvas* is a surface inside a window container on which you place visual objects such as interface items and graphics. It is similar to the canvas upon which a picture is painted. To see a canvas and its contents at run time, you must display it in a window. A canvas always displays in the window to which it is assigned.

Note: Each item in a form must refer to no more than one canvas. An item displays on the canvas to which it is assigned, through its Canvas property. Recall that if the Canvas property for an item is left unspecified, that item is said to be a Null-canvas item and will not display at runtime.

What Is a Viewport?

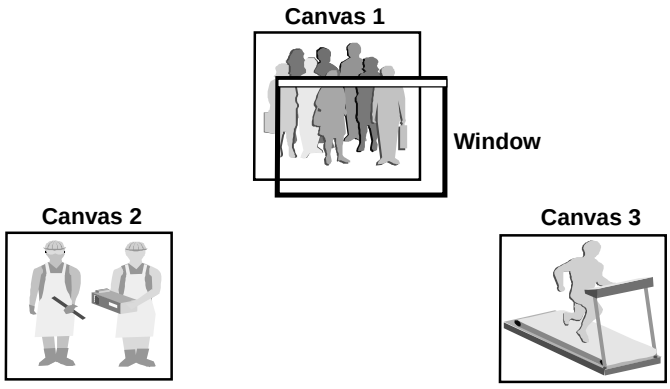
A *viewport* is an attribute of a canvas. It is effectively the visible portion of, or view onto, the canvas.

Content Canvas

- **"Base" canvas**
- **View occupies entire window**
- **Default canvas type**
- **Each window should have at least one content canvas**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Windows and Content Canvases



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Note

At run time, only one content canvas can display in a window at a time. However, you can assign multiple content canvases to a window.

What Is a Content Canvas?

Form Builder offers different types of canvases. A *content canvas* is the base canvas that occupies the entire content pane of the window in which it displays. The content canvas is the default canvas type. Most canvases are content canvases.

The Relationship Between Windows and Content Canvases

You must create at least one content canvas for each window in your application. When you run a form, only one content canvas can display in a window at a time, even though more than one content canvas can be assigned to the same window at design time.

At run time, a content canvas always completely fills its window. As the user resizes the window, Form Builder resizes the canvas automatically. If the window is too small to show all items on the canvas, Form Builder automatically scrolls the canvas to bring the current item into view.

Windows

- **WINDOW1:**
  - **It is created by default with each new form module.**
  - **It is modeless.**
  - **You can delete, rename, or change its attributes.**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Windows

- **Use additional windows to:**
  - **Display two or more content canvases at once**
  - **Modularize form contents**
  - **Switch between canvases without replacing the initial one**
  - **Take advantage of the window manager**
- **Two types of windows:**
  - **Modal**
  - **Modeless**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Displaying a Form Module in Multiple Windows

When you create a new form module, Form Builder creates a new window implicitly. Thus, each new form module has one predefined window, which is called WINDOW1. You can delete or rename WINDOW1, or change its attributes.

Uses and Benefits of a New Window

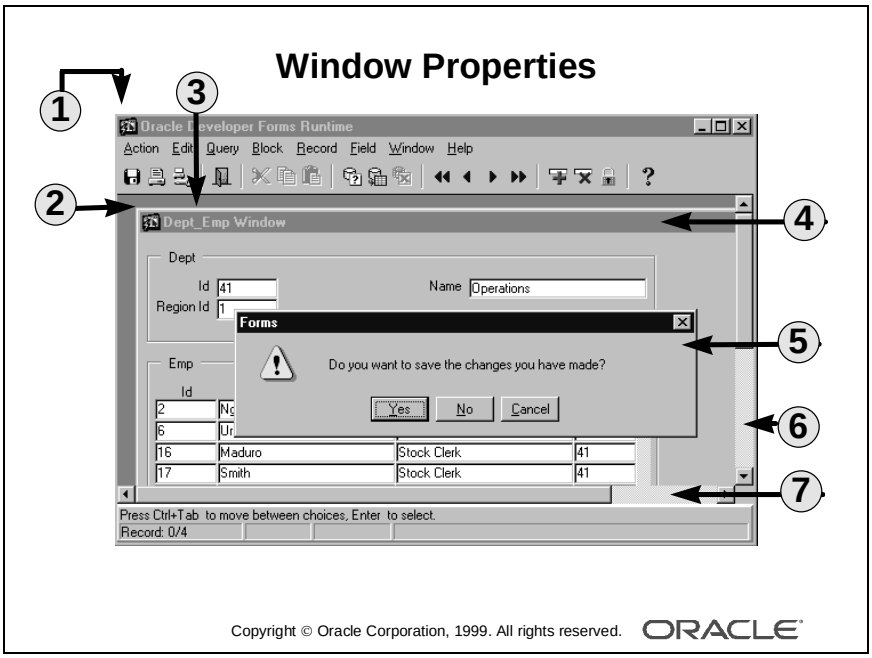
You can create additional windows in which to display your form application. A new or second window provides the ability to do the following:

- Display two or more content canvases at once.
- Modularize the form contents.
- Switch between canvases without replacing the initial one.
- Take advantage of window manager functionality such as minimizing.

Window Types

You can create two different window types: modal and modeless.

- A *modal window* is a restricted window that the user must respond to before moving the input focus to another window. Modal windows:
  - Must be dismissed before control can be returned to a modeless window
  - Become active as soon as they display
  - Require a means of exit or dismissal
- A *modeless window* is an unrestricted window that the user can exit freely. Modeless windows:
  - Can display many at once
  - Are not necessarily active when displayed
  - Are the default window type



1	MDI parent window
2	X/Y position
3	Title
4	Document window
5	Dialog window
6	Show vertical scroll bar
7	Show horizontal scroll bar

Window Properties

Physical Property	Function
X Position	Determines the X coordinate for the window
Y Position	Determines the Y coordinate for the window
Width	Determines the width of the window
Height	Determines the height of the window
Bevel	Determines how the window border displays
Show Horizontal Scrollbar	Determines whether a horizontal scroll bar displays in the window
Show Vertical Scrollbar	Determines whether a vertical scroll bar displays in the window

Functional Property	Function
Title	Specifies a window title to appear in the title bar
Primary Canvas	Specifies the name of the canvas to display in this window when it is invoked programmatically
Window Style	Determines whether the window style is Document or Dialog (Document style windows are fixed and always remain within the application window frame. Dialog style windows are free floating and can be moved outside the application window frame.)
Modal	Determines whether the window is modal (requires user response) or modeless (does not require user response)
Hide on Exit	Specifies whether a modeless window is hidden automatically when the end user navigates to an item in another window
Icon Filename	Specifies the icon resource name that depicts the minimized window

Note: If you do not specify a window title, Form Builder uses the window object name specified in the Name property for the title.

The canvas you choose as the primary canvas must be a content canvas.

The X and Y Position (0,0) of a window is relative to the top left corner of the screen when you set the Window Style to dialog. If you set the Window Style to document, the X and Y Position (0,0) is relative to the top-left corner of the MDI window.

GUI Hints

- **GUI hints are recommendations to the window manager about window appearance and functionality.**
- **If the window manager supports a specific GUI Hint and its property is set to Yes, it will be used.**
- **Functional properties for GUI Hints:**
  - **Close Allowed**
  - **Maximize Allowed**
  - **Move Allowed**
  - **Minimize Allowed**
  - **Resize Allowed**
  - **Inherit Menu**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

What Are GUI Hints?

GUI Hints are recommendations to the window manager about the window appearance and functionality. There are certain properties under the Functional group that enable you to make these recommendations. If the current window manager supports the specific GUI Hint property and it is set to Yes, then Form Builder uses it. However if the window manager does not support the property, Form Builder ignores it.

Functional Property for GUI Hints	**Function**
Close Allowed	Enables the mechanism for closing the window, as provided by the window manager-specific Close command (Form Builder responds to user attempts to close the window by firing a WHEN-WINDOW-CLOSED trigger to actually close it.)
Move Allowed	Determines whether the user can move the window by using the means provided by the window manager
Resize Allowed	Determines whether the user can resize the window at run time
Maximize Allowed	Determines whether the user can resize the window by using the zooming capabilities of the window manager
Minimize Allowed	Determines whether the user can iconify and minimize the window
Inherit Menu	Determines whether the window displays the current form menu

Note: The Minimize Allowed property must be set to Yes in order for Icon Filename to be valid.

How to Create a New Window

1. Click the Windows node in the Object Navigator.
2. Click the Create icon.

   A new window entry displays, with a default name of WINDOW*XX*.
3. If the Property Palette is not already displayed, double-click the window icon to the left of the new window entry.
4. Set the window properties according to your requirements (as described in the tables, earlier in this lesson).

Note: For your new window to display, you must specify its name in the Window property of at least one canvas.

Creating a Content Canvas

- **Implicitly:**
  - **Using the Layout Wizard**
  - **Using the Layout Editor**
- **Explicitly: Using the Create icon in the Object Navigator**

Copyright © Oracle Corporation, 1999. All rights reserved.

Displaying a Form Module on Multiple Layouts

You can have more than one content canvas in your form application. However, remember that only one content canvas can display in a window at one time. To display more than one content canvas at the same time, you can assign each content canvas to a different window.

Now you can display the form module on multiple layouts or surfaces.

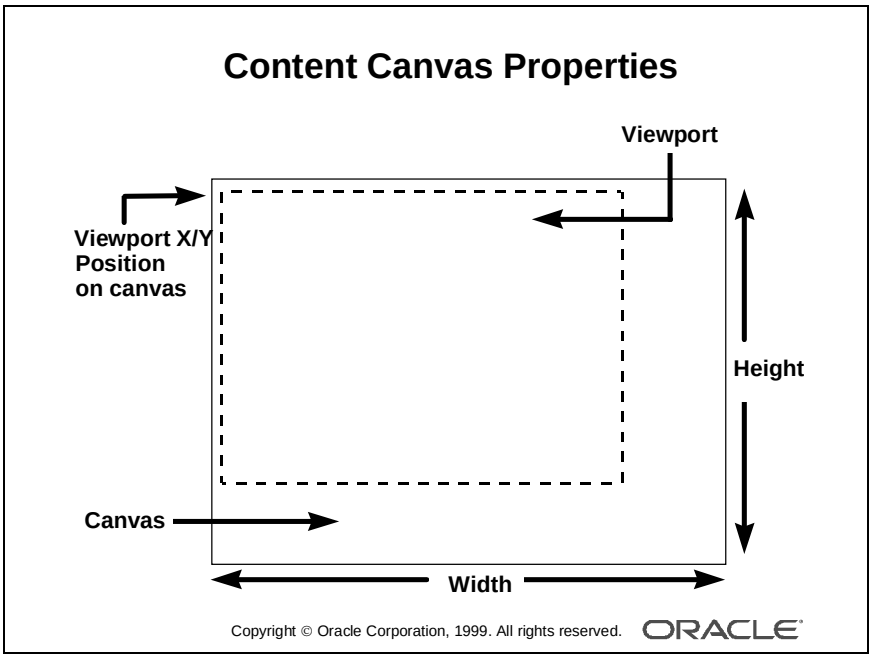
Creating a New Content Canvas Implicitly

There are two ways of implicitly creating a new content canvas:

- Layout Wizard: When you use the Layout Wizard to arrange data block items on a canvas, the wizard enables you to select a new canvas on its Canvas page. In this case, the wizard creates a new canvas with a default name of CANVAS*XX*.
- Layout Editor: When there are no canvases in a form module and you invoke the Layout Editor; Form Builder creates a default canvas on which you can place items.

Creating a New Content Canvas Explicitly

You can create a new content canvas explicitly by using the Create icon in the Object Navigator.


Content Canvas Properties
Viewport
Viewport X/Y
Position
on canvas
Height
Canvas
Width
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Content Canvas Specific Properties

General Property	Function
Canvas Type	Specifies the type of canvas (For a content canvas, this property should be set to Content.)

Physical Property	Function
Window	Specifies the window in which the canvas will be displayed
Viewport X Position on Canvas	Specifies the X coordinate of the top-left corner of the view relative to the upper-left corner of the canvas
Viewport Y Position on Canvas	Specifies the Y coordinate of the top-left corner of the view relative to the upper-left corner of the canvas
Width	Specifies the width of the canvas
Height	Specifies the height of the canvas
Bevel	Specifies a sculpted effect canvas border

Functional Property	Function
Raise on Entry	Determines whether the canvas is always brought to the front of the window when the user navigates to an item on this canvas (Use this property when the canvas is displayed in the same window with other types of canvases.)

Note: For a canvas to display at run time, its Window property must be specified.

How to Create a New Content Canvas

1. Click the Canvases node in the Object Navigator.
2. Click the Create icon.

   A new canvas entry displays with a default name of CANVAS*XX*.
3. If the Property Palette is not already displayed, click the new canvas entry and select Tools—>Property Palette.
4. Set the canvas properties that are described in the above tables according to your requirements.

Note: Double-clicking the icon for a canvas in the Object Navigator will invoke the Layout Editor instead of the Property Palette.

Summary

- **Describing windows and content canvases**
- **Creating a new window**
- **Creating a new content canvas**

Copyright © Oracle Corporation, 1999. All rights reserved.

Summary

In this lesson, you should have learned:

- About the relationship between windows and content canvases
- How to create a new window
- How to create a new content canvas

Practice 10 Overview

This practice covers the following topics:

- **Changing a window size, position, name, and title**
- **Creating a new window**
- **Displaying data block contents in the new window**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 10 in Appendix A, “Practice Solutions.”

Practice 10 Overview

In this practice session, you will customize windows in your form modules. You will resize the windows to make them more suitable for presenting canvas contents. You will also create a new window to display the contents of the S_INVENTORY block.

- Change the size and position of the window in the CUSTOMERS form. Change its name and title. Save and run the form.
- Modify the name and title of the window in the ORDERS form.
- Create a new window in the ORDERS form. Make sure the contents of the S_INVENTORY block display in this window. Save and run the form.

Practice 10

1. Modify the window in the CUSTG*XX* form. Change the name of the window to WIN_CUSTOMER, and change its title to Customer Information. Check that the size and position are suitable.
2. Save, compile, and run the form to test the changes.
3. Modify the window in the ORDG*XX* form. Ensure that the window is called WIN_ORDER. Also change its title to Orders and Items.
4. In the ORDG*XX* form, create a new window called WIN_INVENTORY suitable for displaying the CV_INVENTORY canvas. Use the rulers in the Layout Editor to help you plan the height and width of the window. Set the window title to Stock Levels and the Hide on Exit property to Yes. Place the new window in a suitable position relative to WIN_ORDER.
5. Associate the CV_INVENTORY canvas with the window WIN_INVENTORY. Run the form to ensure that the S_INVENTORY block displays in WIN_INVENTORY when you navigate to this block.
6. Save the form.

11

Working with Other Canvases

Objectives

After completing this lesson, you should be able to do the following:

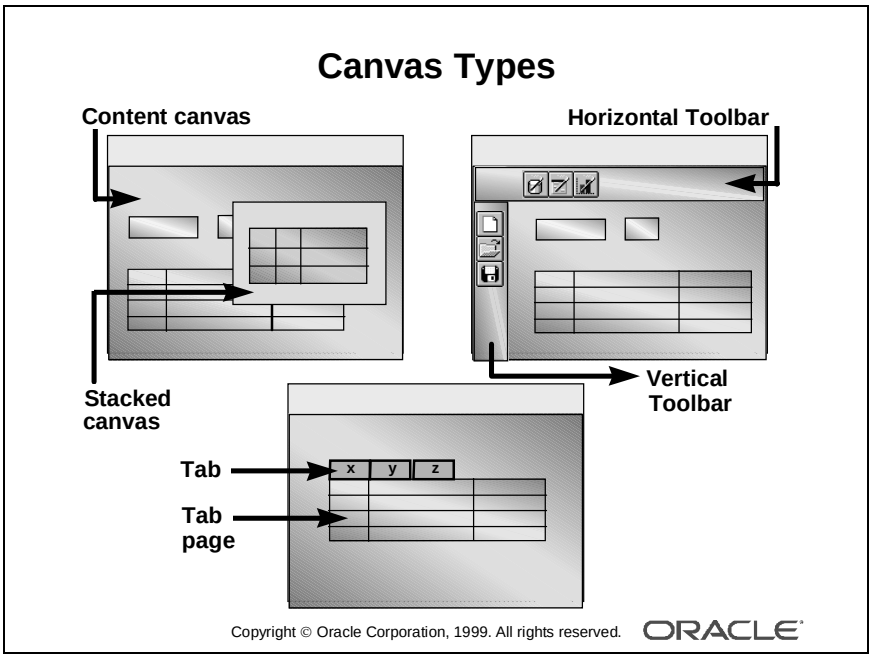
- **Describe the different types of canvases and their relationships to each other**
- **Identify the appropriate canvas type for different scenarios**
- **Create an overlay effect by using stacked canvases**
- **Create a toolbar**
- **Create a tabbed interface**

Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

In addition to the content canvas, the Oracle Developer forms component enables you to create three other canvas types. This lesson introduces you to the stacked canvas, which is ideal for creating overlays in your application. It also introduces you to the toolbar canvas and the tabbed canvas, both of which enable you to provide a user-friendly GUI application.


Canvas Types
Content canvas
Horizontal Toolbar
Stacked canvas
Vertical Toolbar
Tab
Tab page
x
y
z
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Canvases Overview

Besides the content canvas, Form Builder provides three other types of canvases which are:

- Stacked canvas
- Toolbar canvas
- Tab canvas

When you create a canvas, you specify its type by setting the Canvas Type property. The type determines how the canvas is displayed in the window to which it is assigned.

Stacked Canvas

- **Displayed on top of a content canvas**
- **Shares a window with a content canvas**
- **Size:**
  - **Usually smaller than the content canvas in the same window**
  - **Determined by viewport size**
- **Created in:**
  - **Object Navigator**
  - **Layout Editor**

Copyright © Oracle Corporation, 1999. All rights reserved.

Creating a Stacked Canvas

What Is a Stacked Canvas?

A *stacked canvas* is displayed on top of, or stacked on the content canvas assigned to a window. It shares a window with a content canvas and any number of other stacked canvases. Stacked canvases are usually smaller than the window in which they display.

Determining the Size of a Stacked Canvas

Stacked canvases are typically smaller than the content canvas in the same window. Determine the stacked canvas dimensions by setting Width and Height properties. Determine the stacked canvas view dimensions by setting Viewport Width and Viewport Height properties.

Uses and Benefits of Stacked Canvases

- Scrolling views as generated by Oracle Designer
- Creating an overlay effect within a single window
- Displaying headers that display constant information, such as company name
- Creating a cascading or a revealing effect within a single window
- Displaying additional information
- Displaying information conditionally
- Displaying context-sensitive help
- Hiding information

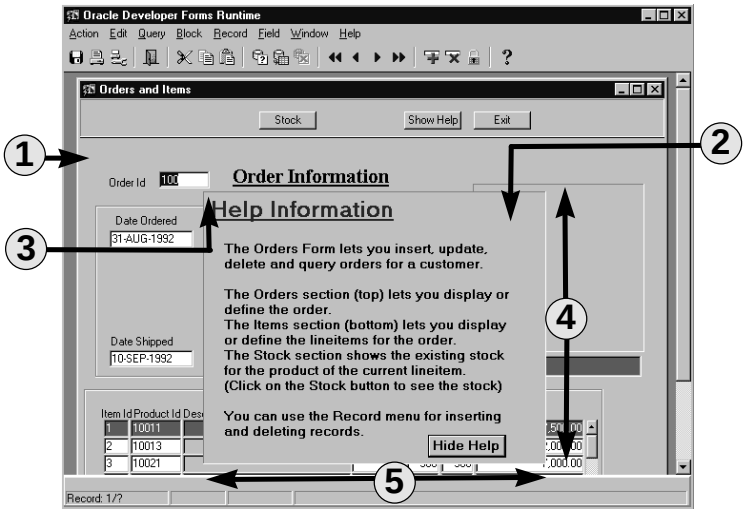
Note: If a data block contains more items than the window can display, Form Builder scrolls the window to display items outside the window frame. This can cause important items, such as primary key values, to scroll out of view. By placing important items on a content canvas, and placing the items that can be scrolled out of sight on a stacked canvas, the stacked canvas becomes the scrolling region, rather than the window itself.

Creating a Stacked Canvas

You can create a stacked canvas in either of the following:

- Object Navigator
- Layout Editor

Stacked Canvas Properties



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

1	Content canvas
2	Stacked canvas
3	Viewport X/Y position
4	Viewport height
5	Viewport width

Stacked Canvas Specific Properties

Viewport Property	Function
Viewport X Position	Specifies the X coordinate of the stacked canvas viewport
Viewport Y Position	Specifies the Y coordinate of the stacked canvas viewport
Viewport Width	Specifies the width of the stacked canvas viewport
Viewport Height	Specifies the height of the stacked canvas viewport

Physical Property	Function
Show Horizontal Scrollbar	Determines whether the stacked canvas displays a horizontal scroll bar
Show Vertical Scroll-bar	Determines whether the stacked canvas displays a vertical scroll bar

How to Create a Stacked Canvas in the Object Navigator

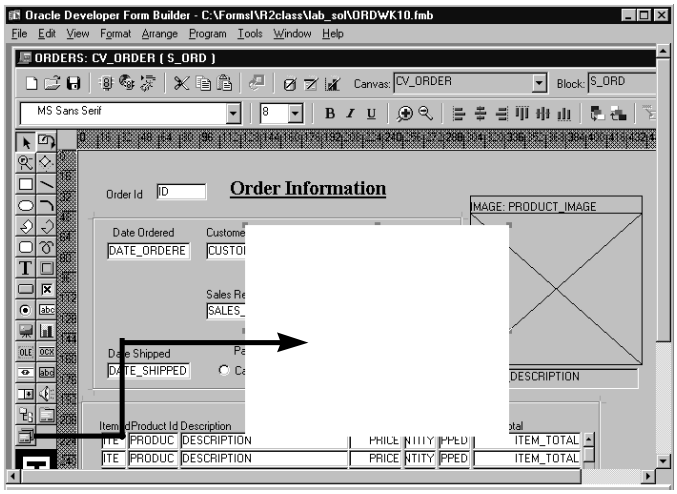
1. Click the Canvases node in the Object Navigator.
2. Click the Create icon.

   A new canvas entry displays with a default name of CANVASXX.
3. If the Property Palette is not already displayed, click the new canvas entry and select Tools—>Property Palette.
4. Set the Canvas Type property to Stacked. Additionally, set the properties that are described in the above table according to your requirements.

Note: To convert an existing content canvas to a stacked canvas, change its Canvas Type property value from Content to Stacked.

In order for the stacked canvas to display properly, make sure that its position in the stacking order places it in front of the content canvas assigned to the same window. The stacking order of canvases is defined by the sequence in which they appear in the Object Navigator.

Creating a Stacked Canvas



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

How to Create a Stacked Canvas in the Layout Editor

1. In the Object Navigator, double-click the object icon for the content canvas on which you wish to create a stacked canvas.

   The Layout Editor displays.
2. Click the Stacked Canvas tool in the toolbar.
3. Click and drag the mouse in the canvas where you want to position the stacked canvas.
4. Open the Property Palette of the stacked canvas. Set the canvas properties according to your requirements (described earlier in the lesson).

Displaying Stacked Canvases in the Layout Editor

You can display a stacked canvas as it sits over the content canvas in the Layout Editor. Check the display position of stacked canvases by doing the following:

1. Select View—>Stacked Views in the Layout Editor. The Stacked/Tab Canvases dialog box is displayed, with a list of all the stacked canvases assigned to the same window as the current content canvas.
2. Select the stacked canvases you want to display in the Layout Editor.

Note: [Control] + Click to clear a stacked canvas that was previously selected.

Toolbars

- **Special type of canvas for tool items**
- **Three types:**
  - **Vertical toolbar**
  - **Horizontal toolbar**
  - **MDI toolbar**
- **Provide:**
  - **Standard look and feel**
  - **Alternative to menu or function key operation**

Copyright © Oracle Corporation, 1999. All rights reserved.

Creating a Toolbar

What Is a Toolbar Canvas?

A *toolbar canvas* is a special type of canvas that you can create to hold buttons and other frequently used GUI elements.

The Three Toolbar Types

- Vertical toolbar: Use a vertical toolbar to position all your tool items down the left or right hand side of your window.
- Horizontal toolbar: Use a horizontal toolbar to position all your tool items and controls across the top or bottom of your window.
- MDI toolbar: Use an MDI toolbar to avoid creating more than one toolbar for a Form Builder application that uses multiple windows.

Uses and Benefits of Toolbars

- Provide a standard look and feel across canvases displayed in the same window.
- Decrease form module maintenance time.
- Increase application usability.
- Create applications similar to others used in the same environment.
- Provide an alternative to menu or function-key driven applications.

Note: The MDI toolbar is only available for Microsoft Windows.

Toolbar Related Properties

- **Canvas properties:**
  - **Canvas Type**
  - **Window**
  - **Width**
  - **Height**

Copyright © Oracle Corporation, 1999. All rights reserved.

Toolbar Related Properties

- **Window properties:**
  - **Horizontal Toolbar Canvas**
  - **Vertical Toolbar Canvas**
- **Form Module properties:**
  - **Form Horizontal Toolbar Canvas**
  - **Form Vertical Toolbar Canvas**

Copyright © Oracle Corporation, 1999. All rights reserved.

Toolbar Related Properties

Once you create a toolbar canvas, you must set its required properties as well as the required properties of the associated window. For MDI toolbars, you must set the required form module properties.

Canvas Property	Function
Canvas Type	Specifies the type of canvas; for a toolbar canvas, set to Horizontal Toolbar or Vertical Toolbar
Window	Specifies which window the toolbar displays in
Width	Determines the width of the toolbar
Height	Determines the height of the toolbar

Window Property	Function
Horizontal Toolbar Canvas/ Vertical Toolbar Canvas	Identifies the horizontal/vertical toolbar to display in this window

Form Module Property	Function
Form Horizontal Toolbar Canvas/ Form Vertical Toolbar Canvas	Identifies the horizontal/vertical toolbar to display in the MDI window

How to Create a Toolbar Canvas

1. Create a new canvas in the Object Navigator.
2. If the Property Palette is not already displayed, click the new canvas entry and select Tools—>Property Palette.
3. Set the canvas properties that are described in the above table.
4. In the Object Navigator select one of the following:
   - The window in which you want to display the toolbar (for a form window toolbar)
   - The Form module (for an MDI Toolbar)
5. Set the Horizontal/Vertical Toolbar Canvas properties.
6. Add GUI elements, boilerplate text, and graphics, as required.

Note: The width of a horizontal toolbar is set to the width of the window (for example, content canvas). Likewise, the height of a vertical toolbar is set to the height of the window.

Tab Canvas

- **Enables you to organize and display related information on separate tabs**
- **Consists of one or more tab pages**
- **Provides easy access to data**
- **Created in:**
  - **Object Navigator**
  - **Layout Editor**

Copyright © Oracle Corporation, 1999. All rights reserved.

Creating a Tab Canvas

What Is a Tab Canvas?

A *tab canvas* is a a special type of canvas that enables you to organize and display related information on separate tabs. Like stacked canvases, tab canvases are displayed on top of a content canvas.

What Is a Tab Page?

A *tab page* is a subobject of a tab canvas. Each tab canvas is made up of one or more tab pages. A tab page displays a subset of the information in the entire tab canvas. Each tab page has a labeled tab that end users can click to access information on the page.

Each tab page occupies an equal amount of space on the tab canvas.

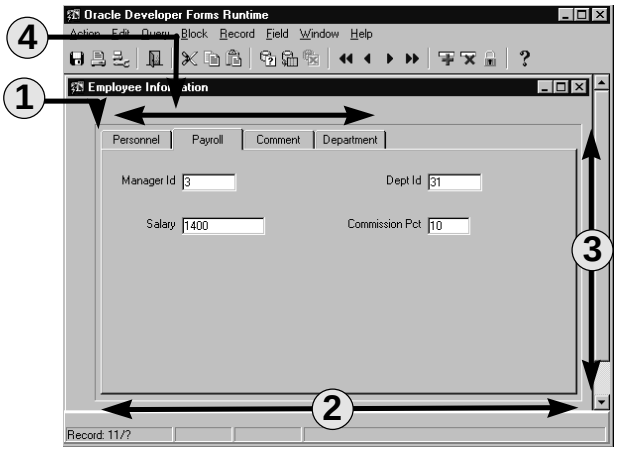
Uses and Benefits of Tab Canvases

- Create an overlay effect within a single window.
- Display large amounts of information on a single canvas.
- Hide information.
- Easily access required information by clicking the tab.

Creating a Tab Canvas

- Create an empty tab canvas in either of the following:
  - Object Navigator
  - Layout Editor
- Define one or more tab pages for the tab canvas.
- Place items on the tab pages.

Tab Canvas Related Properties



Copyright © Oracle Corporation, 1999. All rights reserved.

1	Viewport X/Y position
2	Viewport width
3	Viewport height
4	Tab attachment edge

Tab Canvas Related Properties

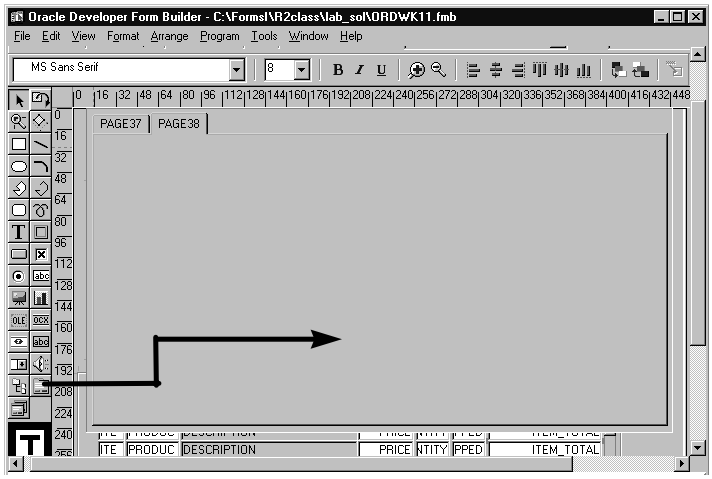
Once you create a tab canvas and its tab pages, you must set the required properties for both of these objects. Place items on a tab page by setting the required item properties.

Tab Canvas Property	Function
Viewport X Position	Specifies the X coordinate of the tab canvas upper-left corner
Viewport Y Position	Specifies the Y coordinate of the tab canvas upper-left corner
Viewport Width	Specifies the width of the view for the tab canvas
Viewport Height	Specifies the height of the view for the tab canvas
Corner Style	Specifies the shape of the labelled tabs on the tab canvas (Select from Chamfered, Square, and Rounded)
Tab Attachment Edge	Specifies the location where tabs are attached to the tab canvas

Tab Page Property	Function
Label	Specifies the text label that appears on the tab page's tab at run time

Item Property	Function
Canvas	Specifies the tab canvas on which the item will be displayed
Tab Page	Specifies the tab page on which the item will be displayed

Creating a Tab Canvas



Copyright © Oracle Corporation, 1999. All rights reserved.

How to Create a Tab Canvas in the Object Navigator

1. Click the Canvases node in the Object Navigator.
2. Click the Create icon.

   A new canvas entry displays.
3. If the Property Palette is not already displayed, click the new canvas entry and select Tools—>Property Palette.
4. Set the Canvas Type property to Tab. Additionally, set the canvas properties according to your requirements (described earlier in the lesson).
5. Expand the canvas node in the Object Navigator.

   The Tab Pages node displays.
6. Click the Create icon.

   A tab page displays in the Object Navigator, with a default name of PAGE*XX*. The Property Palette takes on its context.
7. Set the tab page properties according to your requirements (described earlier in the lesson).
8. Create additional tab pages by repeating steps 6 and 7.

How to Create a Tab Canvas in the Layout Editor

1. In the Object Navigator, double-click the object icon for the content canvas on which you want to create a tab canvas.

   The Layout Editor displays.
2. Click the Tab Canvas tool in the toolbar.
3. Click and drag the mouse in the canvas where you want to position the tab canvas.

   Form Builder creates a tab canvas with two tab pages by default.
4. Open the Property Palette of the tab canvas. Set the canvas properties according to your requirements (described earlier in the lesson).
5. Create additional tab pages, if required, in the Object Navigator.
6. Set the tab page properties according to your requirements (described earlier in the lesson).

Placing Items on a Tab Canvas

- **Place items on each tab page for user interaction.**
- **Set the item properties:**
  - **Canvas**
  - **Tab Page**

Copyright © Oracle Corporation, 1999. All rights reserved.

Placing Items on a Tab Page

Once you create a tab canvas and related tab pages, you must place individual items on the tab pages that the end users can interact with at run time. To accomplish this, do the following:

- Open the Property Palette of the item.
- Set the item's Canvas and Tab Page properties of the item to the desired tab canvas and tab page.

Note: Display the tab canvas as it sits on top of the content canvas, by selecting View—>Stacked View in the Layout Editor.

Summary

- **Creating an overlay effect with a stacked canvas**
- **Creating a toolbar**
- **Creating a tab canvas**

Copyright © Oracle Corporation, 1999. All rights reserved.

Summary

In this lesson you should have learned how to:

- Create an overlay effect with a stacked canvas
- Create a toolbar
- Create a tabbed canvas

Practice 11 Overview

This practice covers the following topics:

- **Creating a toolbar canvas**
- **Creating a stacked canvas**
- **Creating a tab canvas**
- **Adding tab pages to the tab canvas**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 11 in Appendix A, “Practice Solutions.”

Practice 11 Overview

In this practice session, you will create different types of canvases: stacked canvas, toolbar canvas, and tab canvas.

- Create a horizontal toolbar canvas in the ORDERS form. Create new buttons in the Control block, and place them on the horizontal toolbar. Save and run the form.
- Create a stacked canvas in the ORDERS form to add some help text. Position the canvas in the center of the window. Create a button in the Control block. This button will be used later to display the stacked canvas. Add help text on the stacked canvas. Save and run the form.
- Create a tab canvas in the CUSTOMERS form. Create three tab pages on this canvas, and make sure that each tab page displays the appropriate information. Save and run the form.

Practice 11

Toolbar Canvases

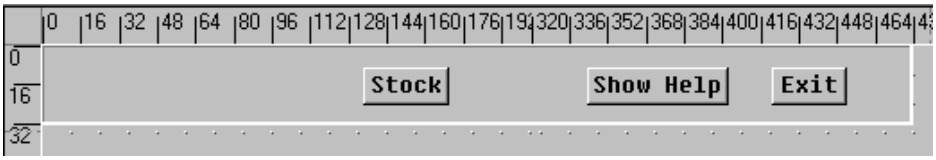
1. In the ORDG*XX* form, create a horizontal toolbar canvas called Toolbar in the WIN_ORDER window, make it the standard toolbar for that window. Suggested height is 30.
2. Save, compile, and run the form to test.

   Notice that the toolbar now uses part of the window's space. Adjust the window size accordingly.

Create three buttons in the CONTROL block, as detailed below, and place them on the Toolbar canvas.

Button Name	**Details**
Stock_Button	Label: Stock Mouse Navigate: No Keyboard Navigable: No Canvas: Toolbar
Show_Help_Button	Label: Show Help Mouse Navigate: No Keyboard Navigable: No Canvas: Toolbar
Exit_Button	Label: Exit Mouse Navigate: No Keyboard Navigable: No Canvas: Toolbar

Suggested positions for the buttons are shown in the following illustration:



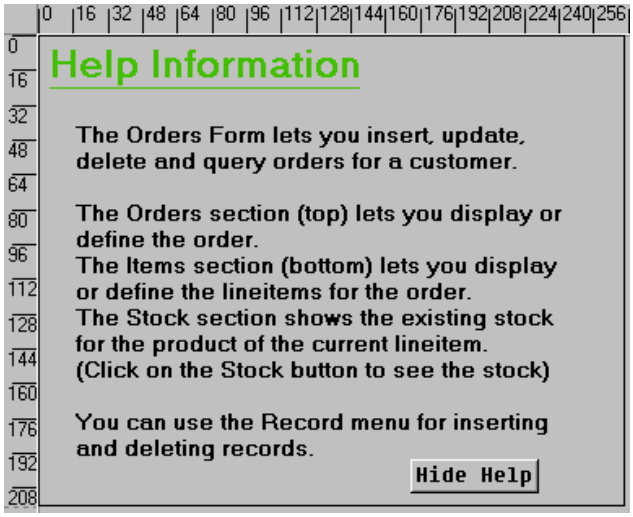
Stacked Canvases

1. Create a stacked canvas named CV_HELP to display help in the WIN_ORDER window of the ORDG*XX* form. Suggested *visible* size is Viewport Width 270, Viewport Height 215 (points). Place some application help text on this canvas.
2. Position the view of the stacked canvas so that it appears in the center of WIN_ORDER. Make sure it will not obscure the first enterable item.

   Do this by planning the view's top-left position in the Layout Editor, while showing CV_ORDER. Define the Viewport X and Viewport Y Positions in the Property Palette. Do not move the view in the Layout Editor.
3. Organize CV_HELP so that it is the last canvas in sequence.

   Do this in the Object Navigator. (This ensures the correct stacking order at run time.)
4. Save, compile, and run the form to test. Note that the stacked canvas displays all the time, providing that it does not obscure the current item in the form.
5. Switch off the Visible property of CV_HELP, then create a button in the control block to hide the Help information when it is no longer needed. We will add the code later. Display this button on the CV_HELP canvas.

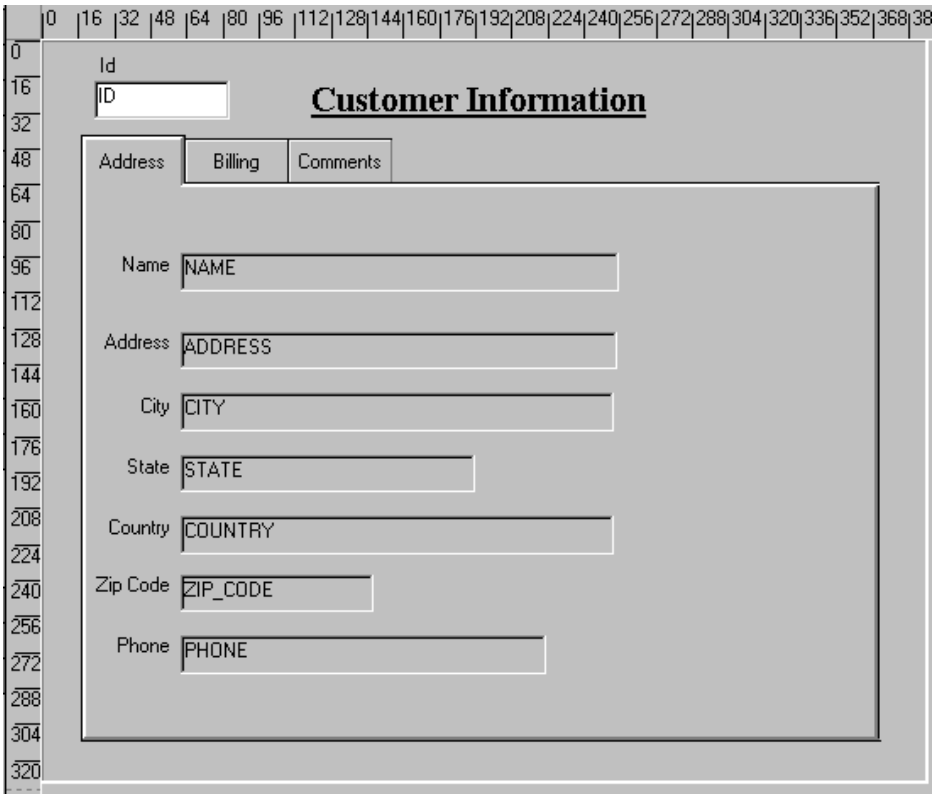
Button Name	**Details**
Hide_Help_Button	Label: Hide Help, Canvas: CV_HELP Mouse Navigate: No



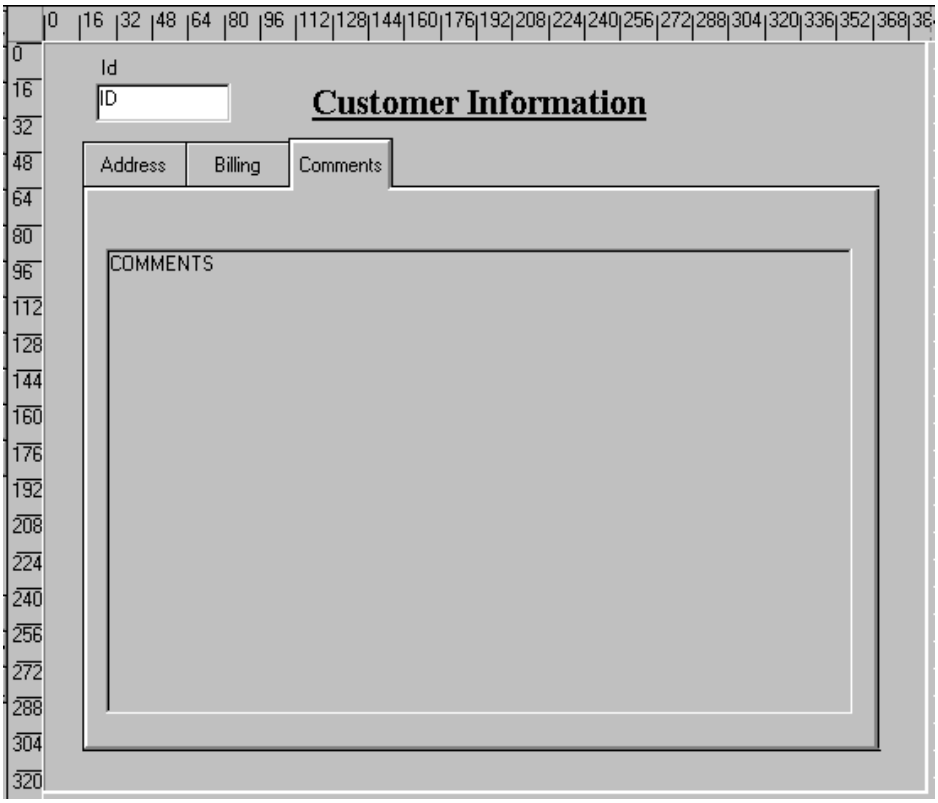
Tab Canvases

Modify the CUSTG*XX* form in order to use a tab canvas:

1. In the Layout Editor, delete the frame object that covers S_CUSTOMER block. Create a tab canvas. In the Layout Editor set the Background Color property to gray, Tab style property to Square, and Bevel property to None.
2. Rename this tab canvas TAB_CUSTOMER. Create three tab pages and label them as `Address`, `Billing`, and `Comments`.
3. Design the tab pages according to the following screenshots. Set the item properties to make them visible on the relevant tab pages.


Id
ID
Customer Information
Address
Billing
Comments
Name
NAME
Address
ADDRESS
City
CITY
State
STATE
Country
COUNTRY
Zip Code
ZIP_CODE
Phone
PHONE


Id
ID
Customer Information
Address
Billing
Comments
Credit Rating
CREDIT_RATING
Sales Rep Id
SALES_REP_ID


Id
ID
Customer Information
Address
Billing
Comments
COMMENTS

Tab Canvases (continued)

4. Reorder the items according to the tab page sequence. Ensure that the user does not move from one tab page to another when tabbing through items. Set Next Navigation Item and Previous Navigation Item properties according to the order of items in the tab pages.
5. Save, compile, and run the form.

12

Introduction to Triggers

Objectives

After completing this lesson, you should be able to do the following:

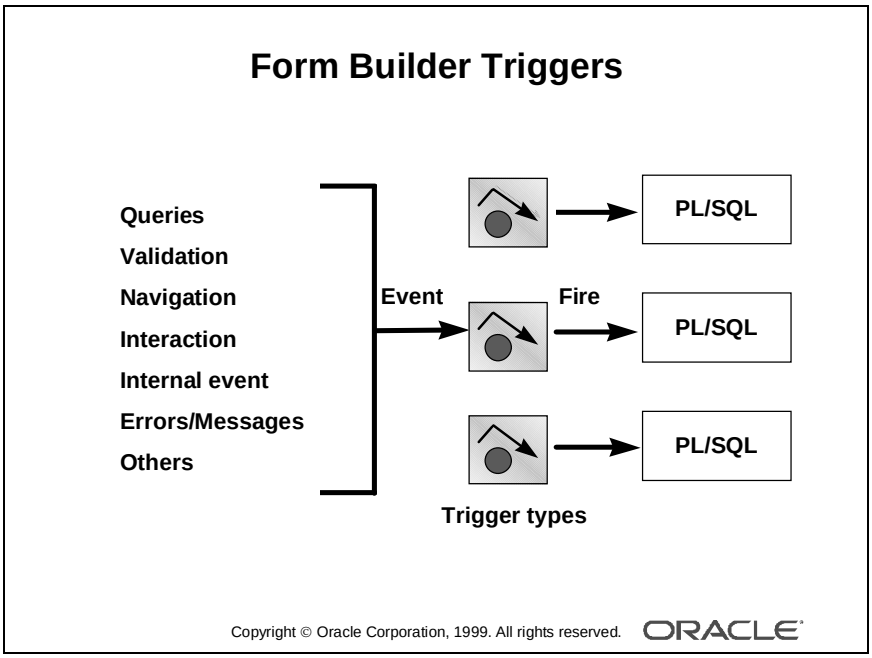
- **Define triggers**
- **Identify the different trigger categories**
- **Plan the type and scope of triggers in a form**
- **Describe the properties that affect the behavior of a trigger**

Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

Triggers are one of the most important mechanisms that you can use to modify or add to the functionality of a form. In this lesson, you learn the essential rules and properties of triggers so that you can use them throughout your application.



Note

Events cause the activation, or firing, of certain trigger types.

What Is a Trigger?

A *trigger* is a program unit that is executed (fired) due to an event. You have already seen that Form Builder enables you to build powerful facilities into applications without writing a single line of code. You can use triggers to add or modify form functionality in a procedural way. As a result, you can define the detailed processes of your application.

Every trigger that you define is associated with a specific event. Form Builder defines a vast range of events for which you can fire a trigger. These events include the following:

- Query-related events
- Data entry and validation
- Logical navigation or physical mouse movement
- Operator interaction with items in the form
- Internal events in the form
- Errors and messages

Trigger Characteristics

As with other Oracle Developer components, you write Form Builder triggers in PL/SQL. These triggers are mostly fired by events within a form module. (Menu modules can initiate an event in a form, but the form module owns the trigger that fires.)

Note: Database events that occur on behalf of a form can fire certain Form Builder triggers, but these database triggers are different from Form Builder triggers.

Trigger Components

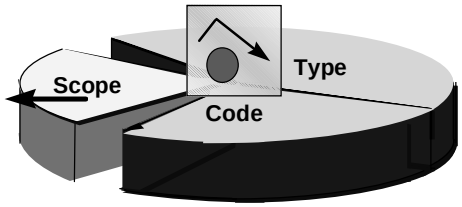
There are three main components to consider when you design a trigger in Form Builder:

Component	Description
Trigger type	Defines the specific event that will cause the trigger to fire
Trigger code	The body of PL/SQL that defines the actions of the trigger
Trigger scope	The level in a form module at which the trigger is defined—determining the scope of events that will be detected by the trigger

Trigger Scope Component

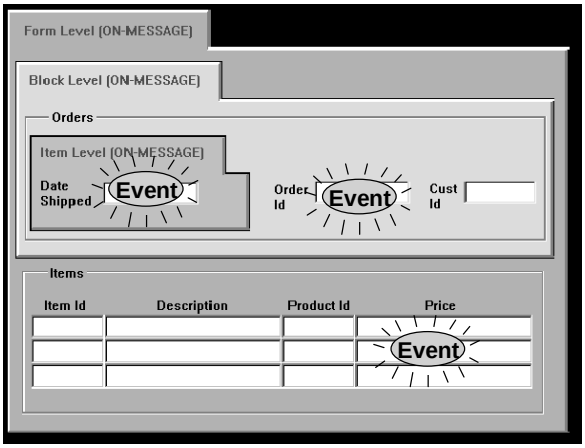
Levels

- **Form**
- **Block**
- **Item**



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Trigger Scope



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Trigger Components

Trigger Scope

The scope of a trigger is determined by its position in the form object hierarchy, that is, the type of object under which you create the trigger.

There are three possible levels:

Scope	Description
Form level	The trigger belongs to the form and can fire due to events across the entire form
Block level	The trigger belongs to a block and can only fire when this block is the current block
Item level	The trigger belongs to an individual item and can only fire when this item is the current item

Some triggers cannot be defined below a certain level. For example, Post-Query triggers cannot be defined at item level, because they fire due to a global or restricted query on a block.

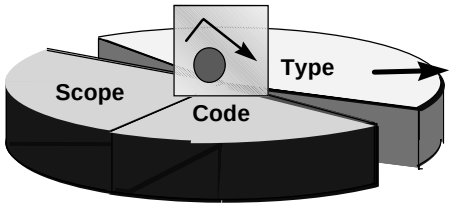
By default, only the trigger that is most specific to the current location of the cursor fires.

Consider the example in the diagram, opposite:

- When the cursor is in the Date_Shipped item, a message fires the On-Message trigger of the Date_Shipped item, because this is more specific than the other triggers of this type.
- When the cursor is elsewhere in the ORDERS block, a message causes the block-level On-Message trigger to fire, because its scope is more specific than the form-level trigger. (You are outside the scope of the item-level trigger.)
- When the cursor is in the ITEMS block, a message causes the form-level On-Message trigger to fire, because the cursor is outside the scope of the other two On-Message triggers.

Note: The On-Message trigger fires whenever Form Builder displays a message.

Trigger Type Component



- Pre-
- Post-
- When-
- On-
- Key-

Copyright © Oracle Corporation, 1999. All rights reserved.

Trigger Type

The trigger type determines which type of event fires it. There are more than 100 built-in triggers, each identified by a specific name.

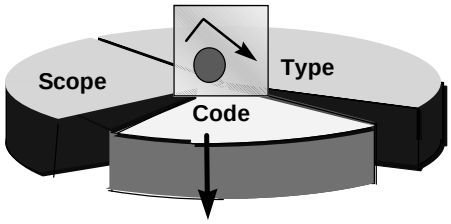
The name of a trigger identifies its type. All built-in trigger types are associated with an event, and their names always contain a hyphen (-). For example:

- When-Validate-Item fires when Form Builder validates an item.
- Pre-Query fires before Form Builder issues a query for a block.

The first part of a trigger name (before the first hyphen) follows a standard convention; this helps you to understand the general nature of the trigger type, and plan the types to use.

Trigger Prefix	**Description**
Key-	Fires in place of the standard action of a function key
On-	Fires in place of standard processing (used to replace or bypass a process)
Pre-	Fires on an event that occurs just before an action (for example, before a query is executed)
Post-	Fires just after an action (for example, after a query is executed)
When-	Fires in addition to standard processing (used to augment functionality)

Trigger Code Component



- **Statements**
- **PL/SQL**
- **User subprograms**
- **Built-in subprograms**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Trigger Code

The code of the trigger defines the actions for the trigger to perform when it fires. Write this code as an anonymous PL/SQL block by using the PL/SQL Editor.

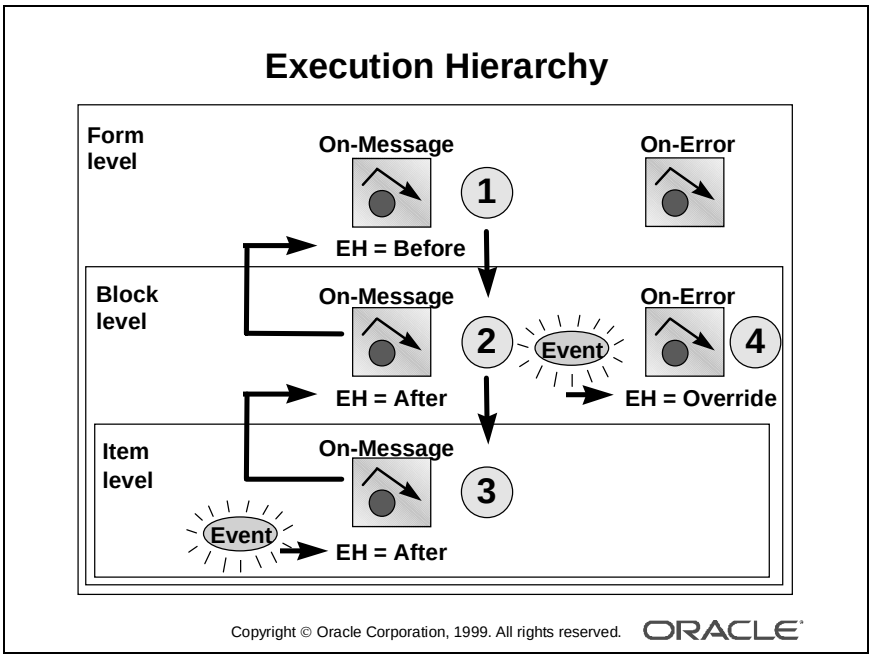
Note: You only need to enter the BEGIN. . . END structure in your trigger text if you start your block with a DECLARE statement or if you need to code subblocks for other reasons.

Statements that you write in a trigger can be constructed as follows:

- Standard PL/SQL constructs (assignments, control statements, and so on).
- SQL statements that are legal in a PL/SQL block; these are passed to the server for execution.
- Calls to user-named subprograms (procedures and functions) in the form, a library, or the database.
- Calls to built-in subprograms and package subprograms; these are procedures and functions that are part of Oracle Developer.

Although you can include SQL statements in a trigger, keep in mind the following rules about their use:

- INSERT, UPDATE, and DELETE statements must be placed only in transactional triggers. These triggers fire during the commit process.
- Transaction control statements (COMMIT, ROLLBACK, SAVEPOINT) cannot be included directly as SQL trigger statements. These actions are carried out by Form Builder as a result of either commands or built-in procedures that you issue.



Note

Broken lines indicate the analysis path before firing. EH stands for execution hierarchy.

1	Fires first
2	Fires second
3	Viewport height
4	Fires independently

Trigger Scope and Execution Hierarchy

As already stated, when there is more than one trigger of the same type Form Builder normally fires the trigger most specific to the cursor location. You can alter the firing sequence of a trigger by setting the execution hierarchy (EH) trigger property.

Execution hierarchy is a trigger property that controls what happens when there are triggers of the same type at different levels, but each trigger is within the scope of an event. The default setting is Override.

Settings for execution hierarchy are:

Setting	**Description**
Override	Only the trigger most specific to the cursor location fires
After	The trigger fires *after* firing the same trigger at the next highest level (if a trigger exists)
Before	The trigger fires *before* firing the same trigger at the next highest level (if one exists)

In the cases of Before and After, you can fire more than one trigger of the same type due to a single event. However, you must define each trigger at a different level.

Summary

- **Trigger: Event-activated program units**
- **Type: Defines the event that fires it**
- **Prefixes:**
  - **Key-**
  - **On-**
  - **Pre-**
  - **Post-**
  - **When-**
- **Code: PL/SQL anonymous block**
- **Scope: Form, block, or item level**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

In this lesson you should have learned the essential rules and properties for triggers.

- Triggers are event-activated program units.
- The trigger type defines the event that fires the trigger.
- Prefixes for trigger names are:
  - Key-
  - On-
  - Pre-
  - Post-
  - When-

  Each has a specific meaning.
- Trigger code consists of a PL/SQL anonymous block.
- The trigger scope determines which events will be detected by the trigger. The three possible levels for a trigger are form, block, and item.
- When an event occurs, the most specific trigger overrides the triggers at a more general level. This can be affected by execution hierarchy.

13

Producing Triggers

Objectives

After completing this lesson, you should be able to do the following:

- **Write trigger code**
- **Explain the use of built-in subprograms in Oracle Developer applications**
- **Describe the When-Button-Pressed trigger**
- **Describe the When-Window-Closed trigger**

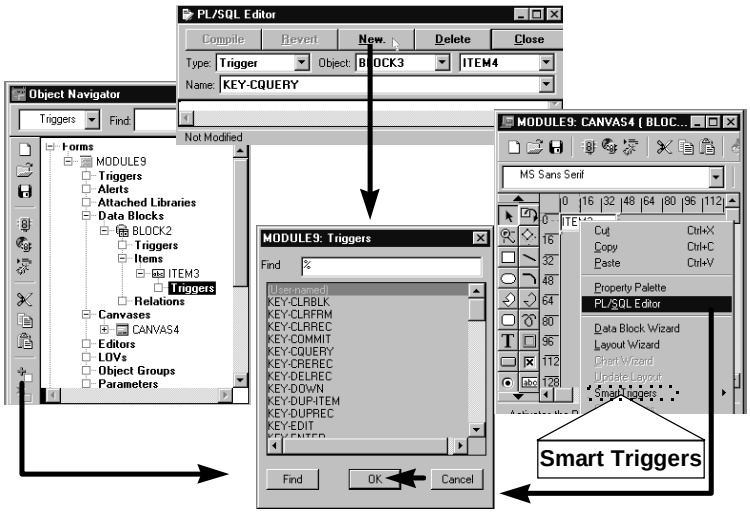
Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

This lesson shows you how to create triggers. You specifically learn how to use built-in subprograms in Oracle Developer applications.

Creating a Trigger



Copyright © Oracle Corporation, 1999. All rights reserved.

Defining Triggers in Form Builder

Using Smart Triggers

When you click an object in the Object Navigator or Layout Editor by using the right mouse button, a pop-up menu displays that includes the item Smart Triggers. The Smart Triggers item expands to a list of common triggers that are appropriate for the selected object. When you click one of these triggers, the Form Builder automatically creates the trigger.

Creating a New Trigger

Using Smart Triggers is the easiest way to create a new trigger, but you can also do it from the Object Navigator, from the Layout Editor, or from the PL/SQL Editor if it is already open:

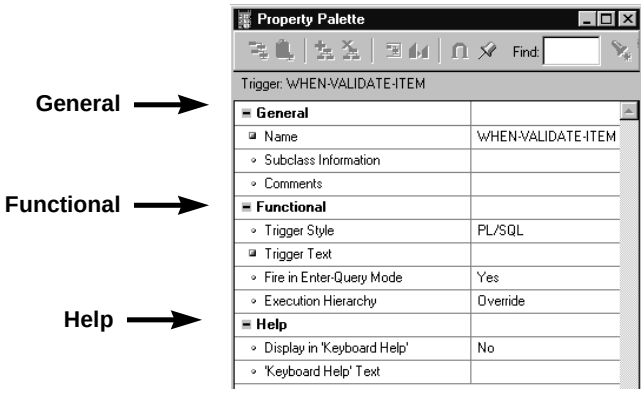
- In the Object Navigator, select the Triggers node of the form, block, or item that will own the trigger. Select Navigator—>Create from the menu, or click Create in the toolbar. This invokes the Trigger LOV.
- If the PL/SQL Editor is open, click New to create a new trigger. This invokes the Trigger LOV.
- In the Layout Editor, select the object, and click the right mouse button to display the pop-up menu. Select PL/SQL Editor, if there is already a trigger attached to the item; its name and code appear in the editor. Click the New button to invoke the Triggers LOV.
- Select the trigger type from the Triggers LOV. The trigger type and scope are now set in the PL/SQL Editor. You can enter the code for the trigger in the source pane of the editor.

Using the PL/SQL Editor

You are already familiar with the PL/SQL Editor, which is common in each Oracle Developer component. In the Form Builder, the PL/SQL Editor has the following specific trigger components:

Component	**Description**
Type	Set to trigger
Object	Enables you to set the scope to either Form Level or a specific block
Item	Enables you to change between specific items (at item level) to access other triggers
Name	Trigger name; enables you to switch to another existing trigger
Source pane	Where trigger code is entered or modified

Trigger Properties



Copyright © Oracle Corporation, 1999. All rights reserved.

Trigger Properties

In the property palette, you can set the following trigger properties:

General

Property	Description
Name	Specifies the internal name of the trigger

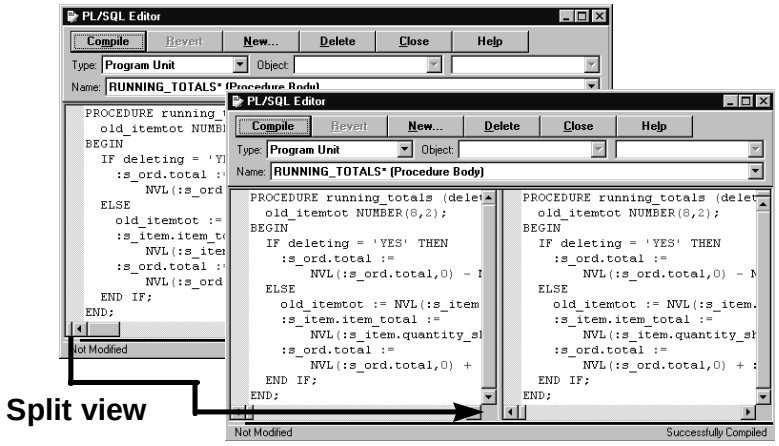
Functional

Property	Description
Trigger Style	PL/SQL: Trigger code is a PL/SQL block (default) V2: Trigger is inherited from version 2.3 or earlier
Fire in Enter Query Mode	Yes: Trigger can fire when an event occurs in Enter Query as well as Normal mode No: Trigger can fire only in Normal mode
Execution Hierarchy	Override, Before, or After

Help

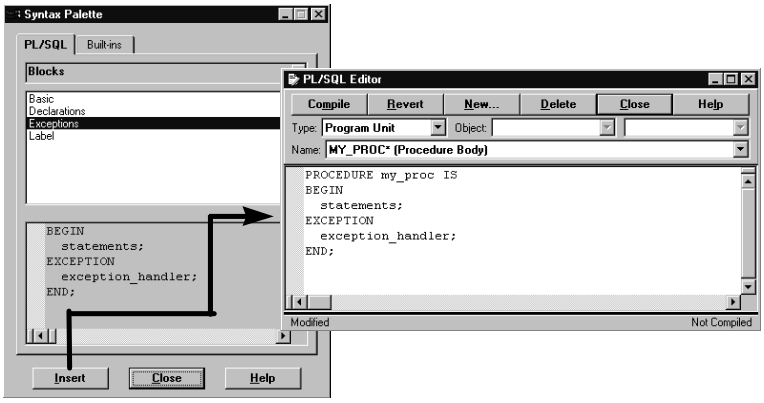
Property	Description
Display in “Keyboard Help”	Set to Yes if you want the name or the description to appear in the Show Keys window; the default is No
“Keyboard Help” text	Set to Yes if you want to specify the trigger description; this property is valid for Key- triggers

Enhanced PL/SQL Editor



Copyright © Oracle Corporation, 1999. All rights reserved.

Enhanced PL/SQL Editor Syntax Palette



Copyright © Oracle Corporation, 1999. All rights reserved.

PL/SQL Editor Features

Automatic Formatting and Coloring of PL/SQL Code

- Automatic Indenting and Color Syntax highlighting
- Drag and Drop text Manipulation
- Unlimited Undo/Redo

Multiple Split Views

You can create up to four separate views of the current program unit in the PL/SQL Editor by using split bars.

Syntax Palette

The Syntax Palette enables you to display and copy the constructs of PL/SQL language elements and build packages into an editor. To invoke the Syntax Palette, select Program—>Syntax Palette from the menu system.

Global Search and Replace

The Find and Replace in Program Units dialog box enables you to search for text across multiple program units without opening individual instances of the Program Unit Editor. Choose to replace every occurrence of the search text string found or in selected occurrences only.

Invoke the Find and Replace in Program Units dialog box by selecting

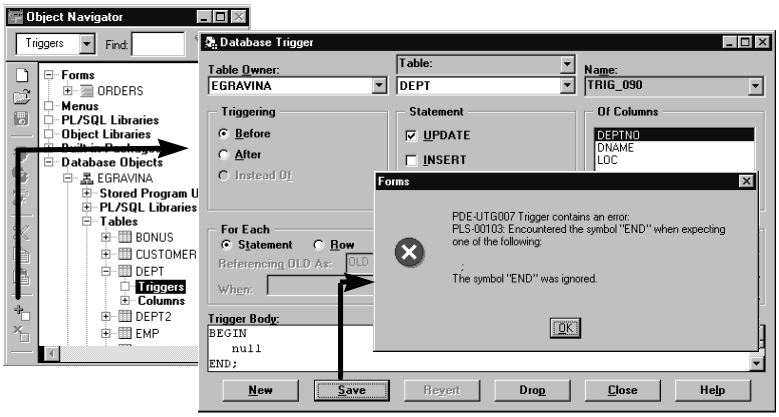
Program—>Find and Replace PL/SQL from the menu system.

Things to Remember About the PL/SQL Editor

- New or changed text in triggers remains uncompiled until you click Compile. (If you select File—>Compile from the menu, it will compile all uncompiled code in the document.)
- Compiling triggers that contain SQL require connection to the database.
- All uncompiled triggers are compiled when the form module is compiled.

The Block and Item pop-up lists do *not* change the current trigger scope. They enable you to switch to another trigger.

Database Trigger Editor



Copyright © Oracle Corporation, 1999. All rights reserved.

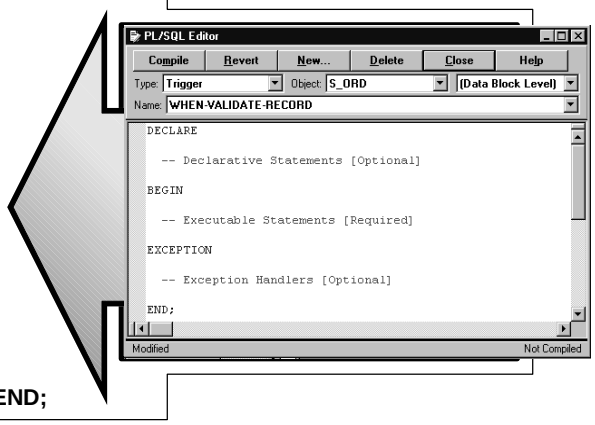
Database Trigger Editor

The logical grouping of items within the Database Trigger Editor enables developers to create row and statement triggers easily. An error message box displays an error when you try to retrieve, store, or drop an invalid trigger. To create a database trigger by using the Database Trigger Editor, perform the following steps:

1. In the Object Navigator, expand the Database Objects node to display the schema nodes.
2. Expand a schema node to display the database objects.
3. Expand the Tables node to display the schema's database tables.
4. Select and expand the desired table.
5. Select the Triggers node and choose Navigator-Create. The Database Trigger Editor appears.
6. In the Database Trigger Editor, define and save the desired program units.

Trigger PL/SQL Blocks

.
BEGIN



.
END;

Copyright © Oracle Corporation, 1999. All rights reserved.

Writing the Trigger Code

Trigger PL/SQL Blocks

The code text of a Form Builder trigger is a PL/SQL block that consists of three sections:

- A declaration section for variables, constants, and exceptions (optional)
- An executable statements section (required)
- An exception handlers section (optional)

If your trigger code does not require defined variables, you do not need to include the BEGIN and END keywords; they are added implicitly.

Example

If the trigger does not require declarative statements, the BEGIN and END keywords are optional. When-Validate_Item trigger:

```
IF :S_ITEM.price IS NULL THEN
   :S_ITEM.price := :S_ITEM.stdprice;
END IF;
calculate_total; -- User-named procedure
```

Example

If the trigger requires declarative statements, the BEGIN and END keywords are required. When-Button-Pressed trigger:

```
DECLARE
   vn_discount NUMBER;
BEGIN
   vn_discount:=calculate_discount(:S_ITEM.product_id,:S_ITEM.quantity);
   MESSAGE('Discount: '||TO_CHAR(vn_discount));
END;
```

Example

To handle exceptions, include EXCEPTION section in trigger. Post-Insert trigger:

```
INSERT INTO LOG_TAB (LOG_VAL, LOG_USER)
VALUES(:S_DEPT.id,:GLOBAL.username);
EXCEPTION
   WHEN OTHERS THEN
      MESSAGE('Error! ',||SQLERRM);
```

Variables in Form Builder

- PL/SQL variables must be declared in a trigger or defined in a package
- Form Builder variables
  - Are not formally declared in PL/SQL
  - Need a colon prefix in reference

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Form Builder Variables

- Items

  For presentation and user interaction
- Global variables

  Session-wide character variable
- System variables

  Form status and control
- Parameters

  Passing values in and out of module

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Using Variables in Form Builder

In triggers and subprograms, Form Builder generally accepts two types of variables for storing values:

- PL/SQL variables: These must be declared in a DECLARE section, and remain available until the end of the declaring block. They are *not* prefixed by a colon. If declared in a PL/SQL package, a variable is accessible across all triggers that access this package.
- Form Builder variables: Variable types maintained by the Form Builder. These are seen by PL/SQL as external variables, and require a colon (:) prefix to distinguish them from PL/SQL objects (except when their name is passed as a character string to a subprogram). Form Builder variables are *not* formally declared in a DECLARE section, and can exist outside the scope of a PL/SQL block.

Form Builder Variables

The following variables are available for the storage and manipulation of values:

Form Builder Variable Type	**Description**
Item (text, list, check box, and so on)	Scope: Current form and attached menu Use: Presentation and interaction with user
Global variable	Scope: All modules in current session Use: Session-wide storage of character data
System variable	Scope: Current form and attached menu Use: Form status and control
Parameter	Scope: Current module Use: Passing values in and out of module

Initializing Global Variables with Default Value

You can use the DEFAULT_VALUE built-in to assign a value to a global variable. Form Builder creates the global variable if it does not exist. If the value of the indicated variable is not null, DEFAULT_VALUE does nothing. The following example creates a global variable named country and initializes it with the value TURKEY:

```
Default_Value('TURKEY','GLOBAL.country');
```

Syntax of Variables

- **:block_name.item_name**
- **:GLOBAL.variable_name**
- **:SYSTEM.variable_name**
- **:PARAMETER.name**

Copyright © Oracle Corporation, 1999. All rights reserved.

Examples of Form Builder Variables

In each of the following examples, note that a colon (:) prefixes Form Builder variables, and a period (.) separates the components of their name. The examples are not complete triggers.

Example

References to items should be prefixed by the name of the owning Form Builder block, which prevents ambiguity when items of the same name exist in different blocks. This is also more efficient than the item name alone:

```
:BLOCK3.product_id := :BLOCK2.product_id;
```

Example

References to global variables must be prefixed by the word *global*. They may be created as the result of an assignment:

```
:GLOBAL.customer_id := :BLOCK1.id;
```

Example

References to system variables must be prefixed by the word *system*:

```
IF :SYSTEM.MODE = 'NORMAL' THEN
   ok_to_leave_block  := TRUE;
END IF;
```

Example

Parameters defined at design-time have the prefix *parameter*:

```
IF :PARAMETER.starting_point = 2 THEN
   GO_BLOCK('BLOCK2');  -- built-in procedure
END IF;
```

Removing Global Variables

You can use the ERASE built-in to remove a global variable. Globals always allocate 255 bytes of storage. To ensure that performance is not impacted more than necessary, always erase any global variable when it is no longer needed.

Form Builder Built-in Subprograms

Built-ins belong to either:

- **The Standard Extensions package where no prefix is required**
- **Another Form Builder package where a prefix is required**

Copyright © Oracle Corporation, 1999. All rights reserved.

Adding Functionality Using Built-in Subprograms

The Form Builder Built-in Subprograms

Form Builder provides a set of predefined subprograms as part of the product. These subprograms are defined within built-in packages as either a procedure or function.

The Form Builder built-in subprograms belong to one of the following:

- Standard Extensions packages: These built-ins are integrated into the Standard PL/SQL command set in Form Builder. You can call them directly, without any package prefix. You can use more than one hundred standard built-ins.
- Other Form Builder packages: Subprograms in other built-in packages provide functionality related to a particular supported feature. These require the package name as a prefix when called.

Package	Description
DDE	Provides Dynamic Data Exchange support
DEBUG	Provides built-ins for debugging PL/SQL program units
EXEC_SQL	Provides built-ins for executing dynamic SQL within PL/SQL procedures
FTREE	Provides built-ins for manipulating hierarchical tree items
OLE2	Provides a PL/SQL API for creating, manipulating, and accessing attributes of OLE2 automation objects
ORA_FFI	Provides built-ins for calling out to foreign (C) functions from PL/SQL
ORA_NLS	Enables you to extract high-level information about your current language environment
ORA_PROF	Provides built-ins for tuning PL/SQL program units
TEXT_IO	Provides built-ins to read and write information from and to files
PECS	Provides built-ins for the Performance Event Collection Services; provided for backward compatibility
TOOL_ENV	Enables you to interact with Oracle environment variables
TOOL_ERR	Enables you to access and manipulate the error stack created by other built-in packages such as Debug
TOOL_RES	Provides built-ins to manipulate resource files
VBX	Provides built-ins for controlling and interacting with VBX controls; this package works only in a 16-bit environment and is provided for backward compatibility
WEB	Provides built-ins for the Web environment

All the built-in subprograms used in this lesson are part of the Standard Extensions package.

Limits of Use

- **Unrestricted built-ins are allowed in any trigger or subprogram.**
- **Restricted built-ins are allowed only in certain triggers and subprograms called from such triggers.**
- **Consult the Help system.**

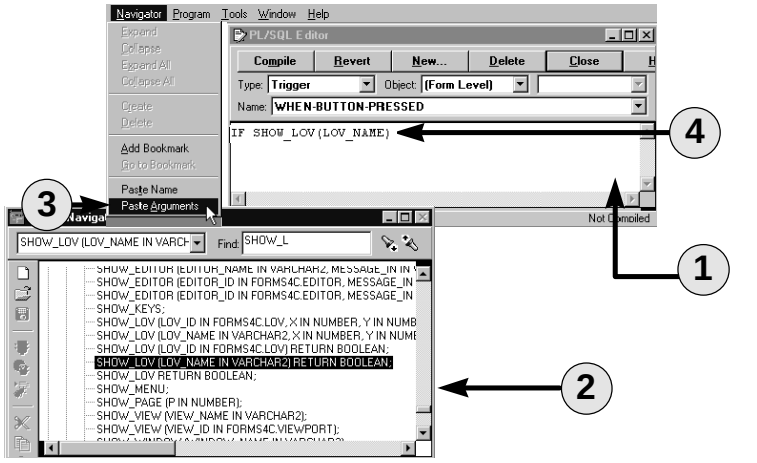
Copyright © Oracle Corporation, 1999. All rights reserved.

Where Can Built-in Subprograms Be Used?

You can call built-ins in any trigger or user-named subprogram in which you use PL/SQL. However, some built-ins provide functionality that is not allowed in certain trigger types. Built-ins are therefore divided into two groups:

- Unrestricted built-ins: Unrestricted built-ins do not affect logical or physical navigation and can be called from any trigger, or from any subprogram.
- Restricted built-ins: Restricted built-ins affect navigation in your form, either external screen navigation, or internal navigation. You can call these built-ins only from triggers while no internal navigation is occurring. The online Help specifies which groups of built-ins can be used in each trigger.

Using Built-in Definitions



Copyright © Oracle Corporation, 1999. All rights reserved.

1	Open PL/SQL code
2	Select built-in
3	Paste names or arguments
4	Modify pasted code

Using Built-in Definitions in the Form Builder

When you are writing a trigger or program unit, the Form Builder enables you to look up built-in definitions, and optionally copy their names and argument prototypes into your code.

1. Place the cursor at the point in your PL/SQL code (in the PL/SQL Editor) where a built-in subprogram is to be called.
2. Expand the Built-in Packages node in the Navigator, and select the procedure or function that you need to use (usually from Standard Extensions).
3. If you want to copy the built-in prototype arguments or name, or both, select Navigator—>Paste Name or Navigator—>Paste Arguments from the menus (Paste Arguments includes the built-in name also).
4. The definition of the built-in is copied to the cursor position in the PL/SQL Editor, where you can insert your own values for arguments, as required.

Note: A subprogram can be either a procedure or a function. Built-in subprograms are therefore called in two distinct ways:

- Built-in procedures: Called as a complete statement in a trigger or program unit with mandatory arguments.
- Built-in functions: Called as *part of* a statement, in a trigger or program unit, at a position where the function's return value will be used. Again, the function call must include any mandatory arguments.

Example

The SHOW_LOV built-in is a function that returns a Boolean value (indicating whether the user has chosen a value from the LOV). It might be called as part of an assignment to a boolean variable. This is *not a complete* trigger.

```
DECLARE
   customer_chosen BOOLEAN;
BEGIN
   customer_chosen := SHOW_LOV('customer_list');
. . .
```

Useful Built-ins

- EDIT_TEXTITEM
- ENTER_QUERY, EXECUTE_QUERY
- EXIT_FORM
- GO_BLOCK, GO_ ITEM
- GET_ITEM_PROPERTY, SET_ITEM_PROPERTY
- MESSAGE
- SHOW_ALERT, SHOW_ EDITOR, SHOW_ LOV
- SHOW_VIEW, HIDE_VIEW

Copyright © Oracle Corporation, 1999. All rights reserved.

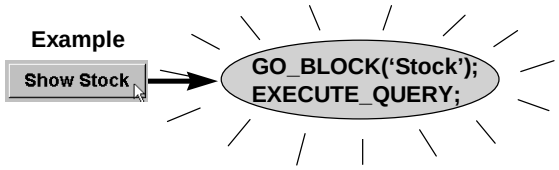
Useful Built-ins for Adding Functionality to Items

Here are just a few built-ins that you can use in triggers to add functionality to items. They are discussed in later lessons.

Built-in Subprogram	Description
EDIT_TEXTITEM procedure	Invokes the Form Runtime item editor for the current text item
ENTER_QUERY procedure	Clears the current block, and creates a sample record (Operators can then specify query conditions before executing the query with a menu or button command. If there are changes to commit, the Form Builder prompts the operator to commit them before continuing ENTER_QUERY processing.)
EXECUTE_QUERY procedure	Clears the current block, opens a query, and fetches a number of selected records (If there are changes to commit, Form Builder prompts the operator to commit them before continuing EXECUTE_QUERY processing.)
EXIT_FORM procedure	Exits current form (or cancels query, if in ENTER-QUERY mode)
GET_ITEM_PROPERTY function	Returns specified property values for the specified item
GO_BLOCK procedure	Navigates to the specified block
GO_ITEM procedure	Navigates to the specified item
HIDE_VIEW procedure	Hides the indicated canvas
LIST_VALUES procedure	Invokes the LOV attached to the current item
MESSAGE procedure	Displays specified text on the message line
SET_ITEM_PROPERTY procedure	Changes setting of specified property for an item
SHOW_ALERT function	Displays the given alert, and returns a numeric value when the operator selects one of three alert buttons
SHOW_EDITOR procedure	Displays the specified editor at the given coordinates and passes a string to the editor, or retrieves an existing string from the editor
SHOW_LOV function	Invokes a specified LOV and returns a Boolean value, indicating whether user selected a value from the list
SHOW_VIEW procedure	Displays the indicated canvas at the coordinates specified by the X Position and Y Position of the canvas property settings (If the view is already displayed, SHOW_VIEW raises it in front of any other views in the same window.)

When-Button-Pressed Trigger

- **Fires when the operator clicks a button.**
- **Accepts restricted and unrestricted built-ins.**
- **Use to provide convenient navigation, to display LOVs and many other frequently used functions.**



Copyright © Oracle Corporation, 1999. All rights reserved.

Using Triggers

When-Button-Pressed Trigger

This trigger fires when the user selects a button. You can define the trigger on an individual item or at higher levels if required.

When-Button-Pressed accepts both restricted and unrestricted built-ins. You can use buttons to provide a wide range of functions for users. These functions include:

- Navigation
- Displaying LOVs
- Invoking calculations and other functions

Example

The Stock_Button in the CONTROL block is situated on the CV_INVENTORY canvas of the ORDERS form. When pressed, the button activates the When-Button-Pressed trigger. The trigger code results in navigation to the S_INVENTORY block and execution of a query on the S_INVENTORY block.

```
GO_BLOCK('S_INVENTORY');
EXECUTE_QUERY;
```

When-Window-Closed Trigger

- **Fires when the operator closes a window by using a window manager-specific close command.**
- **Accepts restricted and unrestricted built-ins.**
- **Used to programmatically close a window when the operator issues a window manager-specific close command. You can close a window by using built-ins.**

Copyright © Oracle Corporation, 1999. All rights reserved.

When-Window-Closed Trigger

This trigger fires when you close a window by using a window manager-specific close command. You define this trigger at the form level.

The When-Window-Closed trigger accepts both restricted and unrestricted built-ins.

Use this trigger to close a window programmatically when the operator issues the window manager Close command. Form Builder does not close the window when the operator issues a window manager-specific close command; it only fires When-Window-Closed trigger. It is the developer's responsibility to write the required functionality in this trigger. You can close a window with the HIDE_WINDOW, SET_WINDOW_PROPERTY, and EXIT_FORM built-in subprograms. You *cannot hide* the window that contains the current item.

Example

When the operator issues the window manager-specific Close command, the following code in a When-Window-Closed trigger closes the W_INVENTORY window by setting the VISIBLE property to FALSE.

```
GO_ITEM('S_ORD.ID'):
SET_WINDOW_PROPERTY('W_INVENTORY', VISIBLE, PROPERTY_FALSE);
```

Summary

To produce a trigger:

1. **Select a scope in the Object Navigator.**
2. **Create a trigger and select a Name from the LOV, or use the SmartTriggers menu option.**
3. **Define code in the PL/SQL Editor.**
4. **Compile.**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

- **Find built-ins in the Navigator under Built-in Packages:**
  - **Paste built-in name and arguments to your code by using the Paste Name and Arguments option.**
  - **Refer to online Help.**
- **The When-Button-Pressed trigger provides a wide range of functionality to users.**
- **Use the When-Window-Closed trigger to provide functionality when the user issues a window manager-specific close command.**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Practice 13 Overview

This practice covers the following topics:

- **Using built-ins to display LOVs**
- **Using the When-Button-Pressed and When-Window-Closed triggers to add functionality to items**
- **Using built-ins to display and hide the Help stack canvas**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 13 in Appendix A, “Practice Solutions.”

Practice 13 Overview

This practice focuses on how to use When-Button-Pressed and When-Window-Closed triggers.

- Using built-ins to display LOVs
- Using When-Button-Pressed and When-Window-Closed triggers to add functionality to items
- Using built-ins to display and hide the Help stacked canvas

Practice 13

1. In the CUSTG*XX* form, write a trigger to display the Sales_Rep_Lov when the Sales_Rep_Lov_Button is pressed. To create the When-Button-Pressed trigger, use the Smart Triggers feature. Find the relevant built-in in the Object Navigator under built-in packages, and use the "Paste Name and Arguments" feature.
2. Create a When-Window-Closed trigger at the form level in order to exit form.
3. Save, compile, and run the form.
4. In the ORDG*XX* form, write a trigger to display the Products_LOV when the Products_LOV_Button is selected.
5. Write a trigger that exits the form when the Exit_Button is selected.
6. Save, compile, and run the form.
7. Create a When-Button-Pressed trigger on the CONTROL.Show_Help_Button that uses the SHOW_VIEW built-in to display the CV_HELP.

   ```
   SHOW_VIEW('CV_HELP');
   ```

8. Create a When-Button-Pressed trigger on CONTROL.Hide_Help_Button that hides the CV_HELP. Use the HIDE_VIEW built-in to achieve this.

   ```
   HIDE_VIEW('CV_HELP');
   ```

9. Save, compile, and run the ORDG*XX* form to test.

   Note: The stacked canvas, CV_HELP, displays only if the current item will not be obscured. Ensure, at least, that the first entered item in the form is one that will not be obscured by CV_HELP.

   You might decide to advertise Help only while the cursor is in certain items, or move the stacked canvas to a position that does not overlay enterable items. The CV_HELP canvas, of course, could also be shown in its own window, if appropriate.

10. Create a When-Button-Pressed trigger on CONTROL.Stock_Button that uses the GO_BLOCK built-in to display the S_INVENTORY block.

14

Debugging Triggers

Objectives

After completing this lesson, you should be able to do the following:

- **Describe the components of the Debugger**
- **Run a form module in debug mode**
- **Debug PL/SQL code**

Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

This lesson shows you how to debug triggers by using the PL/SQL Debugger to execute code one line at a time. This lesson also shows you how to view and change variables while using the Debugger.

Debugging Triggers

Monitor and debug triggers by:

- **Compiling correct errors in the PL/SQL Editor**
- **Displaying debug messages at run time**
- **Invoking the PL/SQL Debugger**

Copyright © Oracle Corporation, 1999. All rights reserved.

Debugging Tips

- **Connect to the database for SQL compilation.**
- **The line that fails is not always responsible.**
- **Watch for missing semicolons and quotation marks.**
- **Define triggers at the correct level.**
- **Place triggers where the event will happen.**

Copyright © Oracle Corporation, 1999. All rights reserved.

Debugging Triggers

How to Debug Triggers in the Form Builder

With the Form Builder you can monitor and debug triggers in a number of ways:

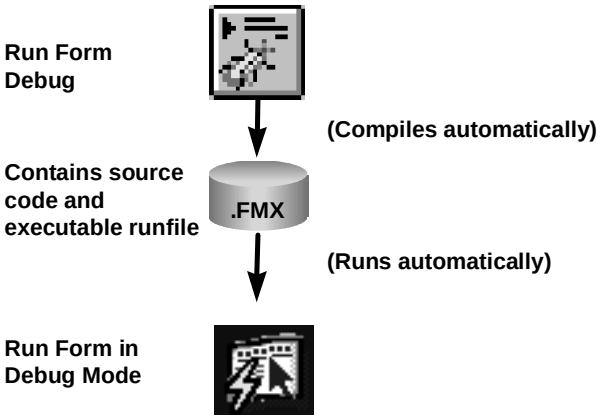
- Compiling: Syntax errors and object reference errors (including references to database objects) are reported when you compile a trigger or generate the form module. This enables you to correct these problems in the PL/SQL Editor before run time.
- Running a form with run time parameter debug_messages=Yes:

  In Debug mode, you can request messages to be displayed to indicate when triggers fire. This helps you see whether certain triggers are firing, their origin and level, and the time at which they fire.
- Invoking the PL/SQL Debugger: With the Debugger you can monitor the execution of code *within* a trigger (and other program units). You can step through the code on a line-by-line basis, and you can monitor called subprograms and variables as you do so. You can also submit arbitrary PL/SQL statements while the form is running, and modify variables.

General Tips to Solve Trigger Problems

- Make sure you are connected to the (correct) database when you compile triggers that contain SQL. Error messages can be deceiving.
- The PL/SQL Editor reports the line that *fails*, but the error may be due to a dependency on an earlier line of code.
- Missing semicolons (;) and mismatched quotes are a common cause of compile errors. Check for this if a compile error does not give an obvious indication to the problem.
- If a trigger seems to fire too often, or on the wrong block or item in the form, check whether it is defined at the required level. For example, a form-level When-Validate-Item trigger fires for every changed item in the form. To check this, you can run the form with Debug Messages on.
- For triggers that populate other items, make sure the trigger belongs to the object where the firing event will occur, *not* on the items to be populated.

Running a Form in Debug Mode



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Running a Form Module in Debug Mode

In Debug mode, you can monitor triggers that fire and use the PL/SQL Debugger. To interact with code in the Debugger, the run time module (`.fmx`) must be rebuilt to include source versions of the form code.

To run a form in Debug mode, follow these steps:

1. Click the Run Form Debug button in the Navigator, or select

   Program—>Run Form—>Debug from the menu.

   The form module is built and runs automatically.
2. When the form module is started, the PL/SQL Debugger is initially displayed so that you can enter Debug actions before the form begins running. When you dismiss the Debugger, the form is entered for running.

Displaying Messages When Triggers Fire

You can display messages that wait for acknowledgment before execution continues each time a trigger fires. These are displayed on the message line, and include the trigger's type and scope.

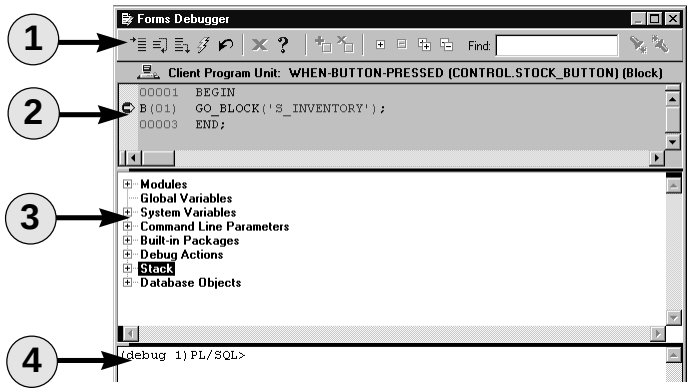
To display messages, run the form from the command line, specifying the Debug_Messages option.

For example in Microsoft Windows:

```
ifrun60 myform scott/tiger debug_messages = YES
```

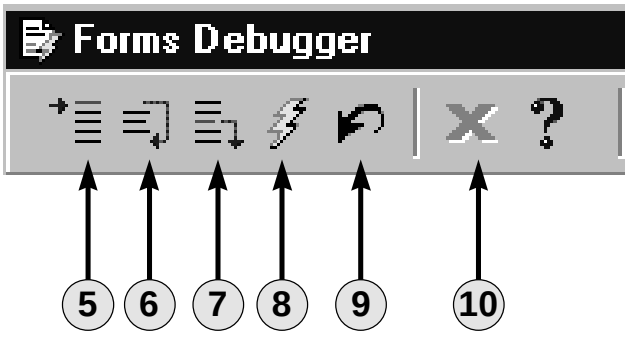
Each message is displayed just before the execution of the trigger, enabling you to see the current state of the form before the effects of the trigger.

PL/SQL Debugger



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

PL/SQL Debugger



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

The PL/SQL Debugger

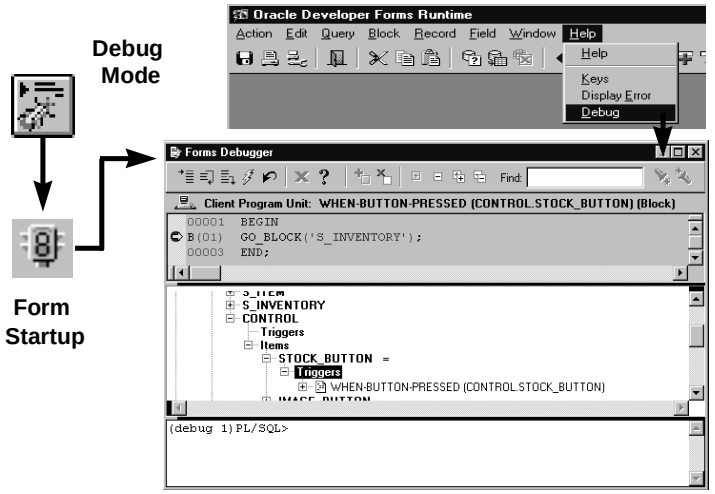
The PL/SQL Debugger enables you to perform the following tasks:

- Step through program units and examine the units as they progress.
- Examine or modify the state of variables during execution.
- Set breakpoints to suspend execution, so that you can analyze the status of the application at a given point.
- Define debug actions that will execute on certain events (Debug triggers).
- Add PL/SQL statements during execution.

The Debugger contains the following components (see slides on opposite page):

1	Navigator controls	Help, Create, Delete, Expand, Collapse, Expand All, Collapse All, Find (Control the Navigator pane as you do in the main Object Navigator.)
2	Source pane	A read-only copy of current program unit (You can select lines of code and set breakpoints in this pane.)
3	Navigator pane	Hierarchical list of programmatic objects (Functions the same as it does in main Object Navigator.)
4	Interpreter pane	Command line area where you enter PL/SQL and Debugger commands
5	Step Into (button)	Executes the STEP INTO command
6	Step Over (button)	Executes the STEP OVER command
7	Step Out (button)	Executes the STEP OUT command
8	Go (button)	Executes the GO command
9	Reset (button)	Executes the RESET command
10	Close (button)	Closes the Debugger

Invoking the Debugger and Breakpoints



Copyright © Oracle Corporation, 1999. All rights reserved.

1	Select trigger
2	Set breakpoint

Invoking the Debugger and Breakpoints

You can invoke the Debugger at any time while a form is running in Debug mode by selecting Help—>Debug from the Runform menu.

Menus and the Debugger

When control is passed to the Debugger, the Main menu includes View, Debug, and Navigator options, each providing additional submenus for controlling the Debugger.

Setting Breakpoints in a Trigger

Breakpoints invoke the Debugger during code execution, and you can analyze and interact with triggers and other program units when specific points in the code are reached. A breakpoint invokes the Debugger just before execution of the line where the breakpoint is set. You can define a breakpoint in two ways.

Method 1

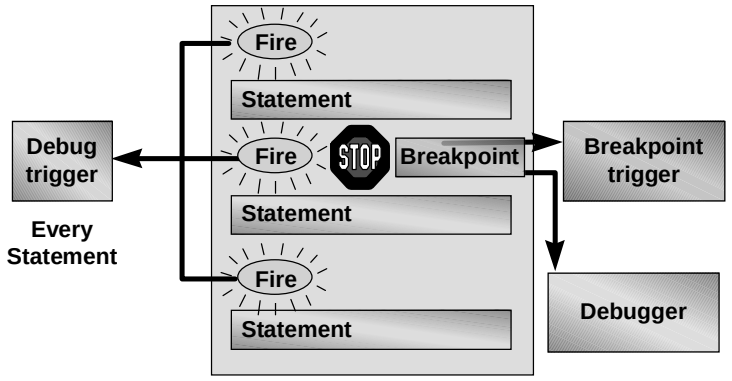
1. Select the desired trigger in the Navigator pane. The Debugger displays its source in the Source pane.
2. Double-click the line in the Source pane where the breakpoint is to be set. You can now dismiss the Debugger, and it will reappear when the specified line is reached during trigger execution.

Method 2 Other debug actions can be attached alternatively as follows:

1. Select the desired trigger, as in the first step of Method 1, and then select Debug—>Break from the menu. This invokes the PL/SQL Breakpoint dialog box.
2. In the trigger area of the Breakpoint dialog box, enter: `RAISE DEBUG.BREAK`. This raises an exception from the Debug package, which passes control to the Debugger when this line is subsequently reached during execution. With this method, you have set up a *Breakpoint trigger* from the Breakpoint dialog box, which fires each time the breakpoint is reached in the normal trigger.

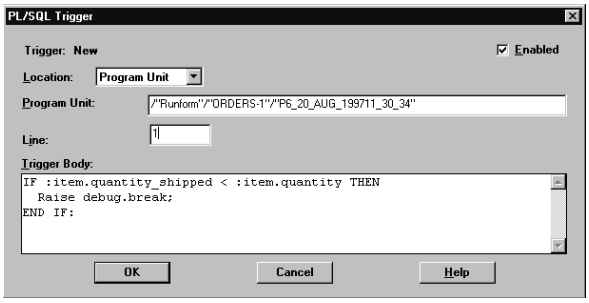
Note: Breakpoints must be attached to an executable statement in the body of the code. Comment lines or NULL commands are not valid for this purpose.

Breakpoint and Debug Triggers



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Debug Triggers



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Breakpoint Triggers and Debug Triggers

You may sometimes want to define debug actions that occur automatically, either on a breakpoint, or when certain program units and triggers are executed. Define debug actions as follows:

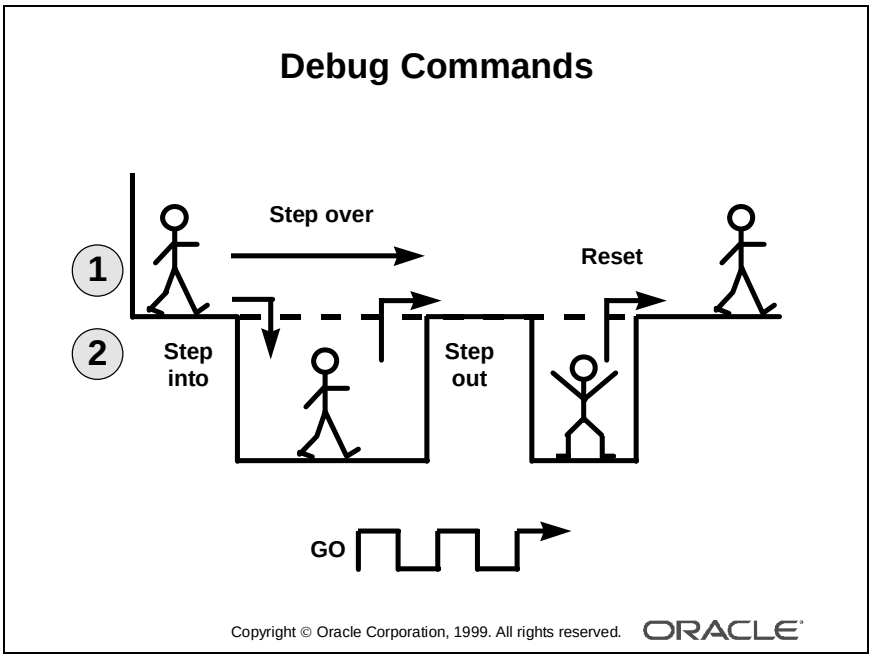
- Breakpoint triggers: A trigger that fires on a breakpoint, and contains PL/SQL and DEBUG functions and procedures.
  - Select a line of code in the Source pane, where a breakpoint is to be set.
  - Select Debug—>Break from the menu, which invokes the Breakpoint dialog box.
  - Enter your debug code in the Trigger area, then click OK.
- Debug triggers: Like Breakpoint triggers, you can define Debug triggers to fire when a breakpoint occurs. Debug triggers, however, can be attached to a program unit, or fired when each line of code is executed in that program unit.

Note: Debug triggers only cause a breakpoint if you raise the DEBUG.BREAK exception within them. Otherwise, they perform their actions in the background.

1. Select a program unit or line of source code in the Debugger.
2. Select Debug—>Trigger from the menu. This opens the PL/SQL Trigger dialog box.
3. Enter your debug code in the Trigger body area, and then click OK.

Disabling Debug Actions

To disable debug actions during run time, you clear the Enabled check box in the PL/SQL trigger or Breakpoint dialog box. You can redisplay these dialogs by locating the debug action below the Debug Actions node in the Navigator, then double-clicking on the listed action you want to display. After selecting the listed action, you can use the pop-up menu to disable or enable the selected action by clicking the right mouse button.



Note

Your trigger code is often nested in the single PL/SQL block that Form Builder provides.

1	Trigger
2	Subprograms

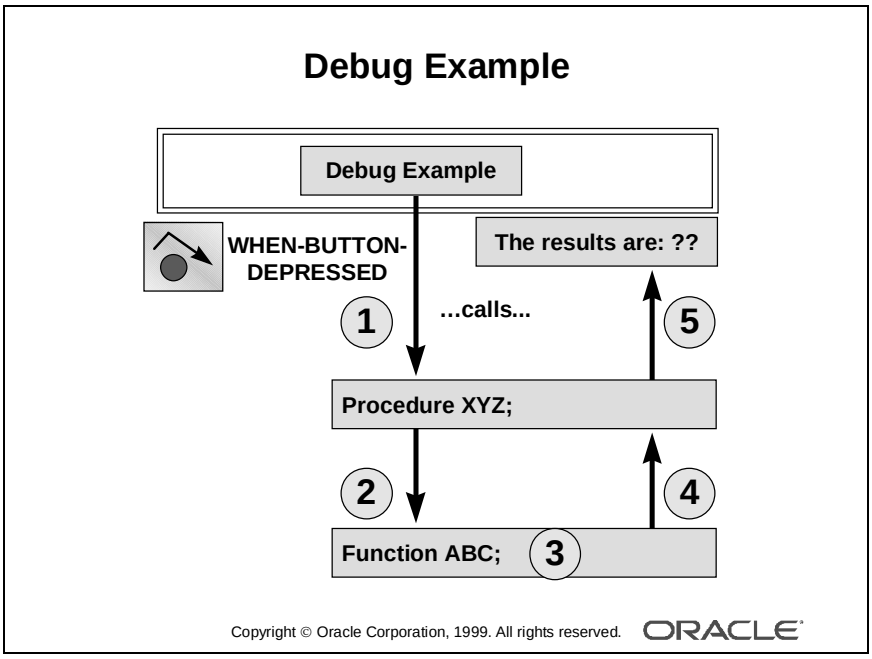
Useful Commands in the Debugger

You can enter following commands in the Interpreter pane. Those marked with asterisks (*) have an equivalent toolbar button:

Command	Description
.STEP INTO*	Advances execution into the next subprogram called by this line
.STEP OVER*	Executes the subprogram without stepping into it; stops at the next line
.STEP OUT*	Resume to end of current level (subprogram)
.GO*	Resumes execution indefinitely
.RESET*	Exits current subprogram now
.SHOW LOCALS	Displays all local variables (PL/SQL variables declared locally) and parameters

The following subprograms from the DEBUG package can be entered on the command line of the Debugger (in the Interpreter pane), or included in Debug and Breakpoint triggers. When you use them, you can display and set values for variables and parameters in the current trigger or subprogram scope:

Subprogram	Description
DEBUG.INTERPRET(string)	A procedure that lets you nest an Interpreter command (like those above) as a string, and then execute from debug triggers
DEBUG.GET***x***(varname)	A function that returns the value of variable varname. (*x* represents datatype (n for NUMBER, d for DATE, c for CHAR or VARCHAR2, i for PLS_INTEGER).)
DEBUG.SET***x***(varname,value)	A procedure that sets a specified value for a variable. (*x* represents datatype (n for NUMBER, d for DATE, c for CHAR or VARCHAR2, i for PLS_INTEGER).)


Debug Example
Debug Example
WHEN-BUTTON-DEPRESSED
The results are: ??
1
...calls...
5
Procedure XYZ;
2
4
Function ABC;
3
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Example

This simple example demonstrates some of the basic features available in the debugger. The example form consists of a single button with trigger code for the When-Button-Pressed event. The code works as follows:

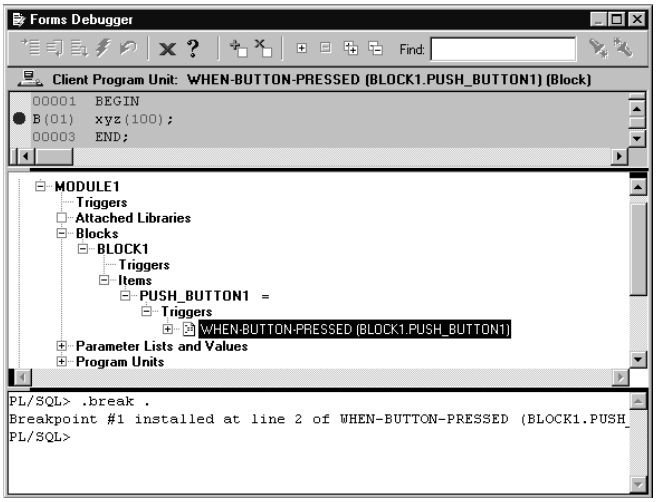
1. The trigger calls the XYZ procedure, passing it a value for the xyz_param input parameter.
2. The XYZ procedure calls the ABC function passing it a value for the abc_param input parameter.

```
PROCEDURE xyz(xyz_param IN NUMBER) IS
v_results NUMBER;
BEGIN
   v_results := ABC(10);
   v_results := v_results + xyz_param;
   MESSAGE('The results are: ' || TO_CHAR(v_results));
END xyz;
```

3. The ABC function multiplies two variables and adds the result to the abc_param input parameter.
4. The ABC function returns the result to the XYZ procedure.
5. The XYZ procedure adds the result to the xyz_param and displays it in the console at the bottom of the form window.

```
FUNCTION abc (abc_param IN NUMBER) RETURN NUMBER IS
v_total NUMBER := 0;
v_num2 NUMBER := 3;
v_num6 NUMBER := 8;
/*-- wrong value should be 6 */
BEGIN
   v_total := v_num3 * v_num6;
   v_total := v_total + abc_param;
   RETURN v_total;
END abc;
```

Debugger: Setting a Breakpoint



Copyright © Oracle Corporation, 1999. All rights reserved.

Debugger: Setting a Breakpoint

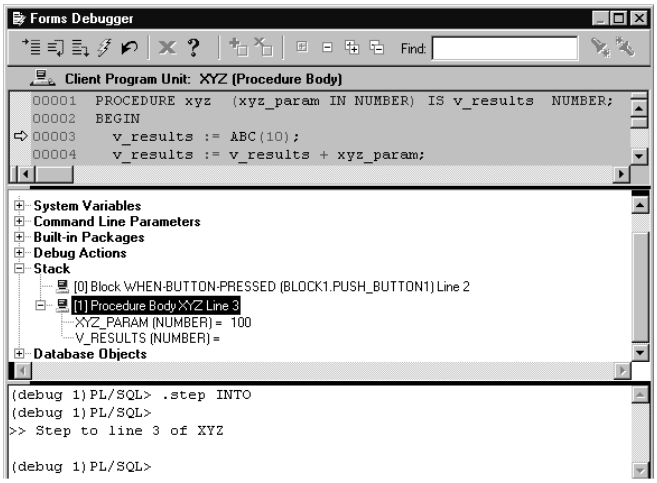
When you click the Debug Example button in the form, “134” displays at the bottom of the screen (console). However, the expected results were “128.” The syntax must be correct because everything compiled correctly. So, there must be something wrong in the logic of the application code within the form. The developer needs to debug the code to find why it produced the wrong results.

1. Run the ORDERS form in Debug mode (use the Run Form Debug button), and locate the When-Button-Pressed trigger in Block1 in the Debugger. Set a breakpoint on the executable line (Source pane) that calls the procedure:

```
00001 BEGIN
B(01) xyz(100);  -- 'B(01)' indicates the break
00003 END;
```

2. Dismiss the Debugger and the forms runs.
3. Click the Debug Example button in the form. The program stops at the breakpoint.

Debugger: Stepping into Code



Copyright © Oracle Corporation, 1999. All rights reserved.

Debugger: Stepping into Code

4 The XYZ procedure now displays in the Source pane, with "=>" to mark current position at the beginning of the executable code.

```
00001 BEGIN
```

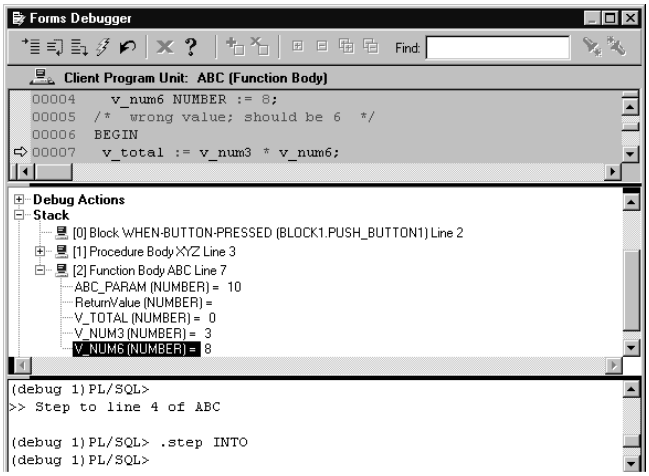
5 Click the Step Into button in the Debugger to advance into the XYZ procedure.

```
00001 PROCEDURE xyz (xyz_param IN NUMBER)IS
v_results NUMBER;
00002 BEGIN
=>003 v_results := ABC(10);
00004 v_results := v_results + xyz_param;
00005 MESSAGE('The results are: ' || TO_CHAR(v_results));
00006 END xyz;
```

6 Examine the Stack values for the xyz_param and v_results parameters (as well as system variables). Everything looks normal in the xyz procedure.

```
-Stack
-[1]Procedure Body XYZ Line 3
   |- XYZ_PARAM (NUMBER)= 100
   |- V_RESULTS (NUMBER)=
```

Debugger: Checking Variables



Copyright © Oracle Corporation, 1999. All rights reserved.

Debugger: Stepping into Code (continued)

7 Click the Step Into button to enter the ABC function. Step through each of the opening assignment statement. Find the problem in the code (v_num6 is incorrectly set to 8 instead of 6).

```
00001 FUNCTION abc (abc_param IN NUMBER) RETURN NUMBER IS
00002 v_total NUMBER := 0;
00003 v_num3 NUMBER := 3;
=>004 v_num6 NUMBER :=8;
00005 /*-- wrong value should be 6
00006 */
```

8 Check the stack values of the v_total, v_num3, and v_num6 variables.

```
-------------------------------------------------------------
-Stack
-[2]Function Body ABC Line 3
   |- ABC_PARAM (NUMBER)= 10
   |- V_TOTAL (NUMBER)=
   |- V_NUM3 (NUMBER)=
   |- V_NUM6 (NUMBER)=
-------------------------------------------------------------
```

Change the stack value of v_num6 to its *correct* value.

```
   |- V_NUM6 (NUMBER)=6
```

9 Continue to step through the ABC function using the Step Into button. Verify the stack values for the v_total variable. At the end of the ABC function, use the Step Into button to return to the XYZ procedure.

```
00003 v_results :=ABC(10)
=>004 v_results := v_results + xyz_param;
00005 MESSAGE('The results are:'||TO_CHAR(v_results));
```

Corrected Code

```
FUNCTION abc (abc_param IN NUMBER) RETURN NUMBER
IS
v_total NUMBER := 0;
v_num3 NUMBER := 3;
v_num6 NUMBER := 6;
/*-- changed value to 6
*/
BEGIN
v_total := v_num3 * v_num6;
v_total := v_total + abc_param;
RETURN v_total;
END abc;
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Debugger: Changing the Code and Rerunning

10 Go back into the ABC function in Form Builder. Change the value assigned to v_num6 from 8 to 6. Rerun the form without the Debugger on. The correct result is displayed.

```
FUNCTION abc (abc_param IN NUMBER) RETURN NUMBER IS
v_total NUMBER := 0;
v_num3 NUMBER := 3;
v_num6 NUMBER := 6;
/*-- corrected value is 6
*/
BEGIN
v_total := v_num3 * v_num6;
v_total := v_total + abc_param;
RETURN v_total;
END abc;
```

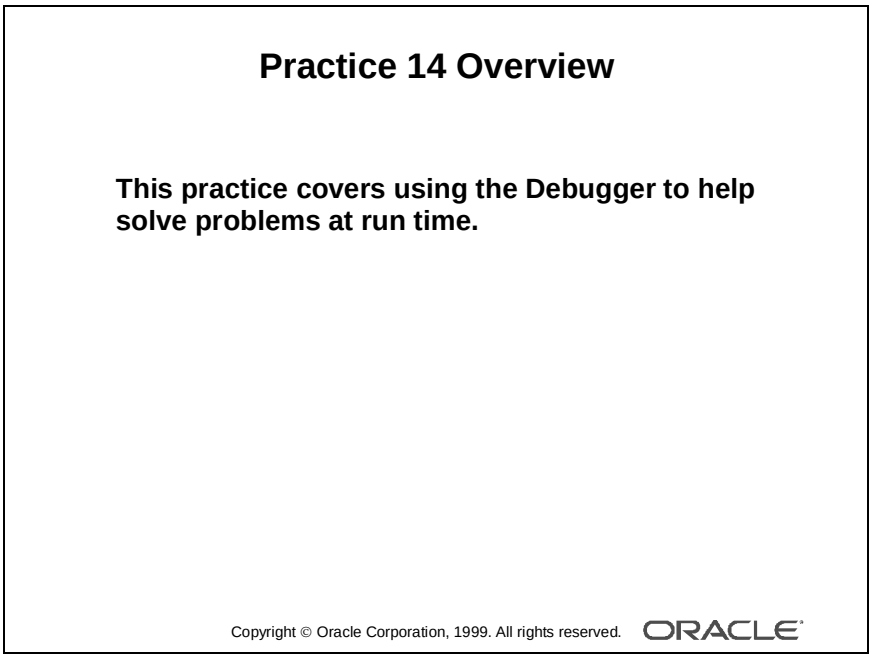
Summary

- **To debug a form: Use the Run Form Debug button, and set breakpoints.**
- **Debug commands can be entered in the Interpreter pane or by using buttons.**
- **Set breakpoints to invoke the Debugger.**
- **Break and Debug triggers are available to program Debug Actions on events in the form.**

Copyright © Oracle Corporation, 1999. All rights reserved.

Summary

- To debug a form, follow these steps:
  - **a** Use the Run Form Debug button (compiles and runs form automatically).
  - **b** Set breakpoints.
- Debug commands can be entered in the Interpreter pane or by using buttons.
- Set breakpoints to invoke the Debugger.
- Break and Debug triggers are available to program Debug Actions on events in the form.



Note

For solutions to this practice, see Practice 14 in Appendix A, “Practice Solutions.”

Practice 14 Overview

In this practice, you will create a generic procedure for the LOV in the CUSTG*XX* form, and you will run this module in Debug mode and step through the code to monitor its progress.

Use the Debugger to help solve problems at run time.

Practice 14

1. Open your CUSTG*XX*.FMB file. In this form, create a procedure that is called List_Of_Values. Import code from the pr14_1.txt file:

```
PROCEDURE list_of_values(p_lov in VARCHAR2,p_text in VARCHAR2)
IS
   v_lov BOOLEAN;
BEGIN
   v_lov:= SHOW_LOV(p_lov);
   IF v_lov THEN
      MESSAGE('You have just selected a '||p_text);
   ELSE
      MESSAGE('You have just cancelled the List of Values');
   END IF;
END;
```

2. Modify the When-Button-Pressed trigger for CONTROL.Sales_Lov_Button in order to call this procedure.

   When-Button-Pressed on CONTROL.Sales_Lov_Button

```
LIST_OF_VALUES('SALES_REP_LOV', 'Sales Representative');
```

3. Compile and run your form in Debug mode. Set a breakpoint in one of your triggers, and investigate the call stack. Try stepping through the code to monitor its progress.

15

Adding Functionality to Items

Objectives

After completing this lesson, you should be able to do the following:

- **Supplement the functionality of input items by using triggers and built-ins**
- **Supplement the functionality of noninput items by using triggers and built-ins**

Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

In this lesson, you will learn how to use triggers to provide additional functionality to GUI items in form applications.

Item Interaction Triggers

- When-Button-Pressed
- When-Checkbox-Changed
- When-Radio-Changed
- When-Image-Pressed
- When-Image-Activated
- When-List-Changed
- When-List-Activated

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Item Interaction Triggers

There are several types of GUI items that the user can interact with by using the mouse or by pressing a function key. Most of these items have default functionality. For example, by selecting a radio button, the user can change the value of the radio group item.

You will often want to add triggers to provide customized functionality when these events occur. For example:

- Performing tests and appropriate actions as soon as the user clicks a radio button, a list, or a check box
- Conveniently displaying an image when the user clicks an image item
- Defining the functionality of a push-button (which has none until you define it)

The following triggers fire due to user interaction with an item, as previously described. They can be defined at any scope.

Trigger	**Firing Event**
When-Button-Pressed	User single-clicks with mouse or uses function key to select
When-Checkbox-Changed	User changes check box state, by single-click or function key
When-Radio-Changed	User selects different button, or deselects current button, in a radio group
When-Image-Pressed	User single-clicks image item
When-Image-Activated	User double-clicks image item
When-List-Changed	User changes value of a list item
When-List-Activated	User double-clicks element in a T-list

Coding Item Interaction Triggers

- **Valid commands:**
  - **SELECT statements**
  - **Standard PL/SQL constructs**
  - **All built-in subprograms**
- **Use When-Validate-"*object*" to trap the operator during validation.**

Copyright © Oracle Corporation, 1999. All rights reserved.

Example of When-Radio-Changed

When-Radio-Changed trigger on :S_ORD.Payment_Type. When the user selects credit as the payment type for an order, this trigger immediately confirms whether the customer has a good or excellent credit rating. If not, then the payment type is set to cash.

```
DECLARE
   v_credit_rating S_CUSTOMER.credit_rating%TYPE;
BEGIN
   IF :S_ORD.payment_type = 'CREDIT' THEN
      SELECT credit_rating INTO v_credit_rating
      FROM S_CUSTOMER
      WHERE id = :S_ORD.customer_id;
      IF v_credit_rating NOT IN('GOOD','EXCELLENT') THEN
         :S_ORD.payment_type := 'CASH';
         MESSAGE('Warning-customer must pay cash');
      END IF;
   END IF;
END;
```

Note: During an unhandled exception, the trigger terminates and sends the Unhandled Exception message to the operator. The item interaction triggers do not fire on navigation or validation events.

Command Types in Item Interaction Triggers

You can use standard SQL and PL/SQL statements in these triggers, like the example above. However, you will often want to add functionality to items by calling built-in subprograms, which provide a wide variety of mechanisms.

Interacting with Check Boxes

When-Checkbox-Changed

```
IF CHECKBOX_CHECKED('S_ORD.order_filled') THEN
  SET_ITEM_PROPERTY('S_ORD.date_shipped',
  UPDATE_ALLOWED, PROPERTY_FALSE);
ELSE
  SET_ITEM_PROPERTY('S_ORD.date_shipped',
  UPDATE_ALLOWED, PROPERTY_TRUE);
END IF;
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Defining Functionality for Input Items

You have already seen an example of adding functionality to radio groups; we now look at adding functionality to other items that accept user input.

Check Boxes

When the user selects or clears a check box, the associated value for the state is set. You may want to perform trigger actions based on this change. Note that the CHECKBOX_CHECKED function enables you to test the state of a check box without needing to know the associated values for the item.

Example

This When-Checkbox-Changed trigger on the :S_ORD.Order_Filled item prevents the Date_Shipped item from being updated if the user marks the order as filled (checked on). If the check box is set to off, then the Date_Shipped is enabled.

```
IF CHECKBOX_CHECKED('S_ORD.order_filled') THEN
   SET_ITEM_PROPERTY('S_ORD.date_shipped',
      UPDATE_ALLOWED,PROPERTY_FALSE );
ELSE
   SET_ITEM_PROPERTY('S_ORD.date_shipped',
      UPDATE_ALLOWED, PROPERTY_TRUE );
END IF;
```

Changing List Items at Run Time

ADD_LIST_ELEMENT

DELETE_LIST_ELEMENT

Copyright © Oracle Corporation, 1999. All rights reserved.

List Items

You can use the When-List-Changed trigger to trap user selection of a list value. For T-lists, you can trap double-clicks with When-List-Activated.

With Form Builder, you can also change the selectable elements in a list as follows:

- Periodically update the list from a two-column record group.
- Add or remove individual list elements through the ADD_ LIST_ELEMENT and DELETE_LIST_ELEMENT built-ins, respectively.

```
ADD_LIST_ELEMENT('list_item_name',index,'label','value')
DELETE_LIST_ELEMENT('list_item_name',index)
```

Parameter	Description
Index	A number identifying the element position in the list (top=1)
Label	The name of the element
Value	The new value for this element

Note: You can eliminate the Null list element of a list by setting the required property to Yes.

Displaying LOVs from Buttons

- **Uses:**
  - **Convenient alternative for accessing LOVs**
  - **Can display independently of text items**
- **Needs:**
  - **When-Button-Pressed trigger**
  - **LIST_VALUES or SHOW_LOV built-in**

Copyright © Oracle Corporation, 1999. All rights reserved.

Defining Functionality for Noninput Items

Displaying LOVs from Buttons

If you have attached a LOV to a text item, then the user can invoke the LOV from the text item by selecting Edit—>Display List or pressing the [List Values] key.

However, it is always useful if a button is available to display a LOV. The button has two advantages:

- It is convenient alternative for accessing the LOV.
- It displays a LOV independently of a text item (using SHOW_LOV).

There are two built-ins that you can call to invoke a LOV from a trigger. These are LIST_VALUES and SHOW_LOV.

LIST_VALUES Procedure

This built-in procedure invokes the LOV that is attached to the current text item in the form. It has an optional argument, which may be set to RESTRICT, meaning that the current value of the text item is used as the initial search string on the LOV. The default for this argument is NO_RESTRICT.

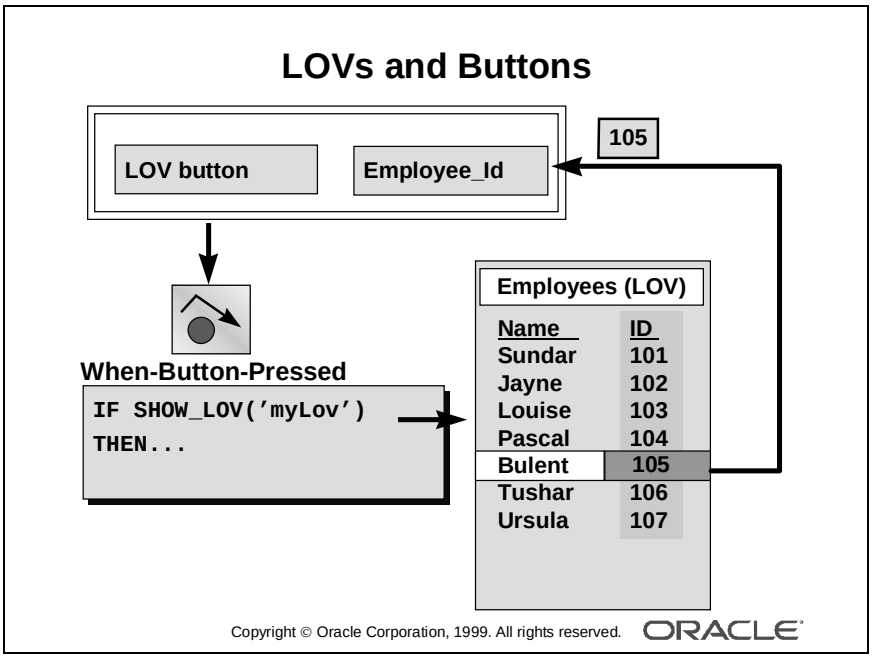
SHOW_LOV Function

This built-in function, without arguments, invokes the LOV of the current item. However, there are arguments that let you define which LOV is to be displayed, and what the x and y coordinates are where its window should appear:

```
SHOW_LOV( 'lov_name', x, y )
SHOW_LOV( lov_id, x, y )
```

Notice that either the LOV name (in quotes) or the LOV ID (without quotes) can be supplied in the first argument.

Note: The lov_id is a PL/SQL variable where the internal ID of the object is stored. Internal IDs are a more efficient way of identifying an object.


LOVs and Buttons
LOV button
Employee_Id
105
When-Button-Pressed
IF SHOW_LOV('myLov')
THEN...
Employees (LOV)
Name
ID
Sundar 101
Jayne 102
Louise 103
Pascal 104
Bulent 105
Tushar 106
Ursula 107
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Using the SHOW_LOV Function

The SHOW_LOV function returns a Boolean value:

- TRUE indicates that the user selected a record from the LOV.
- FALSE indicates that the user dismissed the LOV without choosing a record, or that the LOV returned 0 records from its Record Group.

Note

- You can use the FORM_SUCCESS function to differentiate between the two causes of SHOW_LOV returning FALSE.

  Create the LOV button with a suitable label, such as “Pick,” and arrange it on the canvas where the user intuitively associates it with the items that the LOV supports (even though the button has no direct connection with text items). This is usually adjacent to the main text item that the LOV returns a value to.

  You can use the SHOW_LOV function to display a LOV that is not even attached to a text item, providing that you identify the LOV in the first argument of the function. When called from a button, this invokes the LOV to be independent of cursor location.
- Switch off the button’s Mouse Navigate property of the button. When using LIST_VALUES, the cursor needs to reside in the text item that is attached to the LOV. With SHOW_LOV, this also maintains the cursor to in its original location after the LOV is closed, wherever that may be.

Example

This When-Button-Pressed trigger on the Customer_Lov_Button invokes an LOV in a PL/SQL loop, until the function returns TRUE. Because SHOW_LOV returns TRUE when the user selects a record, the LOV redisplays until they do so.

```
LOOP
   EXIT WHEN SHOW_LOV( 'customer_lov' );
   MESSAGE('You must select a value from list');
END LOOP;
```

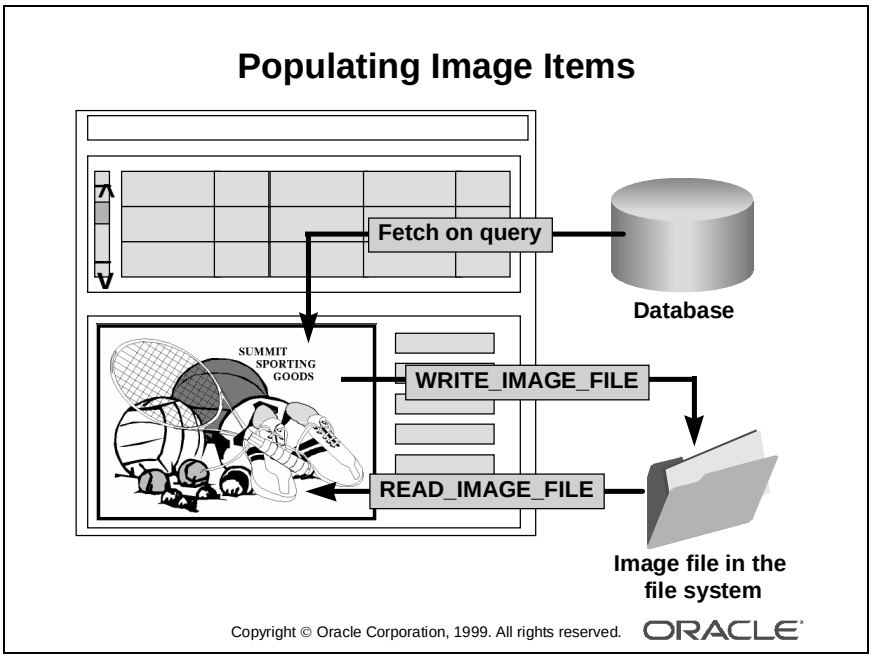

Populating Image Items
Fetch on query
Database
SUMMIT
SPORTING
GOODS
WRITE_IMAGE_FILE
READ_IMAGE_FILE
Image file in the
file system
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Image Items

Image items that have the Database Item property set to Yes automatically populate in response to a query in the owning block (from a LONG RAW column in the base table).

Nonbase table image items, however, need to be populated by other means. For example, from an image file in the file system:
READ_IMAGE_FILE built-in procedure

You might decide to populate an image item from a button trigger, using When-Button-Pressed, but there are two triggers that fire when the user interacts with an image item directly:

- When-Image-Pressed (fires for a single click on image item)
- When-Image-Activated (fires for a double-click on image item)

READ_IMAGE_FILE Procedure

This built-in procedure lets you load an image file, in a variety of formats, into an image item.

```
READ_IMAGE_FILE('filename','filetype','item_name');
```

Parameter	Description
filename	The image file name (Without a specified path, the default path is assumed.)
filetype	The file type of the image (You can use ANY as a value, but it is recommended to set a specific file type for better performance. Refer to the online Help system for file types.)
item_name	The name of the image item (a variable holding the Item_id is also valid for this argument) (This parameter is optional.)

Note

- The filetype parameter is optional in READ_IMAGE_FILE. If you omit filetype, you must explicitly identify the item_name parameter.
- The reverse procedure, WRITE_IMAGE_FILE, is also available. You can use GET_FILE_NAME built-in to display the standard open file dialog box where the user can select an existing file or specify a new file.

Loading the Right Image

```
READ_IMAGE_FILE(
  'F_'||TO_CHAR(:S_ITEM.product_id)||'.BMP',
  'BMP',
  'S_ITEM.product_image' );
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Example of Image Items

The following When-Image-Pressed trigger on the Product_Image item displays a picture of the current product (in the ITEM block) when the user clicks the image item. This example assumes that the related filenames have the format:

F_<product id>.BMP

```
READ_IMAGE_FILE('F_'||TO_CHAR(:S_ITEM.product_id)||'.BMP',
      'BMP','S_ITEM.product_image' );
```

Notice that as the first argument to this built-in is datatype CHAR, the concatenated NUMBER item, product_id, must first be converted by using the TO_CHAR function.

Note: If you load an image into a base table image item by using READ_IMAGE_FILE, then its contents will be committed to the database LONG RAW column when you save changes in the form. You can use this technique to populate a table with images.

Interacting with Sound Items

GET_ITEM_PROPERTY and SET_ITEM_PROPERTY:

- **SHOW_FAST_FORWARD_BUTTON**
- **SHOW_PLAY_BUTTON**
- **SHOW_RECORD_BUTTON**
- **SHOW_REWIND_BUTTON**
- **SHOW_SLIDER**
- **SHOW_TIME_INDICATOR** Total: 1.25 sec
- **SHOW_VOLUME_CONTROL**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Interacting with Sound Items

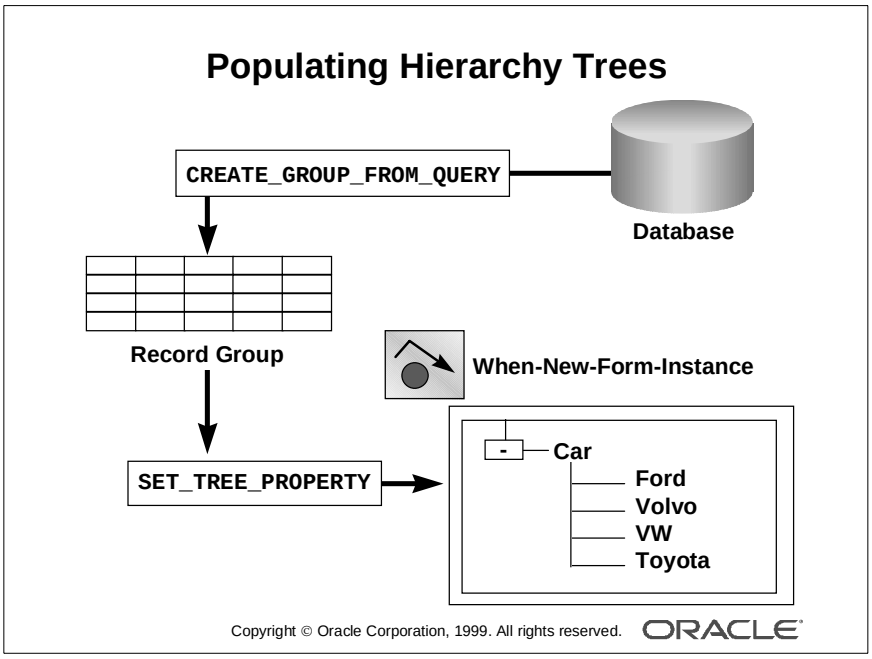
When you create a sound item, Form Builder automatically represents the item in the layout with the sound item control.

You can hide or display or get information about each component of a sound item control programmatically by using SET_ITEM_PROPERTY and GET_ITEM_PROPERTY built-ins (however, you must always have either the Play or Record button displayed for a sound item).

You can use the following properties with these two built-ins:

- SHOW_FAST_FORWARD_BUTTON
- SHOW_PLAY_BUTTON
- SHOW_RECORD_BUTTON
- SHOW_REWIND_BUTTON
- SHOW_SLIDER
- SHOW_TIME_INDICATOR
- SHOW_VOLUME_CONTROL

Use the PROPERTY_TRUE or PROPERTY_FALSE parameters with the SET_ITEM_PROPERTY built-in. The GET_ITEM_PROPERTY built-in returns TRUE or FALSE as data type VARCHAR2.


Populating Hierarchy Trees
CREATE_GROUP_FROM_QUERY
Database
Record Group
When-New-Form-Instance
SET_TREE_PROPERTY
Car
Ford
Volvo
VW
Toyota
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Populating Hierarchical Trees

The hierarchical tree displays data in the form of a standard navigator, similar to the Object Navigator used in Oracle Developer.

You can populate a hierarchical tree with values contained in a Record Group or Query Text. At run time, you can programmatically add, remove, modify, or evaluate elements in a hierarchical tree. You can also use the property palette to populate the hierarchical tree.

Note: All built-ins are located in the FTREE built-in package.

SET_TREE_PROPERTY Procedure

This built-in procedure can be used to change certain properties for the indicated hierarchical tree item It can also be used to populate the indicated hierarchical tree item from a record group.

```
Ftree.Set_Tree_Property(item_name, Ftree.property, value);
```

Parameter	Description
item_name	Specifies the name of the object created at design time. The data type of the name is VARCHAR2.
property	Specifies one of the following properties: RECORD_GROUP: Replaces the data set of the hierarchical tree with a record group and causes it to display QUERY_TEXT: Replaces the data set of the hierarchical tree with an SQL query and causes it to display ALLOW_EMPTY_BRANCHES: Possible values are PROPERTY_TRUE and PROPERTY_FALSE
value	Specifies the value appropriate to the property you are setting: PROPERTY_TRUE: The property is set to the TRUE state. PROPERTY_FALSE: The property is set to the FALSE state.

Displaying Hierarchy Trees

WHEN-NEW-FORM-INSTANCE

```
rg_emps := create_group_from_query('rg_emps'
           'select 1, level, last_name, NULL,
           to_char(id) ' ||
           'from s_emp ' ||
           'connect by prior id= manager_id '||
           'start with title = ''President''');

v_ignore := populate_group(rg_emps);

ftree.set_tree_property('block4.tree5',
           ftree.record_group, rg_emps);
```

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Populating Hierarchical Trees (continued)

You can add data to a tree view by:

- Populating a tree with values contained in a record group or query by using the POPULATE_TREE built-in
- Adding data to a tree under a specific node by using the ADD_TREE_DATA built-in
- Modifying elements in a tree at run time by using built-in subprograms
- Adding or deleting nodes and the data elements under the nodes

Example

This code could be used in a WHEN-NEW-FORM-INSTANCE trigger to initially populate the hierarchical tree with data. The example locates the hierarchical tree first. Then, a record group is created and the hierarchical tree is populated.

```
DECLARE

htree       ITEM;
v_ignore    NUMBER;
rg_emps     RECORDGROUP;

BEGIN
htree := Find_Item('tree_block.htree3');

rg_emps := Create_Group_From_Query('rg_emps',
’select 1, level, ename, NULL, to_char(empno) ’ ||
'from emp '||
'connect by prior empno = mgr '||
’start with job = ’’PRESIDENT’’’);

v_ignore := Populate_Group(rg_emps);

Ftree.Set_Tree_Property(htree, Ftree.RECORD_GROUP, rg_emps);
END;
```

Summary

- **Item interaction triggers accept SELECT statements and other standard PL/SQL constructs.**
- **You use built-ins for check boxes, LOV control, list item control, image file reading, hierarchical tree, and sound item control.**

Copyright © Oracle Corporation, 1999. All rights reserved.

Summary

In this lesson, you learned to use triggers to provide functionality to the GUI items in form applications.

- The item interaction triggers accept SELECT statements and other standard PL/SQL constructs.
- There are built-ins for LOV control, list item control, image file reading, sound item control, hierarchical tree, and so on.

Practice 15 Overview

This practice covers the following topics:

- **Writing a trigger to check whether the customer's credit rating forces him to pay cash**
- **Creating a toolbar button to display and hide product images**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 15 in Appendix A, "Practice Solutions."

Practice 15 Overview

In this practice, you add some triggers that enable interaction with buttons. You also create some additional functionality for a radio group.

- Writing a trigger to check whether the customer’s credit rating forces him to pay cash
- Creating a toolbar button to display and hide product images

Practice 15

1. In the ORDG*XX* form write a trigger that fires when the Payment Type changes, allowing only those customers with a good or excellent Credit Rating to pay for orders on credit. You can import the `pr15_1.txt` file.
2. In the CONTROL block, create a new button called Image_Button and position it on the Toolbar. Set Label property to Image Off.
3. Import the file `pr15_3.txt` into a trigger that fires when the Image_Button is clicked. The file contains code that determines the current value of the visible property of the Product Image item. If the current value is True, the visible property toggles to False for both the Product Image item and the Image Description item. Finally the label changes on the Image_Button to reflect its next toggle state. However, if the visible property is currently False, the visible property toggles to True for both the Product Image item and the Image Description item.

When-Button-Pressed on CONTROL.Image_Button

```
IF GET_ITEM_PROPERTY('S_ITEM.product_image',VISIBLE)='TRUE' THEN
   SET_ITEM_PROPERTY('S_ITEM.product_image', VISIBLE,
   PROPERTY_FALSE);
   SET_ITEM_PROPERTY('S_ITEM.image_description', VISIBLE,
   PROPERTY_FALSE);
   SET_ITEM_PROPERTY('CONTROL.image_button',LABEL,'Image On');
ELSE
   SET_ITEM_PROPERTY('S_ITEM.product_image', VISIBLE,
   PROPERTY_TRUE);
   SET_ITEM_PROPERTY('S_ITEM.image_description', VISIBLE,
   PROPERTY_TRUE);
   SET_ITEM_PROPERTY('CONTROL.image_button',LABEL,
   'Image Off');
END IF;
```

4. Save, compile, and run the form.

16

Runform Messages and Alerts

Objectives

After completing this lesson, you should be able to do the following:

- **Describe the default messaging**
- **Handle errors using built-in subprograms**
- **Identify the different types of Form Builder messages**
- **Control system messages**
- **Create and control alerts**

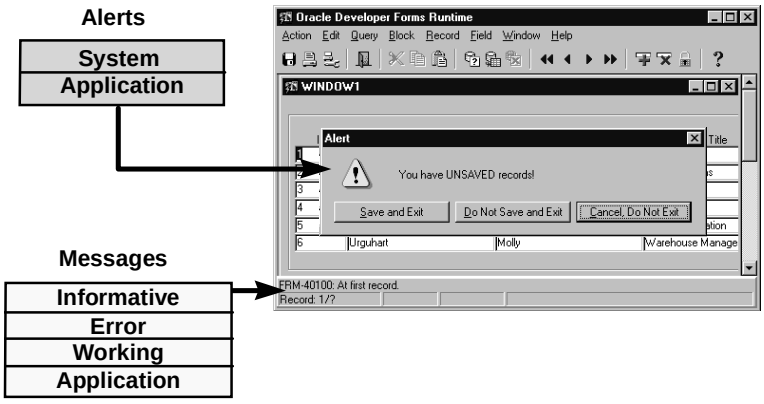
Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

This lesson shows you how to intercept system messages, and if desired, replace them with ones that are more suitable for your application. You will also learn how to handle errors by using built-in subprograms, and how to build customized alerts for communicating with users.

Communicating with the Operator



Copyright © Oracle Corporation, 1999. All rights reserved.

Run-time Messages and Alerts Overview

The Form Builder displays messages at run time to inform the operator of events that occur in the session. As the designer, you may want to either suppress or modify some of these messages, depending on the nature of the application.

The Form Builder can communicate with the user in the following ways:

- Informative message: A message tells the user the current state of processing, or gives context-sensitive information. The default display is on the message line. You can suppress its appearance with an On-Message trigger.
- Error message: This informs the user of an error that prevents the current action. The default display is on the message line. You can suppress message line errors with an On-Error trigger.
- Working message: This tells the operator that the form is currently processing (for example: Working...). This is shown on the message line. This type of message can be suppressed by setting the system variable SUPPRESS_WORKING to True.
- System alert: Alerts give information to the operator that require either an acknowledgment or an answer to a question before processing can continue. This is displayed as a modal window. When more than one message is waiting to show on the message line, the current message also displays as an alert.

You can also build messages and alerts into your application:

- Application message: These are messages that you build into your application by using the MESSAGE built-in. The default display is on the message line.
- Application alert: These are alerts that you design as part of your application, and issue to the operator for a response by using the SHOW_ALERT built-in.

Detecting Run Time Errors

- **FORM_SUCCESS**
  - **TRUE: Action successful**
  - **FALSE: Error/Fatal error occurred**
- **FORM_FAILURE**
  - **TRUE: A nonfatal error occurred**
  - **FALSE: No error/No fatal error**
- **FORM_FATAL**
  - **TRUE: A fatal error occurred**
  - **FALSE: No error/No nonfatal error**

Copyright © Oracle Corporation, 1999. All rights reserved.

Built-ins and Handling Errors

When a built-in subprogram fails, it does not directly cause an exception in the calling trigger or program unit. This means that subsequent code continues after a built-in fails, unless you take action to detect a failure.

Example

A button in the CONTROL block called Stock_Button is situated on the Toolbar canvas of the ORDERS form. When clicked, this When-Button-Pressed trigger navigates to the S_INVENTORY block, and performs a query there.

```
GO_BLOCK('S_INVENTORY');
EXECUTE_QUERY;
```

If the GO_BLOCK built-in procedure fails because the S_INVENTORY block does not exist, or because it is nonenterable, then the EXECUTE_QUERY procedure still executes, and attempts a query in the wrong block.

Built-in Functions for Detecting Success and Failure

The Form Builder supplies some functions that indicate whether the *latest action* in the form was successful:

Built-in Function	**Description of Returned Value**
FORM_SUCCESS	TRUE: Action successful FALSE: Error or fatal error occurred
FORM_FAILURE	TRUE: A nonfatal error occurred FALSE: Either no error, or a fatal error
FORM_FATAL	TRUE: A fatal error occurred FALSE: Either no error, or a nonfatal error

Note: These built-in functions return success or failure of the latest action in the form. The failing action may occur in a trigger that fired as a result of a built-in from the first trigger. For example, the EXECUTE_QUERY procedure, can cause a Pre-Query trigger to fire, which may itself fail.

Errors and Built-Ins

- **Built-in failure does not cause an exception.**
- **Test built-in success with FORM_SUCCESS function.**

  IF FORM_SUCCESS THEN . . .
- **What went wrong?**
  - **ERROR_CODE, ERROR_TEXT, ERROR_TYPE**
  - **MESSAGE _CODE, MESSAGE _TEXT, MESSAGE _TYPE**

Copyright © Oracle Corporation, 1999. All rights reserved.

Errors and Built-Ins

It is usually most practical to use FORM_SUCCESS, because this returns FALSE if either a fatal *or* a nonfatal error occurs. You can then code the trigger to take appropriate action.

Example of FORM_SUCCESS

Here is the same trigger again. This time, the FORM_SUCCESS function is used in a condition to decide if the query should be performed, depending on the success of the GO_BLOCK action.

```
GO_BLOCK('S_INVENTORY');
IF FORM_SUCCESS THEN
   EXECUTE_QUERY;
ELSE
   MESSAGE('An error occurred while navigating to Stock');
END IF;
```

Triggers fail only if there is an unhandled exception or you raise the FORM_TRIGGER_FAILURE exception to fail the trigger in a controlled manner.

Note: Look at the program unit CHECK_PACKAGE_FAILURE, which is written for you as part of Relation management, when you build master-detail blocks. This procedure may be called to fail a trigger if the last action was unsuccessful.

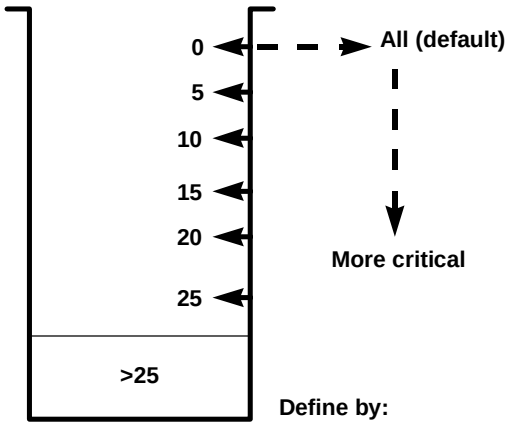
Built-in Functions to Determine the Error

When you detect an error, you may need to identify it to take a specific action. Three more built-in functions provide this information:

Built-in Function	**Description of Returned Value**
ERROR_CODE	Error number (datatype NUMBER)
ERROR_TEXT	Error description (datatype CHAR)
ERROR_TYPE	FRM=Form Builder error, ORA=Oracle error (datatype CHAR)

We will look at these built-ins again when we discuss controlling messages.

Message Severity Levels



Copyright © Oracle Corporation, 1999. All rights reserved.

Controlling System Messages

Suppressing Messages According to Their Severity

You can prevent system messages from being issued, based on their severity level. Form Builder classifies every message with a severity level that indicates how critical or trivial the information is; the higher the numbers, the more critical the message. There are six levels that you can affect.

Severity Level	Description
0	All messages
5	Reaffirms an obvious condition
10	User has made a procedural mistake
15	User attempting action for which the form is not designed
20	Cannot continue intended action due to a trigger problem or some other outstanding condition
25	A condition that could result in the form performing incorrectly
>25	Messages that the designer cannot suppress

In a trigger, you can specify that only messages above a specified severity level are to be issued by the form. You do this by assigning a value to the system variable MESSAGE_LEVEL. Form Builder then only issues messages that are above the severity level defined in this variable.

The default value for MESSAGE_LEVEL (at form startup) is 0. This means that messages of all severities are displayed.

Suppressing Messages

```
:SYSTEM.MESSAGE_LEVEL := '5';
UP;
IF NOT FORM_SUCCESS THEN
  MESSAGE('Already at the first
  Order');
END IF;
:SYSTEM.MESSAGE_LEVEL := '0';
```

```
:SYSTEM.SUPPRESS_WORKING := 'TRUE';
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Example of Suppressing Messages

The following When-Button-Pressed trigger moves up one record, using the built-in procedure UP. If the cursor is already on the first record, the built-in fails and the following message usually displays: `FRM-40100: At first record`.

This is a severity level 5 message. However the trigger suppresses this, and outputs its own application message instead. The trigger resets the message level to normal (0) afterwards.

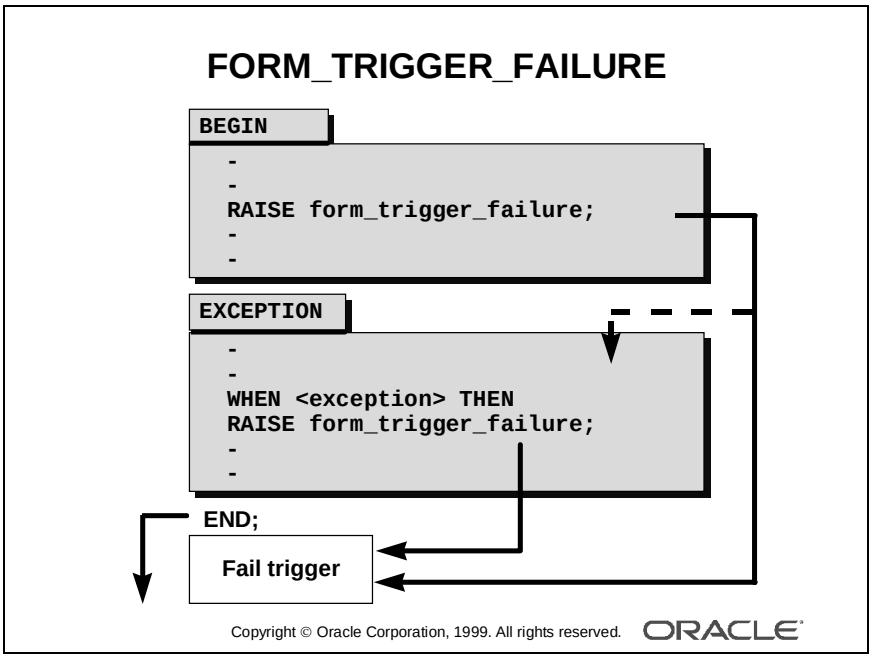
```
:SYSTEM.MESSAGE_LEVEL := '5';
UP;
IF NOT FORM_SUCCESS THEN
   MESSAGE('Already at the first Order');
END IF;
:SYSTEM.MESSAGE_LEVEL := '0';
```

Suppressing Working Messages

Working messages are displayed when the Form Builder is busy processing an action. For example, while querying you receive: `Working...` message. You can suppress this message by setting the system variable SUPPRESS_WORKING to True.

```
:SYSTEM.SUPPRESS_WORKING := 'TRUE';
```

Note: You can set these system variables as soon as the form starts up, if required, by performing the assignments in a When-New-Form-Instance trigger.


FORM_TRIGGER_FAILURE
BEGIN
-
-
RAISE form_trigger_failure;
-
-
EXCEPTION
-
-
WHEN <exception> THEN
RAISE form_trigger_failure;
-
-
END;
Fail trigger
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

The FORM_TRIGGER_FAILURE Exception

Triggers only fail when one of the following occurs:

- During an Unhandled Exception
- When you request Form Builder to fail the trigger by raising the built-in exception FORM_TRIGGER_FAILURE

This exception is defined and handled by Form Builder, beyond the visible trigger text that you write. You can raise this exception:

- In the executable part of a trigger, to skip remaining actions and fail the trigger
- In an exception handler, to fail the trigger *after* your own exception handling actions have been obeyed

In either case, Form Builder has its own exception handler for FORM_TRIGGER_ FAILURE, which fails the trigger but does *not* cause an unhandled exception. This means that you can fail the trigger in a controlled manner.

Example

This example adds an action to the trigger exception handler, raising an exception to fail the trigger when the message is sent, and therefore trapping the user in the Customer_ID item:

```
SELECT name, phone
INTO :S_ORD.customer_name, :S_ORD.customer_phone
FROM S_CUSTOMER WHERE id = :S_ORD.customer_id;
EXCEPTION
WHEN no_data_found THEN
   MESSAGE('Customer with this ID not found');
   RAISE form_trigger_failure;
```

Error Triggers

- **On-Error:**
  - **Fires when a system error message is issued**
  - **Is used to trap Form Builder and Oracle Server errors, and to customize error messages**
- **On-Message:**
  - **Fires when an informative system message is issued**
  - **Is used to suppress or customize specific messages**

Copyright © Oracle Corporation, 1999. All rights reserved.

Triggers for Intercepting System Messages

By writing triggers that fire on message events you can intercept system messages before they are displayed on the screen. These triggers are:

- On-Error: Fires on display of a system error message
- On-Message: Fires on display of an informative system message

These triggers replace the display of a message, so that no message is seen by the operator unless you issue one from the trigger itself.

You can define these triggers at any level. For example, an On-Error trigger at item level only intercepts error messages that occur while control is in that item. However, if you define one or both of these triggers at form level, all messages that cause them to fire will be intercepted regardless of which object in the current form causes the error or message.

On-Error Trigger

Use this trigger to:

- Detect Form Builder and Oracle Server errors. This trigger can perform corrective actions based on the error that occurred.
- Replace the default error message with a customized message for this application.

Remember that you can use the built-in functions ERROR_CODE, ERROR_ TEXT, and ERROR_TYPE to identify the details of the error, and possibly use this information in your own message.

Example of an On-Error Trigger

This On-Error trigger sends a customized message for error 40202 (field must be entered), but reconstructs the standard system message for all other errors.

```
IF ERROR_CODE = 40202 THEN
   MESSAGE('You must fill in this field for an Order');
ELSE
   MESSAGE(ERROR_TYPE || '-' || TO_CHAR(ERROR_CODE) || ': ' ||
   ERROR_TEXT);
END IF;
RAISE FORM_TRIGGER_FAILURE;
```

Handling Informative Messages

- **On-Message trigger**
- **Built-in functions:**
  - **MESSAGE_CODE**
  - **MESSAGE_TEXT**
  - **MESSAGE_TYPE**

Copyright © Oracle Corporation, 1999. All rights reserved.

On-Message Trigger

Use this trigger to suppress informative messages, replacing them with customized application messages, as appropriate.

You can handle messages in On-Message in a similar way to On-Error. However, because this trigger fires due to informative messages, you will use different built-ins to determine the nature of the current message.

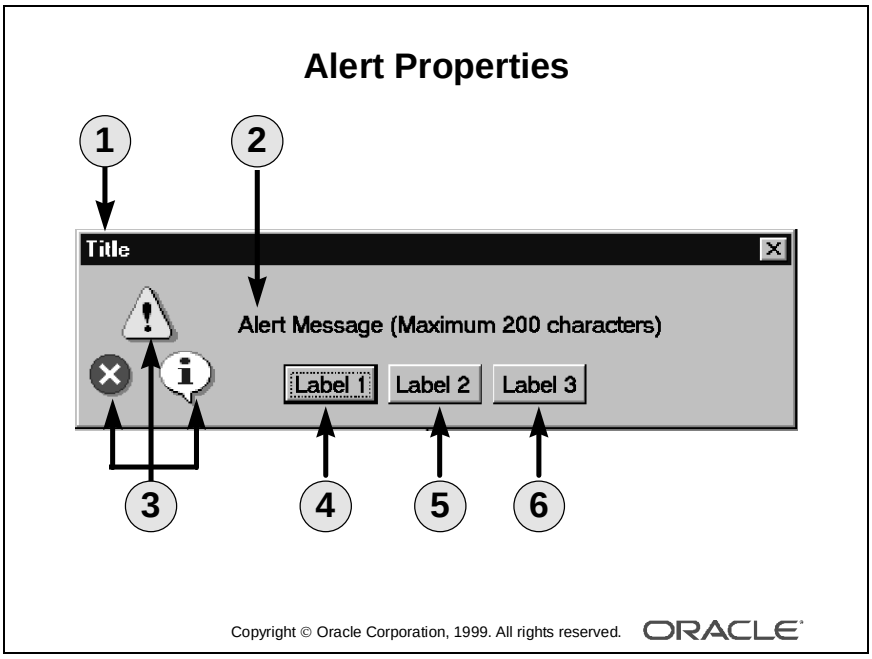
Built-in Function	Description of Returned Value
MESSAGE_CODE	Number of informative message that would have displayed (datatype NUMBER)
MESSAGE_TEXT	Text of informative message that would have displayed (datatype CHAR)
MESSAGE_TYPE	FRM=Form Builder message ORA= Oracle server message NULL=No message issued yet in this session (datatype CHAR)

Note: These functions return information about the most recent message that was issued. If your applications must be supported in more than one national language, then use MESSAGE_CODE in preference to MESSAGE_TEXT when checking a message.

Example of an On-Message Trigger

This On-Message trigger modifies the “Query caused no records to be retrieved” message (40350).

```
IF MESSAGE_CODE = 40350 THEN
   MESSAGE('No Orders found-check your search values');
ELSE
   MESSAGE(MESSAGE_TYPE || '-' || TO_CHAR(MESSAGE_CODE) ||
   ': '|| MESSAGE_TEXT);
END IF;
```



Alert Example

This is a generic example of an alert, showing all three icons and buttons that can be defined.

1	Title
2	Message
3	Alert style (stop, caution, note)
4	Button1 label
5	Button2 label
6	Button3 label

Creating and Controlling Alerts

Alerts are an alternative method for communicating with the operator. Because they display in a modal window, alerts provide an effective way of drawing attention and forcing the operator to answer the message before processing can continue.

Use alerts when you need to do the following:

- Display a message that the operator cannot ignore, and must acknowledge.
- Ask the operator a question where up to three answers are appropriate (typically Yes, No, or Cancel).

You handle the display and responses to an alert by using built-in subprograms. Alerts are therefore managed in two stages:

- Create the alert at design-time, and define its properties in the Property palette.
- Activate the alert at run time by using built-ins, and take action based on the operator's returned response.

How to Create an Alert

Like other objects you create at design-time, alerts are created from the Object Navigator.

1. Select the Alerts node in the Navigator, and then select Create.
2. Define the properties of the alert in the Property Palette.

Here are the properties that are specific to an alert. This is an abridged list.

Property	**Description**
Name	Name for this object
Title	Alert title
Alert Style	Defines the symbol that accompanies message: Stop, Caution, or Note
Button1, Button2, Button3	Labels for each of the three possible buttons (Null indicates that the button is not required.)
Default Alert Button	Button 1, Button 2, or Button 3
Message	Message that will appear in the alert (maximum 200 characters)

Planning Alerts

Yes/No questions

Yes/No/Cancel questions

Caution messages

Informative messages

Copyright © Oracle Corporation, 1999. All rights reserved.

Planning Alerts: How Many Do You Need?

Potentially, you can create an alert for every separate alert message that you need to display, but this is usually unnecessary.

You can define a message for an alert at run time, before it is displayed to the operator. This means that a single alert can be used for displaying many messages, providing that the available buttons are suitable for responding to each of these messages.

Create an alert for each combination of:

- Alert style required
- Set of available buttons (and labels) for operator response

For example, an application might require one Note-style alert with a single button (OK) for acknowledgment, one Caution alert with a similar button, and two Stop alerts that each provide a different combination of buttons for a reply. You can then assign a message to the appropriate alert before its display, through the SET_ ALERT_PROPERTY built-in procedure.

Controlling Alerts

SET_ALERT_PROPERTY

SET_ALERT_BUTTON_PROPERTY

Copyright © Oracle Corporation, 1999. All rights reserved.

Controlling Alerts at Run Time

There are built-in subprograms to change an alert message, to change alert button labels, and to display the alert, which returns the operator's response to the calling trigger.

SET_ALERT_PROPERTY Procedure

Use this built-in to change the message that is currently assigned to an alert. At form startup, the default message (as defined in the Property palette) is initially assigned:

```
SET_ALERT_PROPERTY('alert_name',property,'message')
```

Parameter	Description
Alert_name	The name of the alert, as defined in the Designer (You can alternatively specify an alert_id (unquoted) for this argument.)
Property	The property being set (Use ALERT_MESSAGE_TEXT when defining a new message for the alert.)
Message	The character string that defines the message (You can give a character expression instead of a simple quoted string, if required.)

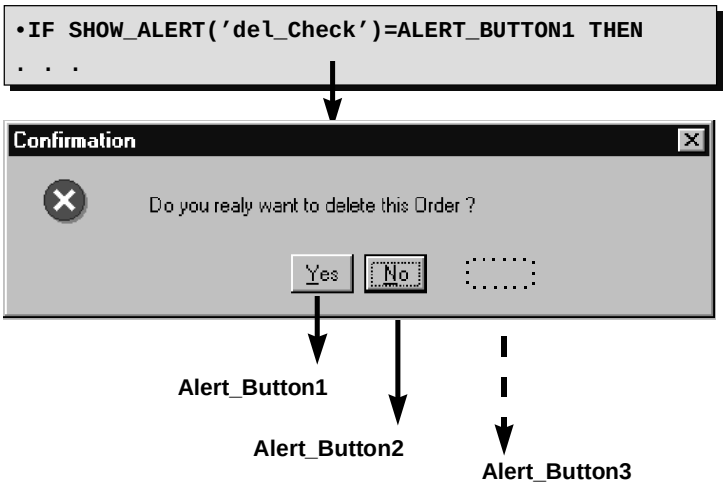
SET_ALERT_BUTTON_PROPERTY Procedure

Use this built-in to change the label on one of the alert buttons:

```
SET_ALERT_BUTTON_PROPERTY('alert_name', button, property,
'value')
```

Parameter	Description
Alert_name	The name of the alert, as defined in the Designer (You can alternatively specify an alert_id (unquoted) for this argument.)
Button	The number that specifies the alert button (Use ALERT_BUTTON1, ALERT_BUTTON2, ALERT_BUTTON3 constants.)
Property	The property being set; use LABEL
Value	The character string that defines the label

SHOW_ALERT Function



Copyright © Oracle Corporation, 1999. All rights reserved.

SHOW_ALERT Function

SHOW_ALERT is how you display an alert at run time, and return the operator's response to the calling trigger:

```
selected_button := SHOW_ALERT('alert_name');
. . .
```

Alert_Name is the name of the alert, as defined in the builder. You can alternatively specify an *Alert_Id* (unquoted) for this argument.

SHOW_ALERT returns a NUMBER constant, that indicates which of the three possible buttons the user pressed in response to the alert. These numbers correspond to the values of three PL/SQL constants, which are predefined by the Form Builder:

If the number equals...	**The Operator selected is...**
ALERT_BUTTON1	Button 1
ALERT_BUTTON2	Button 2
ALERT_BUTTON3	Button 3

After displaying an alert that has more than one button, you can determine which button the operator pressed by comparing the returned value against the corresponding constants.

Example

A trigger that fires when the user attempts to delete a record might invoke the alert, shown opposite, to obtain confirmation. If the operator selects Yes, then the DELETE_RECORD built-in is called to delete the current record from the block.

```
IF SHOW_ALERT('del_check') = ALERT_BUTTON1 THEN
   DELETE_RECORD;
END IF;
```

Directing Errors to an Alert

```
PROCEDURE Alert_On_Failure IS
  n NUMBER;
BEGIN
  SET_ALERT_PROPERTY('error_alert',
     ALERT_MESSAGE_TEXT,ERROR_TYPE||
     '-'||TO_CHAR(ERROR_CODE)||
     ': '||ERROR_TEXT);
  n := SHOW_ALERT('error_alert');
END;
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Directing Errors to an Alert

You may want to display errors automatically in an alert, through an On-Error trigger. The built-in functions that return error information, such as ERROR_TEXT, can be used in the SET_ALERT_PROPERTY procedure, to construct the alert message for display.

Example

The following user-named procedure can be called when the last form action was unsuccessful. The procedure fails the calling trigger and displays `Error_Alert` containing the error information.

```
PROCEDURE alert_on_failure IS
   n NUMBER;
BEGIN
   SET_ALERT_PROPERTY(
      'error_alert',
      ALERT_MESSAGE_TEXT,
      ERROR_TYPE||'-'||TO_CHAR(ERROR_CODE)||': ' ||
      ERROR_TEXT);
   n := SHOW_ALERT('error_alert');
END;
```

Summary

- **Application and system messages appear on message line.**
- **Test for built-in failure by using FORM_SUCCESS or other built-in functions.**
- **Set system variables to suppress system messages: MESSAGE_LEVEL and SUPPRESS_WORKING.**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

- **On-Error trigger intercepts system error messages.**
- **On-Message trigger intercepts system error messages.**
- **Alert types: Stop, Caution, and Note**
- **Up to three buttons are available for operator response.**
- **Display alerts with SHOW_ALERT.**
- **Change alert message with SET_ALERT_PROPERTY.**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

In this lesson, you saw how to intercept system messages, and how to replace them with ones that are more suitable for your application. You also learned how to build customized alerts for communicating with operators.

- The application and system messages appear on the Message line.
- Test for failure of built-ins by using the FORM_SUCCESS built-in function or other built-in functions.
- Set system variables to suppress system messages:
  - Assign a value to MESSAGE_LEVEL to specify that only messages above a specific severity level are to be used by the form.
  - Assign a value of True to SUPPRESS_WORKING to suppress all working messages.
- On-Error trigger intercepts system error messages.
- On-Message trigger intercepts informative system messages.
- Alert types: Stop, Caution, and Note.
- Up to three buttons are available for operator response (NULL indicates that the button is not required.).
- Display alerts at run time with SHOW_ALERT.
- Change alert messages with SET_ALERT_PROPERTY.

Practice 16 Overview

This practice covers the following topics:

- **Using an alert to inform the operator that the customer must pay cash**
- **Using a generic alert to ask the operator to confirm that the form should terminate**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 16 in Appendix A, “Practice Solutions.”

Practice 16 Overview

In this practice, you create some alerts in the ORDG*XX* form. These include a general alert for questions and a specific alert that is customized for payment type.

- Using an alert to inform the operator that the customer must pay cash
- Using a generic alert to ask the operator to confirm that the form should terminate

Practice 16

1. Create an alert in ORDG*XX* called Payment_Type_Alert with a single OK button. The message should read “This customer must pay cash!”

   Suggested Title: Payment Type. Style: Caution.

2. Alter the When-Radio-Changed trigger on Payment_Type to show the Payment_Type_Alert instead of the message when a customer must pay cash.

3. Create a generic alert called Question_Alert that allows Yes and No replies.

   Leave the Message property blank for this alert. Select the Stop style, and define two buttons in the alert: Yes and No.

4. Alter the When-Button-Pressed trigger on CONTROL.Exit_Button that uses Question_Alert to ask the operator to confirm that the form should terminate.

   Call the SET_ALERT_PROPERTY built-in to define the message:

   “Do you really want to leave the form?”

   Test the returned value of SHOW_ALERT, and call the EXIT_FORM built-in if the operator replied Yes.

5. Save, compile, and run the form to test.

17

Query Triggers

Objectives

After completing this lesson, you should be able to do the following:

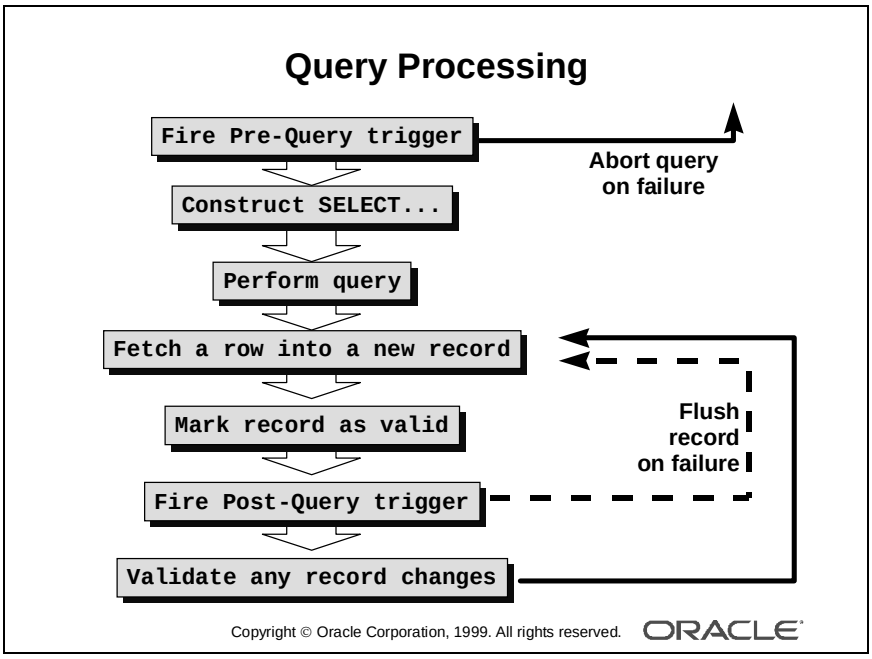
- **Explain the processes involved in querying a data block**
- **Describe query triggers and their scope**
- **Write triggers to supplement query results and screen query conditions**
- **Control trigger action based on the form query status**

Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

In this lesson, you learn how to control events associated with queries on base table data blocks. You can customize the query process as necessary, and supplement the results returned by a query.


Query Processing
Fire Pre-Query trigger
Abort query on failure
Construct SELECT...
Perform query
Fetch a row into a new record
Mark record as valid
Flush record on failure
Fire Post-Query trigger
Validate any record changes
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Query Triggers

Generally, triggers are associated with a query in one of two ways:

- A trigger fires due to the query process itself

  For example: Pre-Query and Post-Query
- An event can fire a trigger in Enter Query mode, if the Fire in Enter Query Mode property of the associated trigger is enabled

The query triggers, Pre-Query and Post-Query, fire due to the query process itself, and are usually defined on the block where the query takes place.

With these triggers you can add to the normal Form Builder processing of records, or possibly abandon a query before it is even executed, if the required conditions are not suitable.

Form Builder Query Processing

When a query is initiated on a data block, either by the operator or by a built-in subprogram, the following major events take place:

1. In Enter Query mode, Form Builder fires the Pre-Query trigger if defined.
2. If the Pre-Query succeeds, Form Builder constructs the query SELECT statement, based on any existing criteria in the block (either entered by the operator or by the Pre-Query).
3. The query is performed.
4. Form Builder fetches the column values of a row into the base table items of a new record in the block.
5. The record is marked Valid.
6. Form Builder fires the Post-Query trigger. If it fails, this record is flushed from the block.
7. Form Builder performs item and record validation if the record has changed (due to a trigger).
8. Step 4 is repeated for any remaining records of this query.

SELECT Statements Issued

```
SELECT   base_column, ..., ROWID
INTO     :base_item, ..., :ROWID
FROM     base_table
WHERE    default_where_clause
 AND     (example_record_conditions)
 AND     (query_where_conditions)
ORDER BY default_order_by_clause |
         query_where_order_by
```

Slightly different for COUNT

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

SELECT Statements Issued During Query Processing

If you have not altered default query processing, Form Builder issues a SELECT statement when you want to retrieve or count records.

```
SELECT base_column, base_column, ... , ROWID
INTO  :base_item,:base_item,  ... , :ROWID
FROM  base_table
WHERE default_where_clause
AND   (example_record_conditions)
AND   (query_where_conditions)
ORDER BY default_order_by_clause | query_where_order_by


SELECT COUNT(*)
FROM  base_table
WHERE default_where_clause
AND   (example_record_conditions)
AND   (query_where_conditions)
ORDER BY default_order_by_clause | query_where_order_by
```

Note: The vertical bar (|) in the ORDER BY clause indicates that either of the two possibilities can be present. Form Builder retrieves the ROWID only when the Key Mode block property is set to Unique (the default). The entire WHERE clause is optional. The ORDER BY clause is also optional.

If you want to count records that satisfy criteria specified in the Query/Where dialog box, enter one or more variables and press [Count Query] in the Example Record.

WHERE Clause

- Three sources for the WHERE clause:
  - WHERE clause block property
  - Example Record
  - Query/Where dialog box
- WHERE clauses are combined by the AND operator

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

ORDER BY Clause

- Two sources for the ORDER BY clause:
  - ORDER BY clause block property
  - Query/Where dialog box
- Second source for ORDER BY clause overrides the first one

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

WHERE and ORDER BY Clauses

The WHERE and ORDER BY clauses of a default base table SELECT statement are derived from several sources. It is important to know how different sources interact.

Three Sources for the WHERE Clause

- WHERE clause block property
- Example Record
- Query/Where dialog box

If more than one source is present, the different conditions will all be used and linked with an AND operator.

Two Sources for the ORDER BY Clause

- ORDER BY clause block property
- Query/Where dialog box

An ORDER BY clause specified in the Query/Where dialog box overrides the value of the ORDER BY clause block property.

Note: You can change the WHERE clause and ORDER BY clause block properties at run time by using the SET_BLOCK_PROPERTY built-in.

Pre-Query Trigger

- **Defined at block level**
- **Fires once, before query is performed**

```
IF   TO_CHAR(:S_ORD.ID)||
     TO_CHAR(:S_ORD.DATE_ORDERED)||
     TO_CHAR(:S_ORD.DATE_SHIPPED)
IS NULL THEN
     MESSAGE('You must query by
     Order ID or Date');
     RAISE form_trigger_failure;
END IF;
```

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Writing Query Triggers

Pre-Query Trigger

You must define this trigger at block level or above. It fires for either a global or restricted query, while the form is in Enter Query mode (that is, before Form Builder executes the query).

If the operator has initiated the query, the trigger fires after the query criteria is entered.

This means you can use Pre-Query as follows:

- To test the operator's query conditions, and to fail the query process if the conditions are not satisfactory for the application
- To add criteria for the query by assigning values to base table items

Example

This Pre-Query trigger on the S_ORD block only permits queries if there is a restriction on either the Order ID, Date Ordered, or Date Shipped. This prevents attempts at very large queries.

```
IF TO_CHAR(:S_ORD.id)||
   TO_CHAR(:S_ORD.date_ordered)||
   TO_CHAR(:S_ORD.date_shipped) IS NULL THEN
      MESSAGE('You must query by Order ID or Date');
      RAISE form_trigger_failure;
END IF;
```

Note: Pre-Query is useful for assigning values passed from other Oracle Developer modules, so that the query is related to data elsewhere in the session. We will look at doing this later.

Post-Query Trigger

- Fires for each fetched record (except during array processing)
- Use to populate nondatabase items and calculate statistics

```
SELECT   COUNT(ord_id)
INTO     :S_ORD.lineitem_count
FROM     S_ITEM
WHERE    ord_id = :S_ORD.id;
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Using SELECT Statements in Triggers

- Form Builder variables are preceded by a colon.
- The query must return one row for success.
- Code exception handlers.
- The INTO clause is mandatory, with a variable for each selected column or expression.
- ORDER BY is not relevant.

Copyright © Oracle Corporation, 1999. All rights reserved.

Post-Query Trigger

This trigger is defined at block level or above. Post-Query fires for each record that is fetched into the block as a result of a query. Note that the trigger only fires on the initial fetch of a record, not when a record is subsequently scrolled back into view a second or third time.

Use Post-Query as follows:

- To populate nondatabase items as records are returned from a query
- To calculate statistics

Example

This Post-Query trigger on the S_ORD block selects the total count of line items for the current Order, and displays this number as a summary value in the nonbase table item :Lineitem_count.

```
SELECT COUNT(ord_id)
INTO :S_ORD.lineitem_count
FROM S_ITEM
WHERE   ord_id = :S_ORD.id;
```

Using SELECT Statements in Triggers

The previous trigger example, populates the Lineitem_Count item through the INTO clause. Again, colons are required in front of Form Builder variables to distinguish them from PL/SQL variables and database columns.

Here is a reminder of some other rules regarding SELECT statements in PL/SQL:

- A single row must be returned from the query, or else an exception is raised that terminates the normal executable part of the block. You usually want to match a form value with a unique column value in your restriction.
- Code exception handlers in your PL/SQL block to deal with possible exceptions raised by SELECT statements.
- The INTO clause is mandatory, and must define a receiving variable for each selected column or expression. You can use PL/SQL variables, form items or global variables in the INTO clause.
- ORDER BY and other clauses that control multiple-row queries are not relevant (unless they are part of an Explicit Cursor definition).

Query Array Processing

- **Reduces network traffic**
- **Enables Query Array processing:**
  - **Enable Array Processing option**
  - **Set Query Array Size property**
- **Query Array Size property**
- **Query All Records property**

Copyright © Oracle Corporation, 1999. All rights reserved.

Query Array Processing

The default behavior of Form Builder is to process records one at a time. With array processing, a structure (array) containing multiple records is sent to or returned from the server for processing.

Form Builder supports both array fetch processing and array DML processing. For both querying and DML operations, you can determine the array size to optimize performance for your needs. This lesson focuses on array query processing.

Enabling Array Processing for Queries

1. Setting preferences:
   - Select Tools—>Preferences.
   - Click the Runtime tab.
   - Select the Array Processing check box.
2. Setting properties:
   - In the Object Navigator, select the Data Blocks node.
   - Double-click the Data Blocks icon to display the Property Palette.
   - Under the Records category, set the Query Array Size property to a number that represents the number of records in the array for array processing.

Query Array Size Property This property specifies the maximum number of records that Form Builder should fetch from the database at one time.

A size of 1 provides the fastest perceived response time, because Form Builder fetches and displays only one record at a time. By contrast, a size of 10 fetches up to ten records before displaying any of them, however, the larger size reduces overall processing time by making fewer calls to the database for records.

Query All Records Property Specifies whether all the records matching the query criteria should be fetched into the data block when a query is executed.

- Yes: Fetches all records from query.
- No: Fetches the number of records specified by the Query Array Size block property.

Coding for ENTER-QUERY Mode

- **Some triggers may fire in Enter-Query mode.**
- **Set to fire in Enter-Query Mode property.**
- **Test mode during execution with :SYSTEM.MODE**
  - **NORMAL**
  - **ENTER-QUERY**
  - **QUERY**

Copyright © Oracle Corporation, 1999. All rights reserved.

Coding Triggers for Enter Query Mode

Some triggers that fire when the form is in Normal mode (during data entry and saving) may also be fired in Enter Query mode. You need to consider the trigger type and actions in these cases.

Fire in Enter Query Mode Property

This property determines whether Form Builder fires a trigger if the associated event occurs in Enter Query mode. Not all triggers can do this; consult the Form Builder online Help, which lists each trigger and whether this property can be set.

By default, the Fire in Enter Query Mode property is set to Yes for triggers that accept this. Set it to No in the Property Palette if you only want the trigger to fire in Normal mode.

Example

If you provide a button for the operator to invoke an LOV, and the LOV is required to help with query criteria as well as data entry, then the When-Button-Pressed trigger needs to fire in both modes. This trigger has Fire in Enter Query Mode set to Yes (default for this trigger type):

```
IF SHOW_LOV('Customers') THEN
   MESSAGE('Selection successful');
END IF;
```

Coding for ENTER-QUERY Mode

- **Example**

```
IF    :SYSTEM.MODE = 'NORMAL'
THEN ENTER_QUERY;
ELSE EXECUTE_QUERY;
END IF;
```

- **Some built-ins are illegal.**
- **Consult online Help.**
- **You cannot navigate to another record in the current form.**

Copyright © Oracle Corporation, 1999. All rights reserved.

Finding Out the Current Mode

When a trigger will fire in both Enter Query mode and Normal modes, you may need to know the current mode at execution time for the following reasons:

- Your trigger needs to perform different actions depending on the mode.
- Some built-in subprograms cannot be used in Enter Query mode.

The read-only system variable, MODE, stores the current mode of the form. Its value (always upper case) is one of the following:

Value of SYSTEM.MODE	Definition
NORMAL	Form is in Normal processing mode.
ENTER-QUERY	Form is in Enter Query mode.
QUERY	Form is in Fetch-processing mode, meaning that Form Builder is currently doing a fetch. (For example, this value always occurs in a Post-Query trigger.)

Example

Consider the following When-Button-Pressed trigger for the Query button.

If the operator clicks the button in Normal mode, then the trigger places the form in Enter Query mode (using the ENTER_QUERY built-in). Otherwise, if already in Enter Query mode, the button executes the query (using the EXECUTE_QUERY built-in).

```
IF :SYSTEM.MODE = 'NORMAL' THEN
   ENTER_QUERY;
ELSE
   EXECUTE_QUERY;
END IF;
```

Using Built-ins in Enter Query Mode

Some built-in subprograms are illegal if a trigger is executed in Enter Query mode. Again, consult the Form Builder online Help which specifies whether an individual built-in can be used in this mode.

One general restriction is that in Enter Query mode you can not navigate to another record in the current form. So any built-in that would potentially enable this is illegal. These include GO_BLOCK, NEXT_BLOCK, PREVIOUS_BLOCK, GO_RECORD, NEXT_RECORD, PREVIOUS_RECORD, UP, DOWN, OPEN_FORM, and others.

Overriding Default Query Processing

Trigger	Do-the-Right-Thing Built-in
On-Close	
On-Count	COUNT_QUERY
On-Fetch	FETCH_RECORDS
Pre-Select	
On-Select	SELECT_RECORDS
Post-Select	

Copyright © Oracle Corporation, 1999. All rights reserved.

Overriding Default Query Processing

You can use certain transactional triggers to replace default commit processing. Some of the transactional triggers can also be used to replace default query processing. You can use “Do-the-right-thing” built-ins to augment default query processing; do not use “Do-the-right-thing” to replace default processing.

Additional Transactional Triggers for Query Processing

Trigger	**Do-the-Right-Thing Built-in**
On-Close	
On-Count	COUNT_QUERY
On-Fetch	FETCH_RECORDS
Pre-Select	
On-Select	SELECT_RECORDS
Post-Select	

Overriding Default Query Processing

- **On-Fetch continues to fire until:**
  - **It fires without executing CREATE_QUERIED_RECORD.**
  - **The query is closed by the user or by ABORT_QUERY.**
  - **It raises FORM_TRIGGER_FAILURE.**
- **On-Select replaces open cursor, parse, and execute phases.**

Copyright © Oracle Corporation, 1999. All rights reserved.

Characteristics of Transactional Triggers for Query Processing

Trigger	Characteristic
On-Close	Fires when Form Builder closes a query (It augments, rather than replaces, default processing.)
On-Count	Fires when Form Builder would usually perform default Count Query processing to determine the number of rows that match the query conditions
On-Fetch	Fires when Form Builder performs a fetch for a set of rows (You can use the CREATE_QUERIED_RECORD built-in to create queried records if you want to replace default fetch processing.) The trigger continues to fire until: • No queried records are created during a single execution of the trigger • The query is closed by the user or by the ABORT_QUERY built-in is executed from another trigger • The trigger raises FORM_TRIGGER_FAILURE
Pre-Select	Fires after Form Builder has constructed the block SELECT statement based on the query conditions, but before it issues this statement
On-Select	Fires when Form Builder would usually issue the block SELECT statement (The trigger replaces the open cursor, parse, and execute phases of a query.)
Post-Select	Fires after Form Builder has constructed and issued the block SELECT statement, but before it fetches the records

Uses for Transactional Triggers for Query Processing

Transactional triggers for query processing are primarily intended to access certain data sources other than Oracle. However, you can also use these triggers to implement special functionality by augmenting default query processing against an Oracle database.

Obtaining Query Information at Run Time

- **SYSTEM.MODE**
- **SYSTEM.LAST_QUERY**
  - **Contains bind variables (ORD_ID = :1) before SELECT_RECORDS**
  - **Contains actual values (ORD_ID = 102) after SELECT_RECORDS**

Copyright © Oracle Corporation, 1999. All rights reserved.

Obtaining Query Information at Run Time

If you want to exercise more control over your queries, use system variables and built-ins to obtain information about queries.

Using SYSTEM.MODE

Use the SYSTEM.MODE system variable to obtain the form mode. The three values are NORMAL, ENTER_QUERY, and QUERY. We discussed this system variable in the section “Finding Out the Current Mode” in this lesson.

Using SYSTEM.LAST_QUERY

Use SYSTEM.LAST_QUERY to obtain the text of the base-table SELECT statement that was last executed by Form Builder. If a user has entered query conditions in the Example Record, the exact form of the SELECT statement depends on when this system variable is used.

If the system variable is used before Form Builder has implicitly executed the SELECT_RECORDS built-in, the SELECT statement contains bind variables (for example, ORD_ID = :1).

If the system variable is used after Form Builder has implicitly executed the SELECT_RECORDS built-in, the SELECT statement contains the actual search values (for example, ORD_ID = 102). For example, the system variable contains bind variables during the Pre-Select trigger and actual search values during the Post-Select trigger.

Obtaining Query Information at Run Time

- GET_BLOCK_PROPERTY
  SET_BLOCK_PROPERTY
  - Get and set:
    DEFAULT_WHERE
    ORDER_BY
    QUERY_ALLOWED
    QUERY_HITS
  - Get only:
    QUERY_OPTIONS
    RECORDS_TO_FETCH

Copyright © Oracle Corporation, 1999. All rights reserved.

Obtaining Query Information at Run Time

- GET_ITEM_PROPERTY
  SET_ITEM_PROPERTY
  - Get and set:
    CASE_INSENSITIVE_QUERY
    QUERYABLE
    QUERY_ONLY
  - Get only:
    QUERY_LENGTH

Copyright © Oracle Corporation, 1999. All rights reserved.

Using GET_BLOCK_PROPERTY and SET_BLOCK_PROPERTY

The following block properties may be useful for obtaining query information. Only the properties marked with an asterisk can be set.

- DEFAULT_WHERE (*)
- ORDER_BY (*)
- QUERY_ALLOWED (*)
- QUERY_HITS (*)
- QUERY_OPTIONS
- RECORDS_TO_FETCH

Using GET_ITEM_PROPERTY and SET_ITEM_PROPERTY

The following item properties may be useful for getting query information. Only the properties marked with an asterisk can be set.

- CASE_INSENSITIVE_QUERY (*)
- QUERYABLE (*)
- QUERY_ONLY (*)
- QUERY_LENGTH

Summary

- **A Pre-Query trigger fires before a query executes. Use it to check or modify query conditions.**
- **A Post-Query trigger fires as each record is fetched (except array processing). Use it to perform calculations and populate additional items.**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

- **Some triggers can fire in both Normal and Enter Query modes:**
  - **Test the current mode with SYSTEM.MODE.**
  - **Some built-ins are illegal in Enter Query mode.**
- **Obtain query information at run time:**
  - **SYSTEM.MODE**
  - **SYSTEM.LAST_QUERY**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

In this lesson, you learned how to control the events associated with queries on base table blocks.

- The Pre-Query trigger fires before the query executes. This trigger is defined at the block level or above. Use the Pre-Query trigger to check or modify query conditions.
- The Post-Query trigger fires as each record is fetched (except array processing). This trigger is defined at the block level or above. Use the Post-Query trigger to perform calculations and populate additional items.
- Some triggers can fire in both Normal and Enter Query modes.
  - Use SYSTEM.MODE to test the current mode.
  - Some built-ins are illegal in Enter Query mode.
- Override default query processing by using “Do-the-right-thing” built-ins.
- Obtain query information at runtime by using:
  - SYSTEM.MODE, SYSTEM.LAST_QUERY
  - Some properties of GET/SET_BLOCK_PROPERTY and GET/SET_ITEM_PROPERTY

Practice 17 Overview

This practice covers the following topics:

- **Populating customer names and sales representative names for each row of the S_ORD block**
- **Populating descriptions for each row of the S_ITEM block**
- **Disabling the effect of the Exit button in Enter Query mode**
- **Adding two check boxes to enable case-sensitive and exact match query**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 17 in Appendix A, “Practice Solutions.”

Practice 17 Overview

In this practice, you create two query triggers to populate nonbase table items. You will also change the default query interface to enable case sensitive and exact match query.

- Populating customer names and sales representative names for each row of the S_ORD block
- Populating descriptions for each row of the S_ITEM block
- Disabling the effect of the Exit button in Enter Query mode
- Adding two check boxes to enable case sensitive and exact match query

Practice 17

1. In the ORDG*XX* form, write a trigger that populates the Customer_Name and the Sales_Rep_Name for every row fetched by a query on the S_ORD block.
2. Write a trigger that populates the Description for every row fetched by a query on the S_ITEM block.
3. Ensure that the Exit_Button has no effect in Enter Query mode.
4. Adjust the default query interface. Open the CUSTOMERS form module. Add a check box called CONTROL.Case_Sensitive to the form so that the user can specify whether or not a query for a customer name should be case sensitive. You can import the `pr17_4.txt` file into the When-Checkbox-Changed trigger. Set the initial value property to "Y."

   In the CONTROL block, add a check box (called CONTROL.Case_Sensitive as shown below) to it, and create the following trigger. Set the "Mouse Navigate" property to No.

When-Checkbox-Changed Trigger on the CONTROL.Case_Sensitive Item (Checkbox)

```
IF NVL(:CONTROL.case_sensitive, 'Y') = 'Y' THEN
   SET_ITEM_PROPERTY('S_CUSTOMER.name', CASE_INSENSITIVE_QUERY,
                     PROPERTY_FALSE);
ELSE
   SET_ITEM_PROPERTY('S_CUSTOMER.name',CASE_INSENSITIVE_QUERY,
                        PROPERTY_TRUE);
END IF;
```

5. Add a check box called CONTROL.Exact_Match to the form so that the user can specify whether or not a query condition for a customer name should exactly match the table value. (If a nonexact match is allowed, the search value can be part of the table value.) You can import the `pr17_5.txt` file into the Pre-Query Trigger. Set the initial value property to "Y." Add another check box (called CONTROL.Exact_Match as shown below) to the CONTROL block and create the following trigger. Set the Mouse Navigate property to No.

Pre-Query Trigger on the S_CUSTOMER Block

```
IF NVL( :CONTROL.exact_match, 'Y' ) = 'N' THEN
   :S_CUSTOMER.name := '%' || :S_CUSTOMER.name || '%';
END IF;
```

18

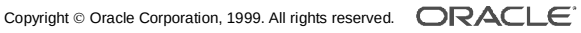
Validation

Objectives

After completing this lesson, you should be able to do the following:

- **Explain the effects of the validation unit upon a form**
- **List Form Builder validation properties**
- **Control validation by using triggers**

Copyright © Oracle Corporation, 1999. All rights reserved.



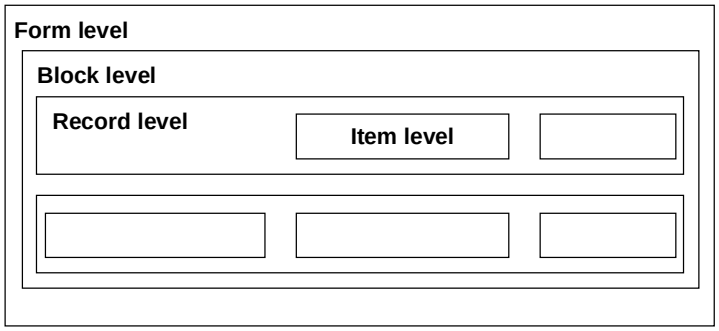
Introduction

Overview

In this lesson, you will learn how to supplement item validation by using both object properties and triggers. You will also learn to control when validation occurs.

Validation

- **Form Builder validates at the following levels:**



Copyright © Oracle Corporation, 1999. All rights reserved.

Validation

- **Validation occurs when:**
  - **[Enter] key or ENTER Built-in is obeyed**
  - **Operator or trigger leaves the validation unit (includes a Commit)**

Copyright © Oracle Corporation, 1999. All rights reserved.

Validation Process

Form Builder performs a validation process at several levels to ensure that records and individual values follow appropriate rules. If validation fails, then control is passed back to the appropriate level, so that the operator can make corrections. Validation occurs at:

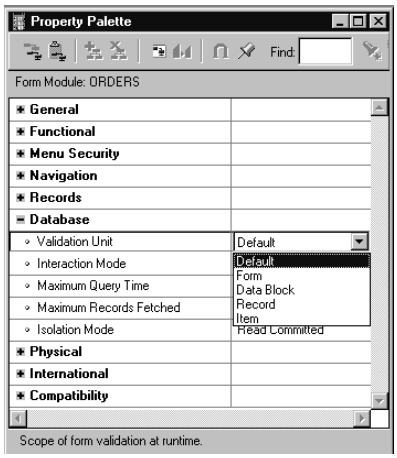
- Item level: Form Builder records a status for each item to determine whether it is currently valid. If an item has been changed and is not yet marked as valid, then Form Builder first performs standard validation checks to ensure that the value conforms to the item's properties. These checks are carried out before firing any When-Validate-Item triggers that you have defined. Standard checks include the following:
  - Format mask
  - Required (if so, then is the item null?)
  - Data type
  - Range (Lowest-Highest Allowed Value)
  - Validate from List (see later in this lesson)
- Record level: After leaving a record, Form Builder checks to see whether the record is valid. If not, then the status of each item in the record is checked, and a When-Validate-Record trigger is then fired, if present. When the record passes these checks, it is set to valid.
- Block and form level: At block or form level, all records below that level are validated. For example, if you commit (save) changes in the form, then all records in the form are validated, unless you have suppressed this action.

When Does Validation Occur?

Form Builder carries out validation for the validation unit under the following conditions:

- The [Enter] key is (ENTER command is not necessary mapped to the key that is physically labeled Enter) pressed or the ENTER built-in procedure is run (whose purpose is to force validation immediately).
- The operator or a trigger navigates out of the validation unit. This includes when changes are committed. The default validation unit is item, but can also be set to record, block, or form by the designer. The validation unit is discussed in the next section.

Validation Unit Property



Copyright © Oracle Corporation, 1999. All rights reserved.

Using Object Properties to Control Validation

You can control when and how validation occurs in a form, even without triggers. Do this by setting properties for the form and for individual items within it.

The Validation Unit

The validation unit defines the maximum amount of data an operator can enter in the form before Form Builder initiates validation. Validation unit is a property of the form module, and it can be set in the Property Palette to any of the following:

- Default
- Item
- Record
- Block
- Form

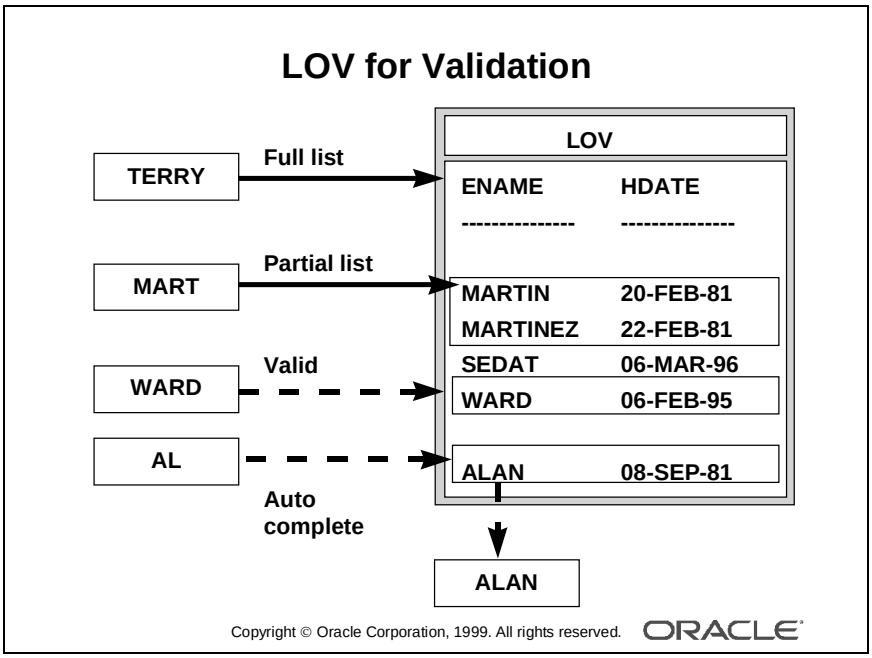
The default setting is item level. The default setting is usually chosen.

In practice, an item-level validation unit means that Form Builder validates changes when an operator navigates out of a changed item. This way, standard validation checks and firing the When-Validate-Item trigger of that item can be done immediately. As a result, operators are aware of validation failure as soon as they attempt to leave the item.

At higher validation units (record, block, or form level), the above checks are postponed until navigation moves out of that unit. All outstanding items and records are validated together, including the firing of When-Validate-Item and When-Validate-Record triggers.

You might set a validation unit above item level under one of the following conditions:

- Validation involves database references, and you want to postpone traffic until the operator has completed a record (record level).
- The application runs in a block-mode environment (block level).


LOV for Validation
TERRY
Full list
MART
Partial list
WARD
Valid
AL
Auto complete
LOV
ENAME
HDATE
MARTIN
20-FEB-81
MARTINEZ
22-FEB-81
SEDAT
06-MAR-96
WARD
06-FEB-95
ALAN
08-SEP-81
ALAN
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Using LOVs for Validation

When you attach an LOV to a text item by setting the LOV property of the item, you can optionally use the LOV contents to validate data entered in the item.

Do this by setting the Validate from List property to Yes for the item. At validation time, Form Builder then automatically uses the item value as a non case-sensitive search string on the LOV contents. The following events then occur, depending on the circumstances:

- If the value in the text item matches one of the values in the first column of the LOV, validation succeeds, the LOV is not displayed, and processing continues normally.
- If the item's value causes a single record to be found in the LOV, but is a partial value of the LOV value, then the full LOV column value is returned to the item (providing that the item is defined as the return item in the LOV). The item then passes this validation phase.
- If the item value causes multiple records to be found in the LOV, Form Builder displays the LOV and uses the text item value as the search criteria to automatically reduce the list, so that the operator must choose.
- If no match is found, then the full LOV contents are displayed to the operator.

Note: Make sure that LOVs you create for validation purposes have the validation column defined first, with a display width greater than 0. You also need to define the Return Item for the LOV column as the item being validated.

For performance reasons, do not use the LOV for Validation property for large LOVs.

Validation Triggers

- **Item level**
  When-Validate-Item
- **Block level**
  When-Validate-Record

```
IF :S_ORD.date_shipped < :S_ORD.date_ordered THEN
  MESSAGE('Ship Date is before Order Date!');
  RAISE form_trigger_failure;
END IF;
```

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Controlling Validation by Using Triggers

There are triggers that fire due to validation, which let you add your own customized actions. There are also some built-in subprograms that you can call from triggers that affect validation.

When-Validate-Item Trigger

You have already used this trigger to add item-level validation. The trigger fires after standard item validation, and input focus is returned to the item if the trigger fails.

Example

This When-Validate-Item trigger on :S_ORD.date_ordered ensures that the Order Date is not later than the current (database) date:

```
IF :S_ORD.date_ordered > SYSDATE THEN
   MESSAGE('Order Date is later than today!');
   RAISE form_trigger_failure;
END IF;
```

When-Validate-Record Trigger

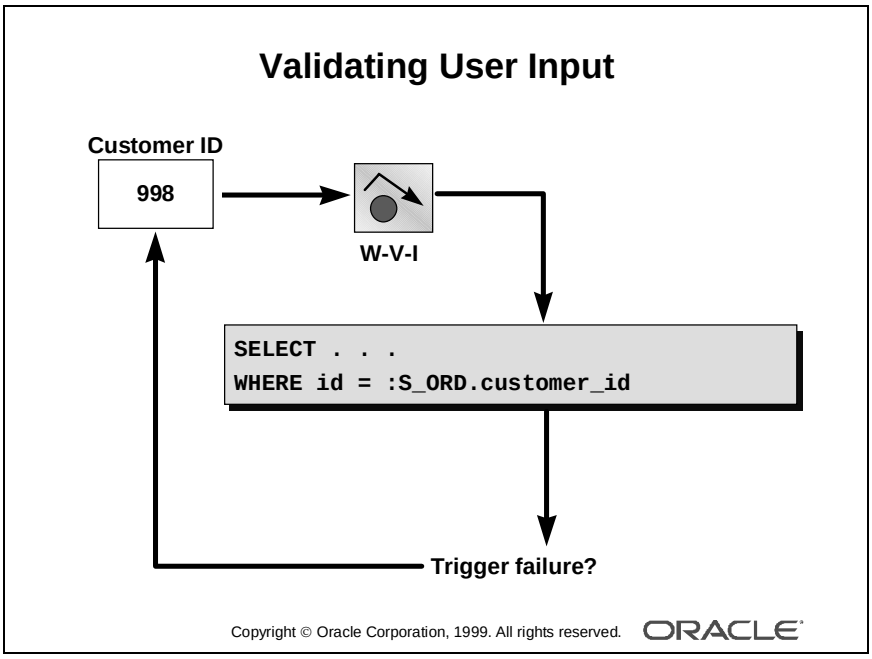
This trigger fires after standard record-level validation, when the operator has left a new or changed record. Because Form Builder has already checked that required items for the record are valid, you can use this trigger to perform additional checks that may involve more than one of the record's items, in the order they were entered.

When-Validate-Record must be defined at block level or above.

Example

This When-Validate-Record trigger on block S_ORD ensures that orders cannot be shipped before they are ordered.

```
IF :S_ORD.date_shipped < :S_ORD.date_ordered THEN
   MESSAGE('Ship Date is before Order Date!');
   RAISE  form_trigger_failure;
END IF;
```


Validating User Input
Customer ID
998
W-V-I
SELECT . . .
WHERE id = :S_ORD.customer_id
Trigger failure?
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Validating User Input

While populating other items, if the user enters an invalid value in the item, a matching row will not be found, and the SELECT statement will cause an exception. The success or failure of the query can, therefore, be used to validate user input.

The exceptions that can occur when a single row is not returned from a SELECT in a trigger are:

- NO_DATA_FOUND
  No rows are returned from the query.
- TOO_MANY_ROWS
  More than one row is returned from the query.

Example

The following When-Validate-Item trigger is again placed on the Customer_ID item, and returns both the Name and Phone Number that correspond to the Customer ID entered by the user.

```
SELECT name, phone
INTO :S_ORD.customer_name, :S_ORD.customer_phone
FROM s_customer
WHERE id = :S_ORD.customer_id;
```

If the Customer_ID item contains a value that is not found in the table, the NO_DATA_FOUND exception is raised, and the trigger will fail because there is no exception handler to prevent the exception from propagating to the end of the trigger.

Note: A failing When-Validate-Item trigger prevents the cursor from leaving the item.

For an unhandled exception, as above, the user receives the message:

```
FRM-40735: <trigger type> trigger raised unhandled exception
<exception>
```

Tracking Validation Status

- **NEW**
  - **When a record is created**
  - **Also for Copy Value from Item or Initial Value**
- **CHANGED**
  - **When changed by user or trigger**
  - **When any item in new record is changed**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Tracking Validation Status

- **VALID**
  - **When validation has been successful**
  - **After records are fetched from database**
  - **After a successful post or commit**
  - **Duplicated record inherits status of source**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Tracking Validation Status

When Form Builder leaves an object, it usually validates any changes that were made to the contents of the object. To determine whether validation must be performed, Form Builder tracks the validation status of items and records.

Item Validation Status

Status	Definition
NEW	When a record is created, Form Builder marks every item in that record as new. This is true even if the item is populated by the Copy Value from Item or Initial Value item properties, or by the When-Create-Record trigger.
CHANGED	Form Builder marks an item as changed under the following conditions: • When the item is changed by the user or a trigger • When any item in a new record is changed, all of the items in the record are marked as changed
VALID	Form Builder marks an item as valid under the following conditions: • All items in record that are fetched from the database are marked as valid • If validation of the item has been successful • After successful post or commit • Each item in a duplicated record inherits the status of its source

Record Validation Status

Status	Definition
NEW	When a record is created, Form Builder marks that record as new. This is true even if the item is populated by the Copy Value from Item or Initial Value item properties, or by the When-Create-Record trigger.
CHANGED	Whenever an item in a record is marked as changed, Form Builder marks that record as changed.
VALID	Form Builder marks a record as valid under the following conditions: • After all items in the record have been successfully validated • All records that are fetched from the database are marked as valid • After successful post or commit • A duplicated record inherits the status of its source

Built-ins for Validation

- CLEAR_BLOCK, CLEAR_ FORM, EXIT_FORM
- ENTER
- SET_FORM_PROPERTY
  - (..., VALIDATION)
  - (..., VALIDATION_UNIT)
- ITEM_IS_VALID item property
- VALIDATE (VALIDATION_ UNIT)

Copyright © Oracle Corporation, 1999. All rights reserved.

Built-ins for Validation

You can use the following built-in subprograms in triggers to affect validation.

CLEAR_BLOCK, CLEAR_FORM, and EXIT_FORM

The first parameter to these built-ins, COMMIT_MODE, controls what will be done with unapplied changes when a block is cleared, the form is cleared, or the form is exited respectively. When the parameter is set to NO_VALIDATE, changes are neither validated nor committed (by default, the operator is prompted for the action).

ITEM_IS_VALID Item Property

You can use GET_ITEM_PROPERTY and SET_ITEM_PROPERTY built-ins with the ITEM_IS_VALID parameter to get or set the validation status of an item. You cannot directly get and set the validation status of a record. However, you can get or set the validation status of all the items in a record.

ENTER

The ENTER built-in performs the same action as the [Enter] key. That is, it forces validation of data in the current validation unit.

SET_FORM_PROPERTY

You can use this to disable Form Builder validation. For example, suppose you are testing a form, and you need to bypass normal validation. Set the Validation property to Property_False for this purpose:

```
SET_FORM_PROPERTY('form_name',VALIDATION, PROPERTY_FALSE);
```

You can also use this built-in to change the validation unit programmatically:

```
SET_FORM_PROPERTY('form_name',VALIDATION_UNIT, scope);
```

VALIDATE

VALIDATE forces Form Builder to immediately execute validation processing for the indicated scope.

Note: Scope is one of DEFAULT_SCOPE, BLOCK_SCOPE, RECORD_SCOPE, or ITEM_SCOPE.

Summary

- Validation occurs at item, record, block, and form levels.
- Validation happens when:
  - [Enter] Key or ENTER built-in is activated
  - Control leaves the validation unit due to navigation or commit

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

- Standard validation occurs before trigger validation.
- Default validation unit is item level.
- Validation status
  - NEW
  - CHANGED
  - VALID
- When-Validate-“object” triggers to supplement validation.

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Summary

In this lesson, you learned to use additional validation features in Form Builder, and to control when validation occurs.

- Validation occurs at three levels:
  - Item level: To ensure that the value conforms to the item's properties
  - Record level: To ensure that the record is valid (If it is not, then the status of each item in the record is checked.)
  - Block and form level: To ensure that the all records below the level are validated.
- Validation happens when:
  - The [Enter] Key or the ENTER built-in procedure is run (to force validation immediately.)
  - Control leaves the validation unit due to navigation or Commit.
- Standard validation occurs before trigger validation.
- The Default validation unit is item level.
- Validation Status:
  - NEW
  - CHANGED
  - VALID
- The When-Validate-"*object*" triggers supplement standard validation.

Practice 18 Overview

This practice covers the following topics:

- **Validating the Sales Representative item value using a LOV**
- **Writing a validation trigger to check that the shipped date is not before the ordered date**
- **Populating customer names, sales representative names, and IDs when a customer ID is changed**
- **Writing a validation trigger to populate the name and the price of the product when the product ID is changed**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Note

For solutions to this practice, see Practice 18 in Appendix A, “Practice Solutions.”

Practice 18 Overview

In this practice, you introduce additional validation to the CUSTG*XX* and ORDG*XX* form modules.

- Validating sales representative item value using an LOV
- Writing a validation trigger to check that the shipped date is not before the ordered date
- Populating customer names, sales representative names, and IDs when a customer ID is changed
- Writing a validation trigger to populate the name and the price of the product when the product ID is changed

Practice 18

1. In the CUSTG*XX* form, cause the Sales_Rep_Lov to display whenever the user enters a Sales_Rep_Id that does not exist in the database.
2. Save, compile, and run the form to test.
3. In the ORDG*XX* form, write a validation trigger to check that the Date_Shipped is not before the Date_Ordered.

   Write a When-Validate-record trigger to compare the values of the Date_Shipped and Date_Ordered. If the Date_Shipped is before the Date_Ordered, fail the trigger with a suitable message.
4. In the ORDG*XX* form, create a trigger to write the correct values to the Customer_Name, Sales_Rep_Name, and Sales_Rep_Id items whenever validation occurs on Customer_Id.

   Fail the trigger if the customer is not found.
5. Create another validation trigger on S_ITEM.Product_Id to derive the name of the product, and write it to the Description item.

   Fail the trigger and display a message if the product is not found.

19

Navigation

Objectives

After completing this lesson, you should be able to do the following:

- **Distinguish between internal and external navigation**
- **Describe and use navigation triggers**
- **Identify built-ins that cause navigation**

Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

The Oracle Developer Form Builder component offers a variety of ways to control cursor movement. This lesson looks at the different methods of forcing navigation both visibly and invisibly.

About Navigation

- **What is the navigation unit?**
  - **Outside the form**
  - **Form**
  - **Block**
  - **Record**
  - **Item**
- **Entering and leaving objects**
- **What happens if navigation fails?**

Copyright © Oracle Corporation, 1999. All rights reserved.

About Navigation

The following sections introduce a number of navigational concepts to help you to understand the navigation process.

What Is the Navigational Unit?

The navigational unit is an invisible, internal object that determines the navigational state of a form. Form Builder uses the navigation unit to keep track of the object that is currently the focus of a navigational process. The navigation unit can be one of the objects in the following hierarchy:

- Outside the form
- Form
- Block
- Record
- Item

When Form Builder navigates, it changes the navigation unit moving through this object hierarchy until the target item is reached.

Entering and Leaving Objects

During navigation, Form Builder leaves and enters objects. Entering an object means changing the navigation unit from the object above in the hierarchy. Leaving an object means changing the navigation unit to the object above.

The Cursor and How it Relates to the Navigation Unit

The cursor is a visible, external object that indicates the current input focus. Form Builder will not move the cursor until the navigation unit has successfully become the target item. In this sense, the navigation unit acts as a probe.

What Happens if Navigation Fails?

If navigation fails, Form Builder reverses the navigation path and attempts to move the navigation unit back to its initial location. Note that the cursor is still at its initial position. If Form Builder cannot move the navigation unit back to its initial location, it exits the form.

Navigation Properties

- **Form module**
  - **Mouse navigation limit**
  - **First navigation data block**
- **Block**
  - **Navigation style**
  - **Previous navigation data block**
  - **Next navigation data block**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Navigation Properties

- **Item**
  - **Enabled**
  - **Keyboard navigable**
  - **Mouse navigate**
  - **Previous navigation item**
  - **Next navigation item**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

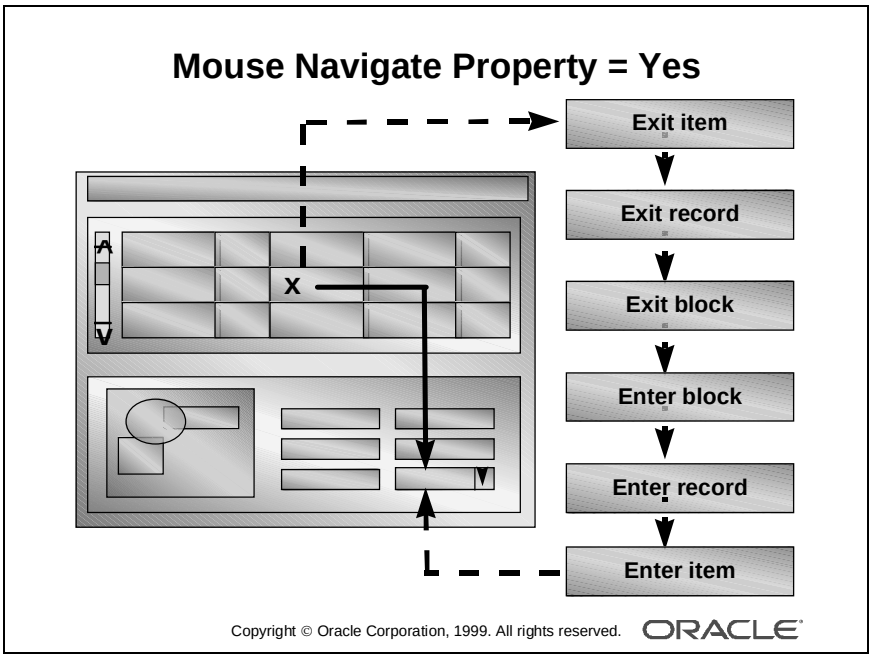
Controlling Navigation

You can control the path through an application by controlling the order in which the user navigates to objects. You have seen navigation properties for blocks and items. There are two other navigation properties that you can set for the form module: Mouse Navigation Limit and First Navigation Block.

Form Module Properties	**Function**
Mouse Navigation Limit	Determines how far outside the current item the user can navigate with the mouse
First Navigation Block	Specifies the name of the block to which Form Builder should navigate on form startup (Setting this property does not override the order used for committing.)

Object	**Property**
Block	Navigation Style
	Previous Navigation Block
	Next Navigation Block
Item	Enabled
	Keyboard Navigable
	Mouse Navigate
	Previous Navigation Item
	Next Navigation Item

Note: In a bitmapped environment, you can use the mouse to navigate to any enabled item regardless of its position in the navigational order.


Mouse Navigate Property = Yes
Exit item
Exit record
Exit block
Enter block
Enter record
Enter item
X
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Mouse Navigate Property

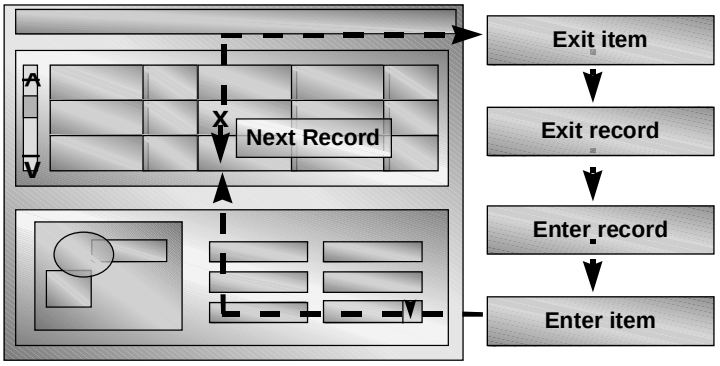
The Mouse Navigate property applies only to mouse-driven applications, and is valid for the following items:

- Push Button
- Check box
- List item
- Radio group
- Hierarchical tree item
- Sound item
- Custom item
  - ActiveX Control
  - VBX Control
  - OLE2 Container
  - Bean Area

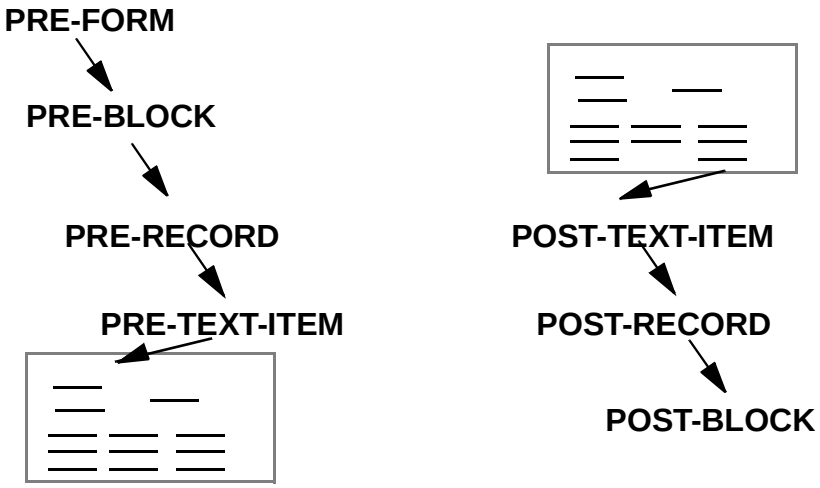
Setting	Use to Ensure That
Yes	Form Builder navigates to the new item which causes the relevant navigational and validation triggers to fire
No	Form Builder does not navigate to the new item or validate the existing item when the user activates the new item with the mouse

Note: The default setting for the Mouse Navigate property is Yes.

Internal Navigation



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE



Understanding Internal Navigation

Navigation occurs when the user or a trigger causes the input focus to move to another object. You have seen that navigation involves changing the location of the input focus on the screen. In addition to the visible navigation that occurs, some logical navigation takes place. This logical navigation is also known as internal navigation.

Example

When you enter a form module, you see the input focus in the first enterable item of the first navigation block. You do not see the internal navigation events that must occur for the input focus to enter the first item. These internal navigation events are as follows:

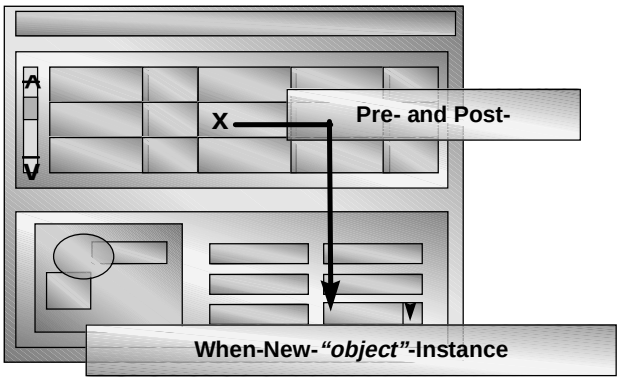
- Entry to form
- Entry to block
- Entry to record
- Entry to item

Example

When you commit your inserts, updates, and deletes to the database, you do not see the input focus moving. However, internally the following navigation events must occur before commit processing begins:

- Exit current item
- Exit current record
- Exit current block

Navigation Triggers



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Navigation Triggers

Pre- and Post-	When-New-*"object"*-Instance
Fire during navigation	Fire after navigation
Does not fire if validation unit is larger than trigger object	Does fire when validation unit is larger than the trigger object
Allow unrestricted built-ins	Allow restricted and unrestricted built-ins
Handle failure by returning to initial object	Are not affected by failure

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Navigation Triggers

The navigation triggers can be subdivided into two general groups:

- Pre- and Post- navigation triggers
- When-New-“*object*”-Instance triggers

When Do Pre- and Post-Navigation Triggers Fire?

The Pre- and Post- navigation triggers fire during navigation, that is just before entry to or just after exit from the object specified as part of the trigger name.

Example

The Pre-Text-Item trigger fires just before entering a text item.

When Do When-New-*“object”*-Instance Triggers Fire?

The When-New-*”object”*-Instance triggers fire immediately after navigation to the object specified as part of the trigger name.

Example

The When-New-Item-Instance trigger fires immediately after navigation to a new instance of an item.

When Do Navigation Triggers Not Fire?

The Pre- and Post- navigation triggers do not fire if they belong to a unit that is smaller than the current validation unit. For instance, if the validation unit is Record, Pre- and Post-Text-Item triggers do not fire.

What Happens When a Navigation Trigger Fails?

If a Pre- or Post navigation trigger fails, the input focus returns to its initial location (where it was prior to the trigger firing). To the user, it appears that the input focus has not moved at all.

Note: Be sure that Pre- and Post- navigation triggers display a message on failure. Failure of a navigation trigger can cause a fatal error to your form. For example, failure of Pre-Form, Pre-Block, Pre-Record, or Pre-Text-Item on entry to the form will cancel execution of the form.

When-New-*"object"*-Instance Triggers

- **When-New-Form-Instance**
- **When-New-Block-Instance**
- **When-New-Record-Instance**
- **When-New-Item-Instance**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

SET_*"object"*_PROPERTY Examples

```
SET_FORM_PROPERTY(FIRST_NAVIGATION_BLOCK,
'S_ITEM');
```

```
SET_BLOCK_PROPERTY('S_ORD', ORDER_BY,
'CUSTOMER_ID');
```

```
SET_RECORD_PROPERTY(3, 'S_ITEM', STATUS,
QUERY_STATUS);
```

```
SET_ITEM_PROPERTY('CONTROL.stock_button',
ICON_NAME, 'stock');
```

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Using the When-New-*“object”*-Instance Triggers

If you include complex navigation paths through your application, you may want to check or set initial conditions when the input focus arrives in a particular block, record, or item. Use the following triggers to do this:

Trigger	**Fires**
When-New-Form-Instance	Whenever Form Builder runs a form, after successful navigation into a form
When-New-Block-Instance	After successful navigation into a block
When-New-Record-Instance	After successful navigation into the record
When-New-Item-Instance	After successful navigation to a new instance of the item

Initializing Form Builder Objects

Use the When-New-“*object*”-Instance triggers, along with the SET_*“object”*_PROPERTY built-in subprograms to initialize Form Builder objects. These triggers are particularly useful if you conditionally require a default setting.

Example

The following example of a When-New-Block-Instance trigger conditionally sets the DELETE ALLOWED property to FALSE.

```
IF GET_APPLICATION_PROPERTY(username) = 'SCOTT' THEN
SET_BLOCK_PROPERTY('S_ITEM',DELETE_ALLOWED, PROPERTY_FALSE);
END IF;
```

Example

Perform a query of all orders, when the ORDERS form is run, by including the following code in your When-New-Form-Instance trigger:

```
EXECUTE_QUERY;
```

The Pre- and Post-Triggers

- **Pre/Post-Form**
- **Pre/Post-Block**
- **Pre/Post-Record**
- **Pre/Post-Text-Item**

Copyright © Oracle Corporation, 1999. All rights reserved.

Post-Block Trigger Example

Disabling Stock_Button when leaving CONTROL block:

```
SET_ITEM_PROPERTY('CONTROL.stock_button', enabled,
property_false);
```

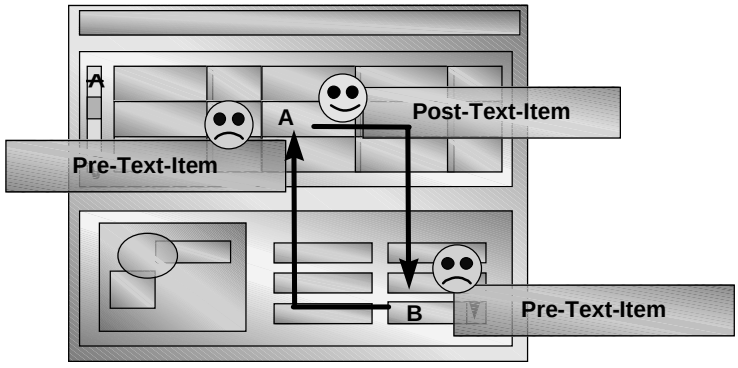
Copyright © Oracle Corporation, 1999. All rights reserved.

Using the Pre- and Post-Triggers

Trigger Type	Use to
Pre-Form	• Validate - User - Time of day • Initialize control blocks Call another form to display messages
Post-Form	• Perform housekeeping Erase global variables • Before exit Display messages to user
Pre-Block	• Authorize access to the block • Set global variables
Post-Block	• Validate the last record that had input focus • Test a condition and prevent the user from leaving
Pre-Record	• Set global variables
Post-Record	• Clear global variables • Set a visual attribute for an item as the user scrolls down through a set of records • Perform cross field validation
Pre-Text-Item	• Derive a complex default value • Record the previous value of a text item
Post-Text-Item	• Calculate or change item values

Note: Define Pre- and Post-Text-Item triggers at item level, Pre- and Post-Block at block level, and Pre- and Post-Form at form level. Pre- and Post-Text-Item triggers fire only for text items.

The Navigation Trap



Copyright © Oracle Corporation, 1999. All rights reserved.

The Navigation Trap

You have seen that the Pre- and Post- navigation triggers fire during navigation, and when they fail the internal cursor attempts to return to the current item (SYSTEM.CURSOR_ITEM).

The diagram on the opposite page illustrates the navigation trap. This can occur when a Pre- or Post- navigation trigger fails and attempts to return the logical cursor to its initial item. However, if the initial item has a Pre-Text-Item trigger that also fails the cursor has nowhere to go, and a fatal error occurs.

Note: Be sure to code against navigation trigger failure.

Navigation in Triggers

- **When-New-Item-Instance**

```
IF CHECKBOX_CHECKED('S_ORD.order_filled')THEN
     SET_ITEM_PROPERTY('S_ORD.date_shipped',
     UPDATE_ALLOWED, property_true);
      GO_ITEM('S_ORD.date_shipped');
END IF;
```

- **Pre-Text-Item**

```
IF CHECKBOX_CHECKED('S_ORD.order_filled') THEN
     SET_ITEM_PROPERTY('S_ORD.date_shipped',
     UPDATE_ALLOWED, property_true);
      GO_ITEM('S_ORD.date_shipped');
END IF;
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Navigation in Triggers

You can initiate navigation programmatically by calling the built-in subprograms, such as GO_ITEM and PREVIOUS_BLOCK from triggers.

Built-in Routines for Navigation	Function
GO_FORM	Navigates to an open form (in a multiple form application)
GO_BLOCK	Navigates to an indicated block
GO_ITEM	Navigates to an indicated item
GO_RECORD	Navigates to a specific record
NEXT_BLOCK	Navigates to the next enterable block
NEXT_ITEM	Navigates to the next enterable item
NEXT_KEY	Navigates to the next enterable, primary key item
NEXT_RECORD	Navigates to the first enterable item in the next record
NEXT_SET	Fetches another set of records from the database and navigates to the first record that the fetch retrieves
UP	Navigates to the instance of the current item in the previous record
DOWN	Navigates to the instance of the current item in the next record
PREVIOUS_BLOCK	Navigates to the previous enterable block
PREVIOUS_ITEM	Navigates to the previous enterable item
PREVIOUS_RECORD	Navigates to the previous record
SCROLL_UP	Scrolls the block so that the records above the top visible one display
SCROLL_DOWN	Scrolls the block so that the records below the bottom visible one display

Summary

- **Controlling navigation through properties**
- **Internal navigation**
- **Navigation triggers**
  - **When-New-“object”-Instance**
  - **Pre-**
  - **Post-**
- **Navigation trap**
- **Navigation in triggers**

Copyright © Oracle Corporation, 1999. All rights reserved.

Summary

In this lesson, you learned at the different methods of forcing visible navigation and also the invisible events.

- You can control navigation through the following properties:
  - Form module properties
  - Data block properties
  - Item properties
- Internal navigation events also occur.
- Navigation triggers:
  - When-New-“*object*”-Instance
  - Pre- and Post-
- Avoid the navigation trap.
- Navigation built-ins are available.

Practice 19 Overview

This practice covers the following topics:

- **Executing a query at form startup**
- **Populating product images when cursor arrives on each record of the S_ITEM block**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 19 in Appendix A, “Practice Solutions.”

Practice 19 Overview

In this practice, you use When-New-*"object"*-Instance triggers to populate the Product_Image item as the operator navigates between records in the ORDG*XX* form. Also, you provide a trigger to automatically perform query at form startup.

- Executing a query at form startup
- Populating product images when the cursor arrives on each record of the S_ITEM block

Practice 19

1. Write a When-New-Form-Instance trigger on the ORDG*XX* form to execute a query at form startup.

   Use the EXECUTE_QUERY built-in.

2. Write a trigger that fires as the cursor arrives in each record of the S_ITEM block, and populate the Product_Image item with a picture of the product, if one exists. Use Get_Product_Image function for this purpose.

   Get_Product_Image function is already created for you. This function returns the image file name for the given product number. If a file is not found, the function returns "`No file`."

   ```
   FUNCTION get_product_image (product_number IN NUMBER) RETURN
   VARCHAR2 IS
   v_filename VARCHAR2(20);
   BEGIN
   SELECT s_image.filename INTO v_filename
   FROM   s_image, s_product
   WHERE  s_image.id = s_product.image_id
   AND s_product.id = product_number;
   IF v_filename is null THEN
   v_filename := 'No file';
   END IF;
   RETURN v_filename;
   EXCEPTION
   WHEN no_data_found THEN return('No file');
   END;
   ```

   If the function returns a usable filename, your trigger should pass this name to the READ_IMAGE_FILE built-in.

3. Define the same trigger type and code on the S_ORD block.

   This will display the image for the first line item's product if the operator changes the displayed order.

4. Is there another trigger where you might also want to place this code?

5. Save, compile, and run the form to test.

20

Transaction Processing

Objectives

After completing this lesson, you should be able to do the following:

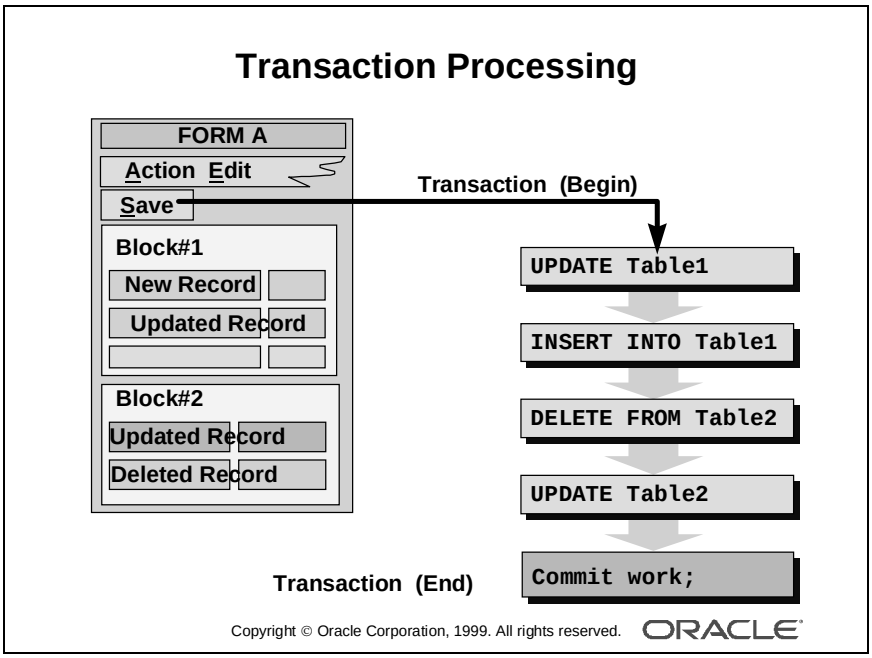
- **Describe details of commit processing and commit triggers**
- **Supplement transaction processing by using triggers**
- **Allocate sequence numbers to records as they are applied to tables**
- **Implement array DML**

Copyright © Oracle Corporation, 1999. All rights reserved.

Introduction

Overview

While applying a user’s changes to the database, the Oracle Developer Form Builder enables you to make triggers fire in order to alter or add to the default behavior. This lesson shows you how to build triggers that can perform additional tasks during this stage of a transaction.


Transaction Processing
FORM A
Action Edit
Save
Block#1
New Record
Updated Record
Block#2
Updated Record
Deleted Record
Transaction (Begin)
UPDATE Table1
INSERT INTO Table1
DELETE FROM Table2
UPDATE Table2
Commit work;
Transaction (End)
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Transaction Processing

When Form Builder is asked to save the changes made in a form by the user, a process takes place involving events in the current database transaction. This process includes:

- Default Form Builder transaction processing: Applies the user's changes to the base tables
- Firing transactional triggers: Needed to perform additional or modified actions in the saving process defined by the designer

When all of these actions are successfully completed, Form Builder commits the transaction, making the changes permanent.

What Happens in Transaction Processing?

The transaction process occurs as a result of either of the following actions:

- The user presses [Save] or selects Action—>Save from the menu, or clicks on the Save button on the default Form toolbar.
- The COMMIT_FORM built-in procedure is called from a trigger.

In either case, the process involves two phases, posting and committing.

Post Posting writes the user's changes to the base tables, using implicit INSERT, UPDATE, and DELETE statements generated by Form Builder. The changes are applied in block sequence order as they appear in the Object Navigator at design time. For each block, deletes are performed first, followed by inserts and updates. Transactional triggers fire during this cycle if defined by the designer.

The built-in procedure POST alone can invoke this posting process.

Commit This performs the database commit, making the applied changes permanent and releasing locks.

Transaction Processing

Transaction processing includes two phases:

- **Post:**
  - **Writes record changes to base tables**
  - **Fires transactional triggers**
- **Commit: Performs database commit**

Errors result in:

- **Rollback of the database changes**
- **Error message**

Copyright © Oracle Corporation, 1999. All rights reserved.

Rollbacks

Form Builder will roll back applied changes to a savepoint if an error occurs in its default processing, or when a transactional trigger fails.

By default, the user is informed of the error through a message, and a failing insert or update results in the record being redisplayed. The user can then attempt to correct the error before trying to save again.

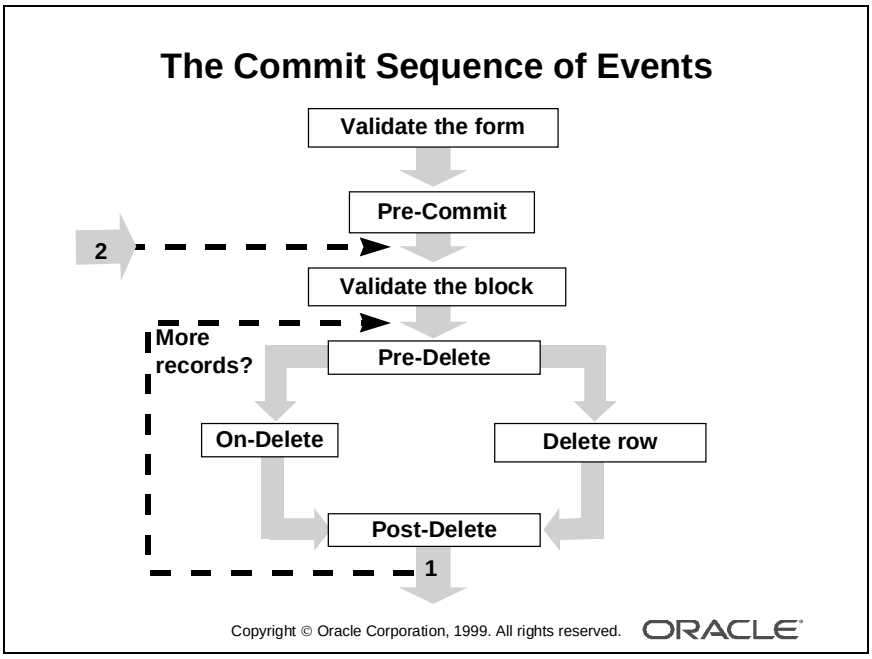
Savepoints

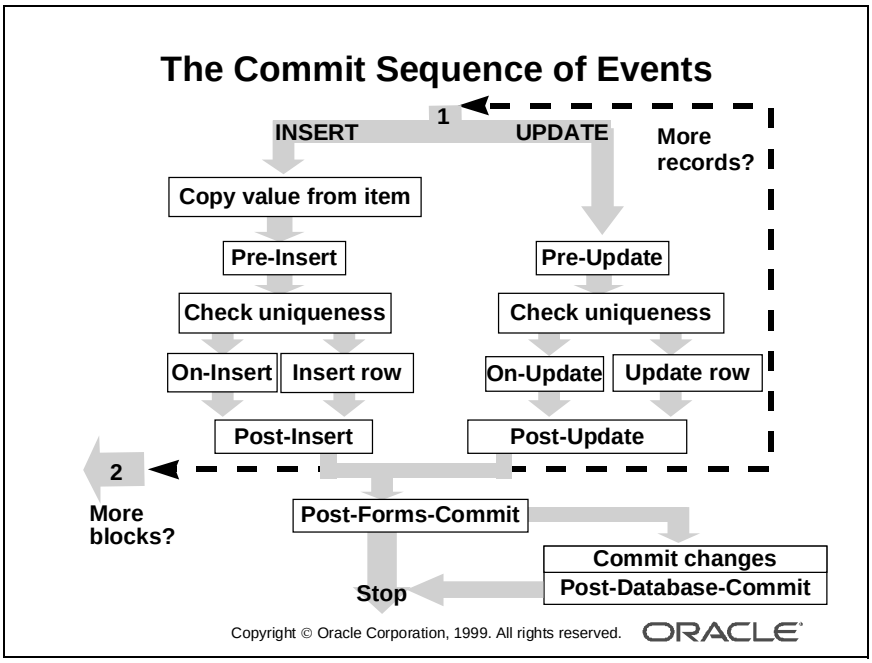
Form Builder issues savepoints in a transaction automatically, and will roll back to the latest savepoint if certain events occur. Generally, these savepoints are for Form Builder internal use, but certain built-ins, such as the EXIT_FORM built-in procedure, can request a rollback to the latest savepoint by using the TO_SAVEPOINT option.

Locking

When you update or delete base table records in a form application, database locks are automatically applied. Locks also apply during the posting phase of a transaction, and for DML statements that you explicitly use in your code.

Note: The SQL statements COMMIT, ROLLBACK, and SAVEPOINT cannot be called from a trigger directly. If encountered in a client-side program unit, Form Builder treats COMMIT as the COMMIT_FORM built-in, and ROLLBACK as the CLEAR_FORM built-in.


The Commit Sequence of Events
Validate the form
Pre-Commit
2
Validate the block
More records?
Pre-Delete
On-Delete
Delete row
Post-Delete
1
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE


The Commit Sequence of Events
1
INSERT
UPDATE
More records?
Copy value from item
Pre-Insert
Pre-Update
Check uniqueness
Check uniqueness
On-Insert
Insert row
On-Update
Update row
Post-Insert
Post-Update
2
More blocks?
Post-Forms-Commit
Commit changes
Post-Database-Commit
Stop
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

The Commit Sequence of Events

The commit sequence of events (when the Array DML size is 1) is as follows:

1. Validate the form.
2. Process savepoint.
3. Fire the Pre-Commit trigger.
4. Validate the block (for all blocks in sequential order).

   For all deleted records of the block (in reverse order of deletion):

   - Fire the Pre-Delete trigger.
   - Delete the row from the base table or fire the On-Delete trigger.
   - Fire the Post-Delete trigger.

   For all inserted or updated records of the block in sequential order:

   If it is an inserted record:

   - Copy Value From Item.
   - Fire the Pre-Insert trigger.
   - Check the record uniqueness.
   - Insert the row into the base table or fire the On-Insert trigger.
   - Fire the Post-Insert trigger.

   If it is an updated record:

   - Fire the Pre-Update trigger.
   - Check the record uniqueness
   - Update the row in the base table or fire the On-Update trigger.
   - Fire the Post-Update trigger.
5. Fire the Post-Forms-Commit trigger.

If the current operation is COMMIT, then:

6. Issue an SQL-COMMIT statement.
7. Fire the Post-Database-Commit trigger.

Characteristics of Commit Triggers

- **Pre-Commit: Fires once if form changes are made or uncommitted changes are posted**
- **Pre- and Post-DML**
- **On-DML: Fires per record, replacing default DML on row**

  Use DELETE_RECORD, INSERT_RECORD, UPDATE_RECORD built-ins

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Characteristics of Commit Triggers

- **Post-Forms-Commit: Fires once even if no changes are made**
- **Post-Database-Commit: Fires once even if no changes are made**

Note: A commit-trigger failure causes a rollback to the savepoint.

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Characteristics of Commit Triggers

You have already seen when commit triggers fire during the normal flow of commit processing. The following table gives more detailed information regarding the conditions under which these triggers fire.

Trigger	**Characteristic**
Pre-Commit	Fires once during commit processing, before base table blocks are processed; fires if there are changes to base table items in the form or if changes have been posted but not yet committed (This trigger always fires in case of uncommitted posts, even if there are no changes to post.)
Pre- and Post-DML	Fire for each record that is marked for insert, update, or delete, just before or after the row is inserted, updated, or deleted in the database
On-DML	Fires for each record that is marked for insert, update, or delete when Forms would typically issue its INSERT, UPDATE, or DELETE statement (These triggers replace the DML statements. Include a call to the INSERT_RECORD, UPDATE_RECORD, or DELETE_RECORD built-in to perform the default processing for these triggers.)
Post-Forms-Commit	Fires once during commit processing, after base table blocks are processed but before the SQL-COMMIT statement is issued; even fires if there are no changes to post or commit.
Post-Database-Commit	Fires once during commit processing, after the SQL-COMMIT statement is issued; even fires if there are no changes to commit (This is also true for the SQL-COMMIT statement itself.)

Note: If a commit trigger—except for the Post-Database-Commit trigger—fails, the transaction is rolled back to the savepoint that was set at the beginning of the current commit processing. This also means that earlier, not yet committed posts are not rolled back.

Commit Triggers Uses

Pre-Commit	**Check user authorization; set up special locking**
Pre-Delete	**Journaling; implement foreign-key delete rule**
Pre-Insert	**Generate sequence numbers; journaling; automatically generated columns; check constraints**
Pre-Update	**Journaling; implement foreign-key update rule; auto-generated columns; check constraints**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Commit Triggers Uses

On-Insert/Update/Delete	**Replace default block DML statements**
Post-Forms-Commit	**Check complex multirow constraints**
Post-Database-Commit	**Test commit success; test uncommitted posts**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Common Uses for Commit Triggers

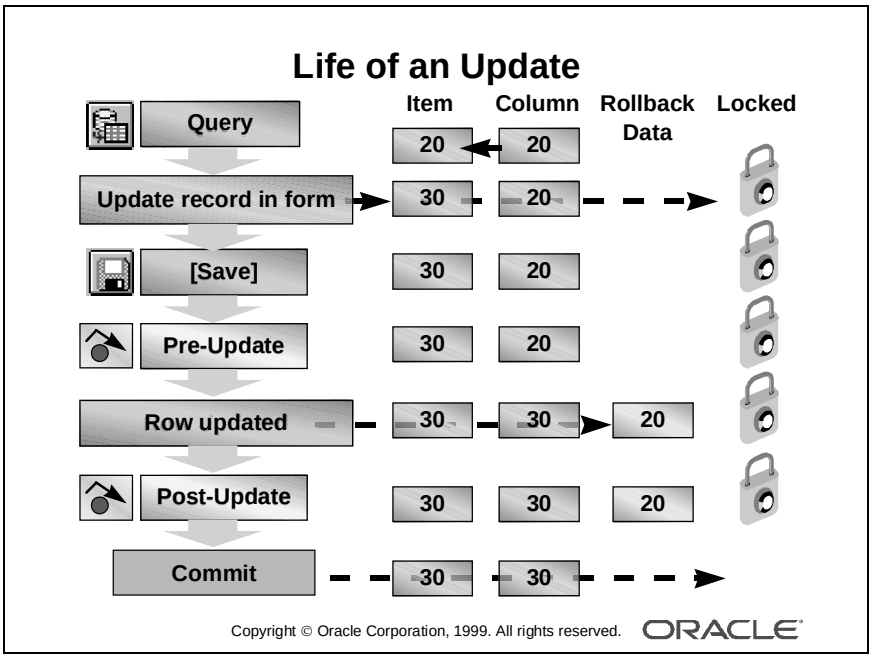
Once you know when a commit trigger fires, you should be able to choose the right commit trigger for the functionality that you want. To help you with this, the most common uses for commit triggers are mentioned below.

Trigger	**Common Use**
Pre-Commit	Checks user authorization; sets up special locking requirements
Pre-Delete	Writes to journal table; implements restricted or cascade delete
Pre-Insert	Writes to journal table; fills automatically generated columns; generates sequence numbers; checks constraints
Pre-Update	Writes to journal table; fills automatically generated columns; checks constraints; implements restricted or cascade update
Post-Delete, Post-Insert, Post-Update	Seldom used
On-Delete, On-Insert On-Update	Replaces default block DML statements; for example, to implement a pseudodelete or to update a join view
Post-Forms-Commit	Checks complex multirow constraints
Post-Database-Commit	Determines if commit was successful; determines if there are posted, uncommitted changes

Where possible, implement functionality such as writing to a journal table, automatically supplying column values, and checking constraints in the server.

Note: Locking is also needed for transaction processing. You can use the On-Lock trigger if you want to amend the default locking of Form Builder.

Use DML statements in commit triggers only; otherwise the DML statements are not included in the administration kept by Form Builder concerning commit processing. This may lead to unexpected and unwanted results.


Life of an Update
Item
Column
Rollback Data
Locked
Query
20
20
Update record in form
30
20
[Save]
30
20
Pre-Update
30
20
Row updated
30
30
20
Post-Update
30
30
20
Commit
30
30
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Life of an Update

To help you decide where certain trigger actions can be performed, consider an update operation as an example.

The price of a product is being updated in a form. After the user queries the record, the following events occur:

1. The user updates the Price item. This is now different from the corresponding database column.
2. The user saves the change, initiating the transaction process.
3. The Pre-Update trigger fires (if present). At this stage, the item and column are still different, because the update has not been applied to the base table. The trigger could compare the two values, for example, to make sure the new price is not lower than the existing one.
4. Form Builder applies the user's change to the database row. The item and column are now the same.
5. The Post-Update trigger fires (if present). It is too late to compare the item against the column, because the update has already been applied. However, the Oracle database retains the old column value as rollback data, so that a failure of this trigger reinstates the original value.
6. Form Builder issues the database commit, thus discarding the rollback data, releasing locks, and making the changes permanent. The user receives the message "`Transaction Completed...`".

Delete Validation

- **Pre-Delete trigger**
- **Final checks before row deletion**

```
DECLARE
      CURSOR C1 IS
      SELECT 'anything' FROM S_ORD
      WHERE customer_id = :S_CUSTOMER.id;
BEGIN
      OPEN C1;
      FETCH C1 INTO :GLOBAL.dummy;
      IF C1%FOUND THEN
           MESSAGE('There are orders for this customer!');
           RAISE form_trigger_failure;
      ELSE
           CLOSE C1;
      END IF;
END;
```

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Delete Validation

Master-detail blocks that are linked by a relation with the nonisolated deletion rule automatically prevent master records from being deleted in the form if matching detail rows exist.

You may, however, wish to implement a similar check, as follows, when a deletion is applied to the database:

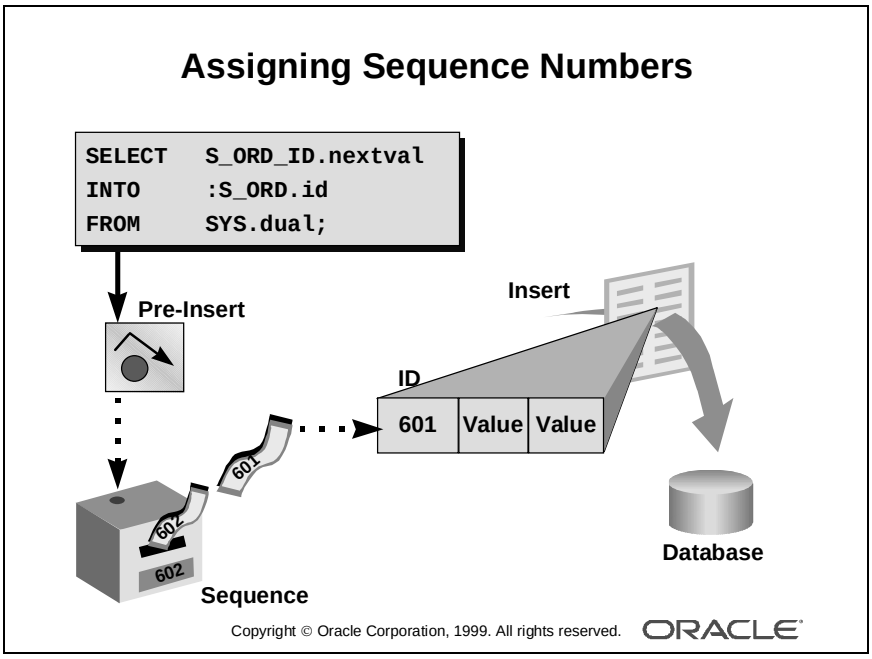
- A final check to ensure that no dependent detail rows have been inserted by another user since the master record was marked for deletion in the form (In an Oracle database, this is usually performed by a constraint or a database trigger.)
- A final check against form data, or checks that involve actions within the application

Note: If you select the “Enforce data integrity” check box in the Data Block Wizard, Form Builder automatically creates the related triggers to implement constraints.

Example

This Pre-Delete trigger on the CUSTOMER block of the CUSTOMERS form prevents deletion of rows if there are existing orders for the customer.

```
DECLARE
   CURSOR C1 IS
   SELECT 'anything' FROM S_ORD
   WHERE customer_id = :S_CUSTOMER.id;
BEGIN
   OPEN C1;
   FETCH C1 INTO :GLOBAL.dummy;
   IF C1%FOUND THEN
      CLOSE C1;
      MESSAGE('There are orders for this customer!');
      RAISE form_trigger_failure;
   ELSE
      CLOSE C1;
   END IF;
END;
```


Assigning Sequence Numbers
SELECT S_ORD_ID.nextval
INTO :S_ORD.id
FROM SYS.dual;
Pre-Insert
Insert
ID
601
Value
Value
601
602
602
Sequence
Database
Copyright © Oracle Corporation, 1999. All rights reserved.
ORACLE

Assigning Sequence Numbers to Records

You will recall that you can assign default values for items from an Oracle sequence, to automatically provide unique keys for records on their creation. However, if the user does not complete a record, the assigned sequence number is "wasted."

An alternative method is to assign unique keys to records from a Pre-Insert trigger, just before their insertion in the base table, by which time the user has completed the record and issued the Save.

Assigning unique keys in the posting phase can:

- Reduce gaps in the assigned numbers
- Reduce data traffic on record creation, especially if records are discarded before saving

Example

This Pre-Insert trigger on the S_ORD block assigns an Order ID from the sequence S_ORD_ID, which will be written to the ID column when the row is subsequently inserted.

```
SELECT S_ORD_ID.nextval
INTO   :S_ORD.id
FROM   SYS.dual;
```

Note: The Insert Allowed and Keyboard Navigable properties on :S_ORD.id should be No, so that the user does not enter an ID manually.

You can also assign sequence numbers from a table. If you use this method, then two transactional triggers are usually involved:

- Use Pre-Insert to select the next available number from the sequence table (locking the row to prevent other users from selecting the same value) and increment the value by the required amount.
- Use Post-Insert to update the sequence table, recording the new upper value for the sequence.

Keeping an Audit Trail

- **Write changes to nonbase tables.**
- **Gather statistics on applied changes.**

Post-Insert example:

```
:GLOBAL.insert_tot :=
  TO_CHAR(TO_NUMBER(:GLOBAL.insert_tot)+1);
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Keeping an Audit Trail

You may want to use the Post event transactional triggers to record audit information about the changes applied to base tables. In some cases, this may involve duplicating inserts or updates in backup history tables, or recording statistics each time a DML operation occurs.

If the base table changes are committed at the end of the transaction, the audit information will also be committed.

Example

This Post-Update trigger writes the current record ID to the UPDATE_AUDIT table, along with a time stamp and the user who performed the update.

```
INSERT INTO update_audit (id, timestamp, who_did_it)
VALUES  ( :S_ORD.id, SYSDATE, USER );
```

Example

This Post-Insert trigger adds to a running total of Inserts for the transaction, which is recorded in the global variable INSERT_TOT. (This global variable is initialized at the start of posting, and recorded in a table at the end, as discussed later.)

```
:GLOBAL.insert_tot := TO_CHAR(TO_NUMBER(:GLOBAL.insert_tot)+1);
```

Testing the Result of Trigger DML

- SQL%FOUND
- SQL%NOTFOUND
- SQL%ROWCOUNT

```
UPDATE S_ORD
 SET date_shipped = SYSDATE
 WHERE id = :S_ORD.id;
IF SQL%NOTFOUND THEN
  MESSAGE('Record not found in database');
  RAISE form_trigger_failure;
END IF;
```

Copyright © Oracle Corporation, 1999. All rights reserved.

Obtaining Cursor Information in PL/SQL

When you perform DML in transactional triggers, you may need to test the results.

Unlike SELECT statements, DML statements do not raise exceptions when zero or multiple rows are processed. PL/SQL provides some useful attributes for obtaining results from the implicit cursor used to process the latest SQL statement (in this case, DML).

PL/SQL Cursor Attribute	Values
SQL%FOUND	TRUE: Indicates > 0 rows processed FALSE: Indicates 0 rows processed
SQL%NOTFOUND	TRUE: Indicates 0 rows processed FALSE: Indicates > 0 rows processed
SQL%ROWCOUNT	Integer indicating the number of rows processed

Example

This When-Button-Pressed trigger records the date of posting as the date shipped for the current Order record. If a row is not found by the UPDATE statement, an error is reported.

```
UPDATE S_ORD
   SET date_shipped = SYSDATE
   WHERE id = :S_ORD.id;
IF SQL%NOTFOUND THEN
   MESSAGE('Record not found in database');
   RAISE form_trigger_failure;
END IF;
```

Note: Triggers containing base table DML can adversely affect the usual behavior of your form, because DML statements can cause some of the rows in the database to lock.

DML Statements Issued During Commit Processing

```
INSERT INTO base_table   (base_column, base_column,...)
VALUES                   (:base_item, :base_item, ...)
```

```
UPDATE   base_table
SET      base_column = :base_item, base_column =
            :base_item, ...
WHERE    ROWID = :ROWID
```

```
DELETE   FROM base_table
WHERE    ROWID = :ROWID
```

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

DML Statements Issued During Commit Processing

Rules:

- **DML statements may fire database triggers.**
- **Form Builder uses and retrieves ROWID.**
- **The Update Changed Columns Only and Enforce Column Security properties affect UPDATE statements.**
- **Locking statements are not issued.**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

DML Statements Issued During Commit Processing

If you have not altered default commit processing, Form Builder issues DML statements at commit time for each database record that is inserted, updated, or deleted.

```
INSERT INTO base_table(base_column, base_column, ...)
VALUES (:base_item,  :base_item,  ...)


UPDATE base_table
SET base_column = :base_item, base_column = :base_item, ...
WHERE ROWID = :ROWID


DELETE  FROM base_table
WHERE   ROWID = :ROWID
```

Rules

- These DML statements may fire associated database triggers.
- Form Builder uses the ROWID construct only when the Key mode block property is set to Automatic (the default).
- If Form Builder successfully inserts a row in the database, it also retrieves the ROWID for that row.
- If the Update Changed Columns Only block property is set to Yes, only base columns with changed values are included in the UPDATE statement.
- If the Enforce Column Security block property is set to Yes, all base columns for which the current user has no update privileges are excluded from the UPDATE statement.

Locking statements are not issued by Form Builder during default commit processing; they are issued as soon as a user updates or deletes a record in the form. If you set the Locking mode block property to delayed, Form Builder waits to lock the corresponding row until commit time.

Overriding Default Transaction

Additional transactional triggers:

Trigger	Do-the-Right-Thing Built-in
On-Check-Unique	CHECK_RECORD_UNIQUENESS
On-Column-Security	ENFORCE_COLUMN_SECURITY
On-Commit	COMMIT_FORM
On-Rollback	ISSUE_ROLLBACK
On-Savepoint	ISSUE_SAVEPOINT
On-Sequence-Number	GENERATE_SEQUENCE_NUMBER

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Overriding Default Transaction Processing

You have already seen that some commit triggers can be used to replace the default DML statements that Form Builder issues during commit processing. You can use several other triggers to override the default transaction processing of Form Builder.

Transactional Triggers

All triggers that are related to accessing a data source are called *transactional triggers*. Commit triggers form a subset of these triggers. Other examples include triggers that fire during logon and logout or during queries performed on the data source.

Additional Transactional Triggers for Commit Processing

Trigger	**Do-the-Right-Thing Built-in**
On-Check-Unique	CHECK_RECORD_UNIQUENESS
On-Column-Security	ENFORCE_COLUMN_SECURITY
On-Commit	COMMIT_FORM
On-Rollback	ISSUE_ROLLBACK
On-Savepoint	ISSUE_SAVEPOINT
On-Sequence-Number	GENERATE_SEQUENCE_NUMBER

Overriding Default Transaction

Transactional triggers for logging on and off:

Trigger	Do-the-Right-Thing Built-in
Pre-Logon	-
Pre-Logout	-
On-Logon	LOGON
On-Logout	LOGOUT
Post-Logon	-
Post-Logout	-

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Transactional Triggers for Logging On and Off

Trigger	Do-the-Right-Thing Built-in
Pre-Logon	-
Pre-Logout	-
On-Logon	LOGON
On-Logout	LOGOUT
Post-Logon	-
Post-Logout	-

Uses for Transactional Triggers

- Transactional triggers, except for the commit triggers, are primarily intended to access certain data sources other than Oracle.
- The logon and logoff transactional triggers can also be used with Oracle databases to change connections at run time.

Running with Data Sources Other than Oracle

- Three ways to run against data sources other than Oracle
  - Oracle Open Gateways
  - Oracle Open Client Adapter for ODBC
  - Write appropriate transactional triggers

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Running with Data Sources Other than Oracle

- Connecting with Open Gateway:
  - Cursor and Savepoint mode form module properties
  - Key mode and Locking mode block properties
- Using transactional triggers:
  - Call 3GL programs
  - Database data block property

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Running Against Data Sources Other than Oracle

Three Ways to Run Against Data Sources Other than Oracle

- Use Oracle Open Gateway products.
- Use Oracle Open Client Adapter for ODBC.
- Write the appropriate set of Transactional triggers.

Connecting with Open Gateway

When you connect to a data source other than Oracle with an Open Gateway product, you should be aware of these transactional properties:

- Cursor mode form module property
- Savepoint mode form module property
- Key mode block property
- Locking mode block property

You can set these properties to specify how Form Builder should interact with your data source. The specific settings depend on the capabilities of the data source.

Using Transactional Triggers

If no Open Gateway or Open Client Adapter drivers exist for your data source, you must define transactional triggers. From these triggers, you must call 3GL programs that implement the access to the data source.

Database Data Block Property

This block property identifies a block as a transactional control block; that is, a control block that should be treated as a base table block. Setting this property to Yes ensures that transactional triggers will fire for the block, even though it is not a base table block. If you set this property to Yes, you must define all On-Event transactional triggers, otherwise you will get an error during form generation.

Getting and Setting the Commit Status

- **What is commit status?**
- **SYSTEM.RECORD_STATUS:**
  - **NEW**
  - **INSERT (also caused by control items)**
  - **QUERY**
  - **CHANGED**

Copyright © Oracle Corporation, 1999. All rights reserved.

Getting and Setting the Commit Status

If you want to process a record in your form, it is often useful to know if the record is in the database or if it has been changed, and so on. You can use system variables and built-ins to obtain this information.

What Is the Commit Status of a Record?

The commit status of a record of a base table block determines how the record will be processed during the next commit process. For example, the record can be inserted, updated, or not processed at all.

The Four Values of SYSTEM.RECORD_STATUS

Value	Description
NEW	Indicates that the record has been created, but that none of its items have been changed yet (The record may have been populated by default values.)
INSERT	Indicates that one or more of the items in a newly created record have been changed (The record will be processed as an insert during the next commit process if its block has the CHANGED status; see below. Note that when you change a control item of a NEW record, the record status also becomes INSERT.)
QUERY	Indicates that the record corresponds to a row in the database, but that none of its base table items have been changed
CHANGED	Indicates that one or more base table items in a database record have been changed (The record will be processed as an update (or delete) during the next commit process.)

Getting and Setting the Commit Status

- **SYSTEM.BLOCK_STATUS:**
  - **NEW (may contain records with status INSERT)**
  - **QUERY (also possible for control block)**
  - **CHANGED (block will be committed)**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Getting and Setting the Commit Status

- **SYSTEM.FORM_STATUS:**
  - **NEW**
  - **QUERY**
  - **CHANGED**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Three Values of SYSTEM.BLOCK_STATUS

Value	Description
NEW	Indicates that all records of the block have the status NEW (Note that a base table block with the status NEW may also contain records with the status INSERT caused by changing control items).
QUERY	Indicates that all records of the block have the status QUERY if the block is a base table block (A control block has the status QUERY if it contains at least one record with the status INSERT.)
CHANGED	Indicates that the block contains at least one record with the status INSERT or CHANGED if the block is a base table block (The block will be processed during the next commit process. Note that a control block cannot have the status CHANGED.)

Three Values of SYSTEM.FORM_STATUS

Value	Description
NEW	Indicates that all blocks of the form have the status NEW
QUERY	Indicates that at least one block of the form has status QUERY and all other blocks have the status NEW
CHANGED	Indicates that at least one block of the form has the status CHANGED

Getting and Setting the Commit Status

- System variables versus built-ins for commit status
- Built-ins for getting and setting commit status:
  - GET_BLOCK_PROPERTY
  - GET_RECORD_PROPERTY
  - SET_ RECORD _PROPERTY

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Getting and Setting the Commit Status

- Example: If the third record of block S_ORD is a changed database record, set the status back to QUERY.
- Warnings:
  - Do not confuse commit status with validation status.
  - The commit status is updated during validation.

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Using Built-ins to Get the Commit Status

The system variables SYSTEM.RECORD_STATUS and SYSTEM.BLOCK_STATUS apply to the record and block where the cursor is located. You can use built-ins to obtain the status of other blocks and records.

Built-in	Description
GET_BLOCK_PROPERTY	Use the STATUS property to obtain the block status of the specified block.
GET_RECORD_PROPERTY	Use the STATUS property to obtain the record status of the specified record in the specified block.
SET_RECORD_PROPERTY	Set the STATUS property of the specified record in the specified block to one of the following constants: • NEW_STATUS, INSERT_STATUS • QUERY_STATUS • CHANGED_STATUS

Example

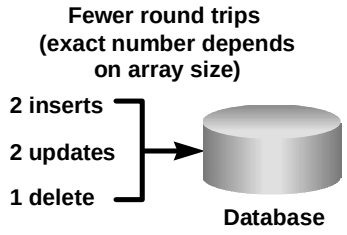
If the third record of the S_ORD block is a changed database record, set the status back to QUERY.

```
BEGIN
IF GET_RECORD_PROPERTY(3, 'S_ORD',status)= 'CHANGED' THEN
   SET_RECORD_PROPERTY(3, 'S_ORD', status, query_status);
END IF;
END;
```

Array DML

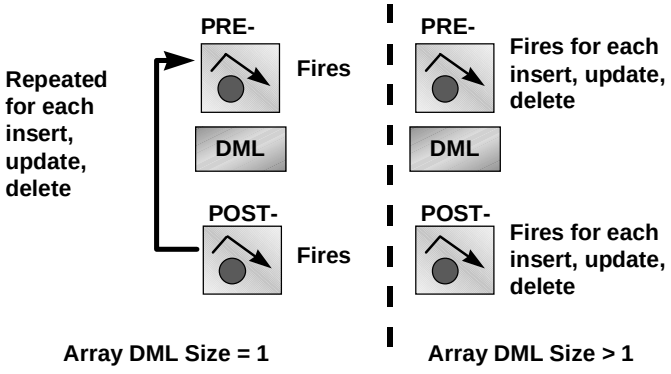
- Performs array inserts, updates, and deletes
- Vastly reduces network traffic

Empno	Ename	Job	Hiredate
1234	Jones	Clerk	01-Jan-95
1235	Smith	Clerk	01-Jan-95
1236	Adams	Clerk	01-Jan-95
1237	Clark	Clerk	01-Jan-95



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Effect of Array DML on Transactional Triggers



Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE

Array Processing

Overview

Array processing is an option in Form Builder that alters the way records are processed. The default behavior of Form Builder is to process records one at a time. By enabling array processing, you can process groups of records at a time, reducing network traffic and thereby increasing performance. With array processing, a structure (an array) containing multiple records is sent to or returned from the server for processing.

Form Builder supports both array fetch processing and array DML processing. For both querying and DML operations, you can determine the array size to optimize performance for your needs. This lesson focuses on array DML processing.

Array processing is available for query and DML operations for blocks based on tables, views, procedures, and subqueries; it is not supported for blocks based on transactional triggers.

Effect of Array DML on Transactional Triggers

With DML Array Size set to 1, the Pre-Insert, Pre-Update, and Pre-Delete triggers fire for each new, changed, and deleted record; the DML is issued, and the Post- trigger for that record fires.

With DML Array Size set to greater than 1, the appropriate Pre- triggers fire for all of the new, changed, and deleted rows; all of the DML statements are issued, and all of the Post- triggers fire.

If you change 100 rows and DML Array Size is 20, you get 100 Pre- triggers, 5 arrays of 20 DML statements, and 100 Post- triggers.

Implementing Array DML

1. **Enable the Array Processing option.**
2. **Specify a DML Array Size of greater than 1.**
3. **Specify block primary keys.**

Copyright © Oracle Corporation, 1999. All rights reserved.

How to Implement Array DML

1. To set preferences:
   - Select Tools—>Preferences.
   - Click the Runtime tab.
   - Select the Array Processing check box.
2. To set properties:
   - In the Object Navigator, select the Data Blocks node.
   - Double-click the Data Blocks icon to display the Property Palette.
   - Under the Advanced Database category, set the DML Array Size property to a number that represents the number of records in the array for array processing. You can also set this property programmatically.

Note: When the DML Array Size property is greater than 1, you must specify the primary key. Key mode can still be unique.

The Oracle server uses the ROWID to identify the row, except after an array insert. If you update a record in the same session that you inserted it, the server locks the record by using the primary key.

Summary

- **Post and commit phases**
- **Flow of commit processing**
- **DML statements issued during commit processing**
- **Characteristics and common uses of commit triggers**
- **Overriding default transaction processing**
- **Getting and setting the commit status**
- **Implementing Array DML**

Copyright © Oracle Corporation, 1999. All rights reserved.

Summary

This lesson showed you how to build triggers that can perform additional tasks during the save stage of a current database transaction.

- Transactions are processed in two phases:
  - Post: Applies form changes to the base tables and fires transactional triggers
  - Commit: Commits the database transaction
- Flow of commit processing
- DML statements issued during commit processing:
  - Based on base table items
  - UPDATE and DELETE statements use ROWID by default
- Characteristics of commit triggers:
  - The Pre-Commit, Post-Forms-Commit, and Post-Database-Commit triggers fire once per commit process, but consider uncommitted changes or posts.
  - The Pre-, On-, and Post-Insert, Update, and Delete triggers fire once per processed record.
- Common uses for commit triggers: check authorization, set up special locking requirements, generate sequence numbers, check complex constraints, replace default DML statements issued by Form Builder.
- Overriding default transaction processing:
  - Transactional On-Event triggers and “Do-the-Right-Thing” built-ins
  - Data sources other than Oracle use Open Gateway, ODBC, or transactional triggers
- Getting and setting the commit status:
  - System variables
  - Built-ins
- Array DML

Practice 20 Overview

This practice covers the following topics:

- **Automatically populating order IDs by using a sequence**
- **Automatically populating item IDs by adding the current highest order ID**
- **Customizing the commit messages in the CUSTOMERS form**
- **Customizing the login screen in the CUSTOMERS form**

Copyright © Oracle Corporation, 1999. All rights reserved.

Note

For solutions to this practice, see Practice 20 in Appendix A, “Practice Solutions.”

Practice 20 Overview

In this practice, you add transactional triggers to the ORDG*XX* form to automatically provide sequence numbers to records at save time. You also customize commit messages and the login screen in the CUSTG*XX* form.

- Automatically populating order IDs by using a sequence
- Automatically populating item IDs by adding the current highest order ID
- Customizing the commit messages in the CUSTOMERS form
- Customizing the login screen in the CUSTOMERS form

Practice 20

1. In the ORDG*XX* form write a transactional trigger on the S_ORD block that populates S_ORD.Id with the next value from the S_ORD_ID sequence.

   Create a Pre-Insert trigger that assigns a value from this sequence. If an exception occurs in the trigger, fail the trigger with a message.
2. In the S_ORD block, set the Enabled property of the ID item to No.
3. Save, compile, and run the form to test.

   Insert a new order. The unique ID for the order should appear when you save it.
4. Create a similar trigger on the S_ITEM block that assigns the Item_Id when a new record is saved.

   Derive this number by adding to the current highest Item_Id for the order. Perform the action in a Pre-Insert trigger. Set the Required and Enabled properties to No for Item_Id.
5. Save, compile, and run the form to test.

   Insert a new line-item record in the S_ITEM block, then save it.
6. Open the CUSTG*XX* form module. Create three global variables called GLOBAL.INSERT, GLOBAL.UPDATE, and GLOBAL.DELETE. These variables indicate respectively the number of inserts, updates, and deletes. You need to write Post-Insert, Post-Update, and Post-Delete triggers to initialize and increment the value of each global variable.

Practice 20 (continued)

7 Create a procedure called HANDLE_MESSAGE. Import the `pr20_10.txt` file. This procedure receives two arguments. The first one is a message number, and the second is a Boolean error indicator. This procedure uses the three global variables to display a customized commit message and then erases the global variables.

```
PROCEDURE handle_message( message_number IN NUMBER, IS_ERROR IN
BOOLEAN ) IS
BEGIN
   IF message_number IN ( 40400, 40406, 40407 ) THEN
      DEFAULT_VALUE( '0', 'GLOBAL.insert' );
      DEFAULT_VALUE( '0', 'GLOBAL.update' );
      DEFAULT_VALUE( '0', 'GLOBAL.delete' );
      MESSAGE('Save Ok: ' ||
      :GLOBAL.insert || 'records inserted, '||
      :GLOBAL.update || 'records updated, '||
      :GLOBAL.delete || 'records deleted !!!' );
   ELSIF is_error = TRUE THEN
      MESSAGE('ERROR: '|| ERROR_TEXT );
   ELSE
      MESSAGE( MESSAGE_TEXT );
   END IF;
END ;
```

Call the procedure when an error occurs. Pass the error code and TRUE. Call the procedure when a message occurs. Pass the message code and FALSE.

8 Open the CUSTG*XX* form module. Write an On-Logon trigger to control the number of connection tries. Use the LOGON_SCREEN built-in to simulate the default login screen and LOGON to connect to the database. You can import the `pr20_11.txt` file.

On-Logon at Form Level

```
DECLARE
   connected BOOLEAN := FALSE;
   tries NUMBER := 3;
   un VARCHAR2(30);
   pw VARCHAR2(30);
   cs VARCHAR2(30);
BEGIN
   SET_APPLICATION_PROPERTY(CURSOR_STYLE, 'DEFAULT');
   WHILE connected = FALSE and tries > 0 LOOP
      LOGON_SCREEN;
      un := GET_APPLICATION_PROPERTY( USERNAME );
      pw := GET_APPLICATION_PROPERTY( PASSWORD );
      cs := GET_APPLICATION_PROPERTY( CONNECT_STRING );
      LOGON( un, pw || '@' || cs, FALSE );
      IF FORM_SUCCESS THEN
         connected := TRUE ;
      END IF;
      tries := tries - 1;
   END LOOP;
   IF NOT CONNECTED THEN
      MESSAGE('Too many tries!');
      RAISE FORM_TRIGGER_FAILURE;
   END IF;
END;
```