

Oracle Forms Developer 10g: Build Internet Applications

Instructor Guide • Volume 2

D17251GC10

Edition 1.0

June 2004

D39560

ORACLE®

Author

Pam Gamer

**Technical Contributors
and Reviewers**

Alena Bugarova

Purjanti Chang

Laurent Dereac

Punita Handa

Mark Pare

Jasmin Robayo

Bryan Roberts

Divya Sandeep

Raza Siddiqui

John Soltani

Lex van der Werff

Editors

Nishima Sachdeva

Elizabeth Treacy

Publisher

Giri Venugopal

Copyright © 2004, Oracle. 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 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, Redwood Shores, CA 94065. Oracle Corporation does not warrant that this document is error-free.

Oracle and all references to Oracle Products 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.

Contents

Preface

I Introduction

- Objectives 1-2
- Course Objectives 1-3
- Course Content 1-5

1 Introduction to Oracle Forms Developer and Oracle Forms Services

- Objectives 1-2
- Internet Computing Solutions 1-3
- Plugging into the Grid 1-4
- Oracle Enterprise Grid Computing 1-5
- Oracle 10g Products and Forms Development 1-7
- Oracle Application Server 10g Architecture 1-8
- Oracle Application Server 10g Components 1-9
- Oracle Forms Services Overview 1-10
- Forms Services Architecture 1-11
- Benefits and Components of Oracle Developer Suite 10g 1-12
- Oracle Developer Suite 10g Application Development 1-13
- Oracle Developer Suite 10g Business Intelligence 1-14
- Oracle Forms Developer Overview 1-15
- Oracle Forms Developer: Key Features 1-16
- Summit Office Supply Schema 1-17
- Summit Application 1-18
- Summary 1-20

2 Running a Forms Developer Application

- Objectives 2-2
- Testing a Form: OC4J Overview 2-3
- Testing a Form: Starting OC4J 2-4
- Running a Form 2-5
- Running a Form: Browser 2-6
- The Java Runtime Environment 2-7
- Starting a Run-Time Session 2-8
- The Forms Servlet 2-11
- The Forms Client 2-12
- The Forms Listener Servlet 2-13
- The Runtime Engine 2-14
- What You See at Run Time 2-15
- Identifying the Data Elements 2-17
- Navigating a Forms Developer Application 2-18
- Modes of Operation: Enter-Query Mode 2-20
- Modes of Operation: Normal Mode 2-21
- Retrieving Data 2-22
- Retrieving Restricted Data 2-23
- Query/Where Dialog Box 2-25
- Inserting, Updating, and Deleting 2-27

Making Changes Permanent 2-29
Displaying Errors 2-30
Summary 2-31
Practice 2 Overview 2-34

3 Working in the Forms Developer Environment

Objectives 3-2
Forms Builder Key Features 3-3
Forms Builder Components: Object Navigator 3-4
Forms Builder Components: Property Palette 3-5
Forms Builder Components: Layout Editor 3-6
Forms Builder Components: PL/SQL Editor 3-7
Getting Started in the Forms Builder Interface 3-8
Forms Builder: Menu Structure 3-10
Blocks, Items, and Canvases 3-12
Navigation in a Block 3-14
Data Blocks 3-15
Forms and Data Blocks 3-17
Form Module Hierarchy 3-19
Customizing Your Forms Builder Session 3-21
Saving Preferences 3-23
Using the Online Help System 3-24
Forms Developer Executables 3-25
Forms Developer Module Types 3-27
Defining Forms Environment Variables for Run Time 3-29
Defining Forms Environment Variables for Design Time 3-30
Environment Variables and Y2K Compliance 3-32
Forms Files to Define Run-Time Environment Variables 3-34
Testing a Form: The Run Form Button 3-35
Summary 3-37
Practice 3 Overview 3-39

4 Creating a Basic Form Module

Objectives 4-2
Creating a New Form Module 4-3
Form Module Properties 4-6
Creating a New Data Block 4-8
Navigating the Wizards 4-10
Launching the Data Block Wizard 4-11
Data Block Wizard: Type Page 4-12
Data Block Wizard: Table Page 4-13

- Data Block Wizard: Finish Page 4-14
- Layout Wizard: Items Page 4-15
- Layout Wizard: Style Page 4-16
- Layout Wizard: Rows Page 4-17
- Data Block Functionality 4-18
- Template Forms 4-19
- Saving a Form Module 4-20
- Compiling a Form Module 4-21
- Module Types and Storage Formats 4-22
- Deploying a Form Module 4-24
- Text Files and Documentation 4-25
- Summary 4-26
- Practice 4 Overview 4-27

5 Creating a Master-Detail Form

- Objectives 5-2
- Form Block Relationships 5-3
- Data Block Wizard: Master-Detail Page 5-5
- Relation Object 5-7
- Creating a Relation Manually 5-8
- Join Condition 5-9
- Deletion Properties 5-10
- Modifying a Relation 5-11
- Coordination Properties 5-12
- Running a Master-Detail Form Module 5-13
- Modifying the Structure of a Data Block 5-14
- Modifying the Layout of a Data Block 5-15
- Summary 5-17
- Practice 5 Overview 5-18

6 Working with Data Blocks and Frames

- Objectives 6-2
- Managing Object Properties 6-3
- Displaying the Property Palette 6-4
- Property Palette: Features 6-5
- Property Controls 6-6
- Visual Attributes 6-8
- How to Use Visual Attributes 6-9
- Font, Pattern, and Color Pickers 6-10
- Controlling Data Block Behavior and Appearance 6-11
- Navigation Properties 6-12

- Records Properties 6-13
- Database Properties 6-15
- Scroll Bar Properties 6-18
- Controlling Frame Properties 6-19
- Displaying Multiple Property Palettes 6-21
- Setting Properties on Multiple Objects 6-22
- Copying Properties 6-24
- Creating a Control Block 6-26
- Deleting a Data Block 6-27
- Summary 6-28
- Practice 6 Overview 6-29

7 Working with Text Items

- Objectives 7-2
- Text Item Overview 7-3
- Creating a Text Item 7-4
- Modifying the Appearance of a Text Item: General and Physical Properties 7-6
- Modifying the Appearance of a Text Item: Records Properties 7-7
- Modifying the Appearance of a Text Item: Font and Color Properties 7-8
- Modifying the Appearance of a Text Item: Prompts 7-9
- Associating Text with an Item Prompt 7-10
- Controlling the Data of a Text Item 7-11
- Controlling the Data of a Text Item: Format 7-12
- Controlling the Data of a Text Item: Values 7-13
- Controlling the Data of a Text Item: Copy Value from Item 7-15
- Controlling the Data of a Text Item: Synchronize with Item 7-16
- Altering Navigational Behavior of Text Items 7-17
- Enhancing the Relationship Between Text Item and Database 7-18
- Adding Functionality to a Text Item 7-19
- Adding Functionality to a Text Item: Conceal Data Property 7-20
- Adding Functionality to a Text Item: Keyboard Navigable and Enabled 7-21
- Adding Functionality to a Text Item: Multi-line Text Items 7-22
- Displaying Helpful Messages: Help Properties 7-23
- Summary 7-24
- Practice 7 Overview 7-26

8 Creating LOVs and Editors

- Objectives 8-2
- Overview of LOVs and Editors 8-3
- LOVs and Record Groups 8-6
- Creating an LOV Manually 8-8

Creating an LOV with the LOV Wizard: SQL Query Page 8-9
Creating an LOV with the LOV Wizard: Column Selection Page 8-10
Creating an LOV with the LOV Wizard: Column Properties Page 8-11
Creating an LOV with the LOV Wizard: Display Page 8-12
Creating an LOV with the LOV Wizard: Advanced Properties Page 8-13
Creating an LOV with the LOV Wizard: Assign to Item Page 8-14
LOV Properties 8-15
Setting LOV Properties 8-16
LOVs: Column Mapping 8-17
Defining an Editor 8-19
Setting Editor Properties 8-20
Associating an Editor with a Text Item 8-21
Summary 8-22
Practice 8 Overview 8-23

9 Creating Additional Input Items

Objectives 9-2
Input Items Overview 9-3
Check Boxes Overview 9-4
Creating a Check Box 9-5
Converting an Existing Item into a Check Box 9-6
Creating a Check Box in the Layout Editor 9-7
Setting Check Box Properties 9-8
Check Box Mapping of Other Values 9-10
List Items Overview 9-11
Creating a List Item 9-13
Converting an Existing Item into a List Item 9-14
Creating a List Item in the Layout Editor 9-15
Setting List Item Properties 9-16
List Item Mapping of Other Values 9-17
Radio Groups Overview 9-18
Creating a Radio Group 9-19
Converting Existing Item to Radio Group 9-20
Creating Radio Group in Layout Editor 9-21
Setting Radio Properties 9-22
Radio Group Mapping of Other Values 9-23
Summary 9-24
Practice 9 Overview 9-25

10 Creating Noninput Items

Objectives 10-2
Noninput Items Overview 10-3

- Display Items 10-4
- Creating a Display Item 10-5
- Image Items 10-6
- Image File Formats 10-8
- Creating an Image Item 10-9
- Setting Image-Specific Item Properties 10-10
- Push Buttons 10-12
- Push Button Actions 10-13
- Creating a Push Button 10-14
- Setting Push Button Properties 10-15
- Calculated Items 10-16
- Creating a Calculated Item by Setting Properties 10-17
- Setting Item Properties for the Calculated Item 10-18
- Summary Functions 10-19
- Calculated Item Based on a Formula 10-20
- Rules for Calculated Item Formulas 10-21
- Calculated Item Based on a Summary 10-22
- Rules for Summary Items 10-23
- Creating a Hierarchical Tree Item 10-24
- Setting Hierarchical Tree Item Properties 10-25
- Bean Area Items 10-26
- Creating a Bean Area Item 10-27
- Setting Bean Area Item Properties 10-28
- The JavaBean at Run Time 10-29
- Summary 10-30
- Practice 10 Overview 10-32

11 Creating Windows and Content Canvases

- Objectives 11-2
- Windows and Canvases 11-3
- Window, Canvas, and Viewport 11-4
- The Content Canvas 11-5
- Relationship Between Windows and Content Canvases 11-6
- The Default Window 11-7
- Displaying a Form Module in Multiple Windows 11-8
- Creating a New Window 11-9
- Setting Window Properties 11-10
- GUI Hints 11-11
- Displaying a Form Module on Multiple Layouts 11-12
- Creating a New Content Canvas 11-13
- Setting Content Canvas Properties 11-15

Summary 11-16
Practice 11 Overview 11-17

12 Working with Other Canvas Types

Objectives 12-2
Overview of Canvas Types 12-3
The Stacked Canvas 12-4
Creating a Stacked Canvas 12-6
Setting Stacked Canvas Properties 12-8
The Toolbar Canvas 12-9
The MDI Toolbar 12-10
Creating a Toolbar Canvas 12-11
Setting Toolbar Properties 12-12
The Tab Canvas 12-13
Creating a Tab Canvas 12-14
Creating a Tab Canvas in the Object Navigator 12-15
Creating a Tab Canvas in the Layout Editor 12-16
Setting Tab Canvas, Tab Page, and Item Properties 12-17
Placing Items on a Tab Canvas 12-18
Summary 12-19
Practice 12 Overview 12-21

13 Introduction to Triggers

Objectives 13-2
Trigger Overview 13-3
Grouping Triggers into Categories 13-4
Defining Trigger Components 13-6
Trigger Type 13-7
Trigger Code 13-9
Trigger Scope 13-10
Specifying Execution Hierarchy 13-12
Summary 13-14

14 Producing Triggers

Objectives 14-2
Creating Triggers in Forms Builder 14-3
Creating a Trigger 14-4
Setting Trigger Properties 14-7
PL/SQL Editor Features 14-8
The Database Trigger Editor 14-10
Writing Trigger Code 14-11

- Using Variables in Triggers 14-13
- Forms Builder Variables 14-14
- Adding Functionality with Built-In Subprograms 14-16
- Limits of Use 14-18
- Using Built-In Definitions 14-19
- Useful Built-Ins 14-21
- Using Triggers: When-Button-Pressed Trigger 14-23
- Using Triggers: When-Window-Closed Trigger 14-24
- Summary 14-25
- Practice 14 Overview 14-27

15 Debugging Triggers

- Objectives 15-2
- The Debugging Process 15-3
- The Debug Console 15-4
- The Debug Console: Stack Panel 15-5
- The Debug Console: Variables Panel 15-6
- The Debug Console: Watch Panel 15-7
- The Debug Console: Form Values Panel 15-8
- The Debug Console: PL/SQL Packages Panel 15-9
- The Debug Console: Global/System Variables Panel 15-10
- The Debug Console: Breakpoints Panel 15-11
- The Debug Console 15-12
- Setting Breakpoints in Client Code 15-13
- Setting Breakpoints in Stored Code 15-14
- Debugging Tips 15-15
- Running a Form in Debug Mode 15-16
- Stepping Through Code 15-17
- Debug Example 15-18
- Summary 15-20
- Practice 15 Overview 15-21

16 Adding Functionality to Items

- Objectives 16-2
- Item Interaction Triggers 16-3
- Coding Item Interaction Triggers 16-5
- Interacting with Check Boxes 16-7
- Changing List Items at Run Time 16-8
- Displaying LOVs from Buttons 16-9
- LOVs and Buttons 16-11
- Populating Image Items 16-13

- Loading the Right Image 16-15
- Populating Hierarchical Trees 16-16
- Displaying Hierarchical Trees 16-18
- Interacting with JavaBeans 16-19
- Summary 16-25
- Practice 16 Overview 16-27

17 Run Time Messages and Alerts

- Objectives 17-2
- Run-Time Messages and Alerts Overview 17-3
- Detecting Run-Time Errors 17-5
- Errors and Built-Ins 17-7
- Message Severity Levels 17-9
- Suppressing Messages 17-11
- The `FORM_TRIGGER_FAILURE` Exception 17-13
- Triggers for Intercepting System Messages 17-15
- Handling Informative Messages 17-17
- Setting Alert Properties 17-19
- Planning Alerts 17-21
- Controlling Alerts 17-22
- `SHOW_ALERT` Function 17-24
- Directing Errors to an Alert 17-26
- Causes of Oracle Server Errors 17-27
- Trapping Server Errors 17-29
- Summary 17-30
- Practice 17 Overview 17-33

18 Query Triggers

- Objectives 18-2
- Query Processing Overview 18-3
- `SELECT` Statements Issued During Query Processing 18-5
- `WHERE` Clause 18-7
- `ONETIME_WHERE` Property 18-8
- `ORDER BY` Clause 18-9
- Writing Query Triggers: Pre-Query Trigger 18-10
- Writing Query Triggers: Post-Query Trigger 18-11
- Writing Query Triggers: Using `SELECT` Statements in Triggers 18-12
- Query Array Processing 18-13
- Coding Triggers for Enter-Query Mode 18-15
- Overriding Default Query Processing 18-19
- Obtaining Query Information at Run Time 18-22

Summary 18-25
Practice 18 Overview 18-27

19 Validation

Objectives 19-2
The Validation Process 19-3
Controlling Validation Using Properties: Validation Unit 19-5
Controlling Validation Using Properties: Validate from List 19-7
Controlling Validation Using Triggers 19-9
Example: Validating User Input 19-11
Using Client-Side Validation 19-13
Tracking Validation Status 19-16
Controlling When Validation Occurs with Built-Ins 19-18
Summary 19-20
Practice 19 Overview 19-22

20 Navigation

Objectives 20-2
Navigation Overview 20-3
Understanding Internal Navigation 20-5
Using Object Properties to Control Navigation 20-7
Mouse Navigate Property 20-9
Writing Navigation Triggers 20-10
Navigation Triggers 20-11
When-New-*<object>*-Instance Triggers 20-12
SET_*<object>*_PROPERTY Examples 20-13
The Pre- and Post-Triggers 20-15
Post-Block Trigger Example 20-17
The Navigation Trap 20-18
Using Navigation Built-Ins in Triggers 20-19
Summary 20-21
Practice 20 Overview 20-23

21 Transaction Processing

Objectives 21-2
Transaction Processing Overview 21-3
The Commit Sequence of Events 21-6
Characteristics of Commit Triggers 21-8
Common Uses for Commit Triggers 21-10
Life of an Update 21-12
Delete Validation 21-14

- Assigning Sequence Numbers 21-16
- Keeping an Audit Trail 21-18
- Testing the Results of Trigger DML 21-19
- DML Statements Issued During Commit Processing 21-21
- Overriding Default Transaction Processing 21-23
- Running Against Data Sources Other than Oracle 21-25
- Getting and Setting the Commit Status 21-27
- Array DML 21-31
- Effect of Array DML on Transactional Triggers 21-32
- Implementing Array DML 21-33
- Summary 21-34
- Practice 21 Overview 21-38

22 Writing Flexible Code

- Objectives 22-2
- What Is Flexible Code? 22-3
- Using System Variables for Current Context 22-4
- System Status Variables 22-6
- GET_<object>_PROPERTY Built-Ins 22-7
- SET_<object>_PROPERTY Built-Ins 22-9
- Referencing Objects by Internal ID 22-11
- FIND_ Built-Ins 22-12
- Using Object IDs 22-13
- Increasing the Scope of Object IDs 22-15
- Referencing Objects Indirectly 22-17
- Summary 22-20
- Practice 22 Overview 22-22

23 Sharing Objects and Code

- Objectives 23-2
- Benefits of Reusing Objects and Code 23-3
- What Are Property Classes? 23-5
- Creating a Property Class 23-6
- Inheriting from a Property Class 23-8
- What Are Object Groups? 23-10
- Creating and Using Object Groups 23-11
- Copying and Subclassing Objects and Code 23-13
- Subclassing 23-14
- What Are Object Libraries? 23-16
- Benefits of the Object Library 23-18
- Working with Object Libraries 23-19

- What Is a SmartClass? 23-20
- Working with SmartClasses 23-21
- Reusing PL/SQL 23-22
- What Are PL/SQL Libraries? 23-24
- Writing Code for Libraries 23-25
- Creating Library Program Units 23-26
- Attach Library Dialog Box 23-27
- Calls and Searches 23-28
- Summary 23-30
- Practice 23 Overview 23-32

24 Using WebUtil to Interact with the Client

- Objectives 24-2
- WebUtil Overview 24-3
- Benefits of the WebUtil Utility 24-4
- Integrating WebUtil into a Form 24-11
- When to Use WebUtil Functionality 24-13
- Interacting with the Client 24-14
- Example: Opening a File Dialog on the Client 24-15
- Example: Reading an Image File into Forms from the Client 24-16
- Example: Writing Text Files on the Client 24-17
- Example: Executing Operating System Commands on the Client 24-18
- Example: Performing OLE Automation on the Client 24-19
- Example: Obtaining Environment Information about the Client 24-22
- Summary 24-23
- Practice 24 Overview 24-24

25 Introducing Multiple Form Applications

- Objectives 25-2
- Multiple Form Applications Overview 25-3
- Multiple Form Session 25-4
- Benefits of Multiple Form Applications 25-5
- Starting Another Form Module 25-6
- Defining Multiple Form Functionality 25-8
- Conditional Opening 25-10
- Closing the Session 25-11
- Closing a Form with `EXIT_FORM` 25-12
- Other Useful Triggers 25-13
- Sharing Data Among Modules 25-15
- Linking by Global Variables 25-16
- Global Variables: Opening Another Form 25-17
- Global Variables: Restricted Query at Startup 25-18

Assigning Global Variables in the Opened Form 25-19
Linking by Parameter Lists 25-20
Linking by Global Record Groups 25-23
Linking by Shared PL/SQL Variables 25-24
Summary 25-26
Practice 25 Overview 25-28

Appendix A: Practice Solutions

Appendix B: Table Descriptions

Appendix C: Introduction to Query Builder

Appendix D: Locking in Forms

Appendix E: Oracle Object Features

Appendix F: Using the Layout Editor

Preface

Profile

Before you begin this course

Before you begin this course, you should be able to:

- Create SQL statements.
- Create PL/SQL constructs, including conditional statements, loops, procedures and functions.
- Create PL/SQL stored (server) procedures, functions, and packages.
- Use a graphical user interface (GUI).
- Use a Web browser.

Prerequisites

Either

- Oracle Database 10g: SQL Fundamentals I
- or the following CBT Library:
 - Oracle SQL: Basic SELECT statements
 - Oracle SQL: Data Retrieval Techniques
 - Oracle SQL: DML and DDL
- or Introduction to Oracle Database 10g for Experienced SQL Users (InClass)
- or Oracle Database 10g: Introduction to SQL (InClass)

And either

- Oracle Database 10g: Program with PL/SQL (InClass)
- or the following CBT Library:
 - PL/SQL: Basics
 - PL/SQL: Procedures, Functions, and Packages
 - PL/SQL: Database Programming
- Or both:
 - Oracle Database 10g: PL/SQL Fundamentals (InClass)
 - Oracle Database 10g: Develop PL/SQL Program Units (InClass)

Suggested prerequisites

- Oracle Database 10g: SQL Fundamentals II (InClass) (if you attended the Oracle Database 10g: SQL Fundamentals I (InClass))
- Oracle Database 10g: Advanced PL/SQL (InClass)
- Oracle Forms Developer 10g: Move to the Web (eStudy)

How this course is organized

Oracle Forms Developer 10g: Build Internet Applications is an instructor-led course featuring lecture and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills introduced.

Related Publications

Oracle publications

Title	Part Number
<i>Oracle Forms Developer, Release 6i: Getting Started (Windows 95/NT)</i>	A73154-01
<i>Oracle Forms Developer and Reports Developer, Release 6i: Guidelines for Building Applications</i>	A73073-02
<i>Oracle Application Server Forms Services Deployment Guide 10g (9.0.4)</i>	B10470-01

Additional publications

Release notes: <ORACLE_HOME\doc\welcome\release_notes\chap_forms.htm

Typographic Conventions

Typographic conventions in text

Convention	Element	Example
Bold italic	Glossary term (if there is a glossary)	The <i>algorithm</i> inserts the new key.
Caps and lowercase	Buttons, check boxes, triggers, windows	Click the Executable button. Select the Can't Delete Card check box. Assign a When-Validate-Item trigger to the ORDERS block. Open the Master Schedule window.
Courier new, case sensitive (default is lowercase)	Code output, directory names, filenames, passwords, pathnames, URLs, user input, usernames	Code output: <code>debug.set ('I', 300);</code> Directory: bin (DOS), \$FMHOME (UNIX) Filename: Locate the <code>init.ora</code> file. Password: User <code>tiger</code> as your password. Pathname: Open <code>c:\my_docs\projects</code> URL: Go to <code>http://www.oracle.com</code> User input: Enter <code>300</code> Username: Log on as <code>scott</code>
Initial cap	Graphics labels (unless the term is a proper noun)	Customer address (<i>but</i> Oracle Payables)
Italic	Emphasized words and phrases, titles of books and courses, variables	Do <i>not</i> save changes to the database. For further information, see <i>Oracle7 Server SQL Language Reference Manual</i> . Enter <code>user_id@us.oracle.com</code> , where <i>user_id</i> is the name of the user.
Quotation marks	Interface elements with long names that have only initial caps; lesson and chapter titles in cross-references	Select "Include a reusable module component" and click Finish. This subject is covered in Unit II, Lesson 3, "Working with Objects."
Uppercase	SQL column names, commands, functions, schemas, table names	Use the <code>SELECT</code> command to view information stored in the <code>LAST_NAME</code> column of the <code>EMP</code> table.

Typographic Conventions (continued)

Typographic conventions in text (continued)

Convention	Element	Example
Right arrow	Menu paths	Select File > Save.
Brackets	Key names	Press [Enter].
Commas	Key sequences	Press and release keys one at a time: [Alternate], [F], [D]
Plus signs	Key combinations	Press and hold these keys simultaneously: [Ctrl]+[Alt]+[Del]

Typographic conventions in code

Convention	Element	Example
Caps and lowercase	Oracle Forms triggers	When-Validate-Item
Lowercase	Column names, table names	SELECT last_name FROM s_emp;
	Passwords	DROP USER scott IDENTIFIED BY tiger;
	PL/SQL objects	OG_ACTIVATE_LAYER (OG_GET_LAYER ('prod_pie_layer'))
Lowercase italic	Syntax variables	CREATE ROLE <i>role</i>
Uppercase	SQL commands and functions	SELECT userid FROM emp;

I Introduction

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	15 minutes	Lecture
	15 minutes	Total

Objectives

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

- **Identify the course objectives**
- **Identify the course content and structure**

ORACLE

I-2

Copyright © 2004, Oracle. All rights reserved.

Introduction

Overview

This lesson introduces you to the *Oracle Forms Developer 10g: Build Internet Applications* course:

- The objectives that the course intends to meet
- The topics that it covers
- How the topics are structured over the duration of the course

Course Objectives

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

- **Create form modules including components for database interaction and GUI controls.**
- **Display form modules in multiple windows and a variety of layout styles.**
- **Test form modules in a Web browser.**
- **Debug form modules in a three-tier environment.**

ORACLE

I-3

Copyright © 2004, Oracle. All rights reserved.

Course Objectives

Course Description

In this course, you will learn to build, test, and deploy interactive Internet applications. Working in a graphical user interface (GUI) environment, you will learn how to create and customize forms with user input items such as check boxes, list items, and radio groups. You will also learn how to modify data access by creating event-related triggers, and you will display Forms elements and data in multiple canvases and windows.

Course Objectives

- **Implement triggers to:**
 - Enhance functionality
 - Communicate with users
 - Supplement validation
 - Control navigation
 - Modify default transaction processing
 - Control user interaction
- **Reuse objects and code**
- **Link one form module to another**

ORACLE

Course Content

Day 1

- **Lesson 1: Introduction to Oracle Forms Developer and Oracle Forms Services**
- **Lesson 2: Running a Forms Builder Application**
- **Lesson 3: Working in the Forms Developer Environment**
- **Lesson 4: Creating a Basic Form Module**
- **Lesson 5: Creating a Master-Detail Form**
- **Lesson 6: Working with Data Blocks and Frames**

ORACLE

I-5

Copyright © 2004, Oracle. All rights reserved.

Course Content

The lesson titles show the topics that is covered in this course, and the usual sequence of lessons. However, the daily schedule is an estimate, and may vary for each class.

Course Content

Day 2

- **Lesson 7: Working with Text Items**
- **Lesson 8: Creating LOVs and Editors**
- **Lesson 9: Creating Additional Input Items**
- **Lesson 10: Creating Noninput Items**

ORACLE

Course Content

Day 3

- **Lesson 11: Creating Windows and Content Canvases**
- **Lesson 12: Working with Other Canvas Types**
- **Lesson 13: Introduction to Triggers**
- **Lesson 14: Producing Triggers**
- **Lesson 15: Debugging Triggers**

ORACLE

Course Content

Day 4

- **Lesson 16: Adding Functionality to Items**
- **Lesson 17: Run-time Messages and Alerts**
- **Lesson 18: Query Triggers**
- **Lesson 19: Validation**
- **Lesson 20: Navigation**

ORACLE

Course Content

Day 5

- **Lesson 21: Transaction Processing**
- **Lesson 22: Writing Flexible Code**
- **Lesson 23: Sharing Objects and Code**
- **Lesson 24: Using WebUtil to Interact with the Client**
- **Lesson 25: Introducing Multiple Form Applications**

ORACLE

13

Introduction to Triggers

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	15 minutes	Lecture
	15 minutes	Total

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**

ORACLE

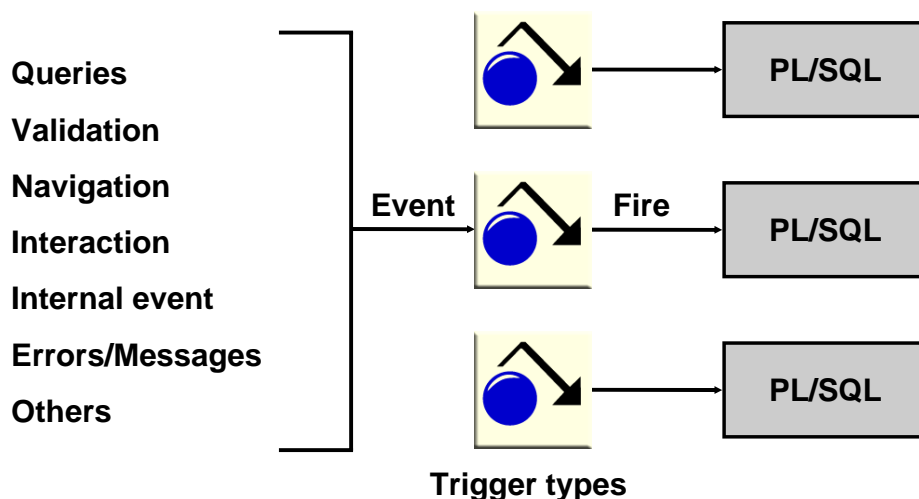
13-2

Copyright © 2004, Oracle. All rights reserved.

Objectives

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.

Trigger Overview



Which trigger would you use to perform complex calculations after a user enters data into an item?

ORACLE

13-3

Copyright © 2004, Oracle. All rights reserved.

Trigger Overview

A trigger is a program unit that is executed (fired) due to an event. As explained earlier, Forms 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.

You write Forms Builder triggers in PL/SQL. Every trigger that you define is associated with a specific event. Forms 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 or physical navigation
- Operator interaction with items in the form
- Internal events in the form
- Errors and messages

Events cause the activation, or firing, of certain trigger types. The next page shows the categories of triggers and when they fire. Which trigger would you use to perform complex calculations after a user enters data into an item?

Grouping Triggers into Categories

Triggers may be grouped into functional categories:

- Block processing triggers
- Interface event triggers
- Master-detail triggers
- Message handling triggers
- Navigational triggers
- Query-time triggers
- Transactional triggers
- Validation triggers

Triggers may be grouped into categories based on name:

- When-Event triggers
- On-Event triggers
- Pre-Event triggers
- Post-Event triggers
- Key triggers

ORACLE

Trigger Categories

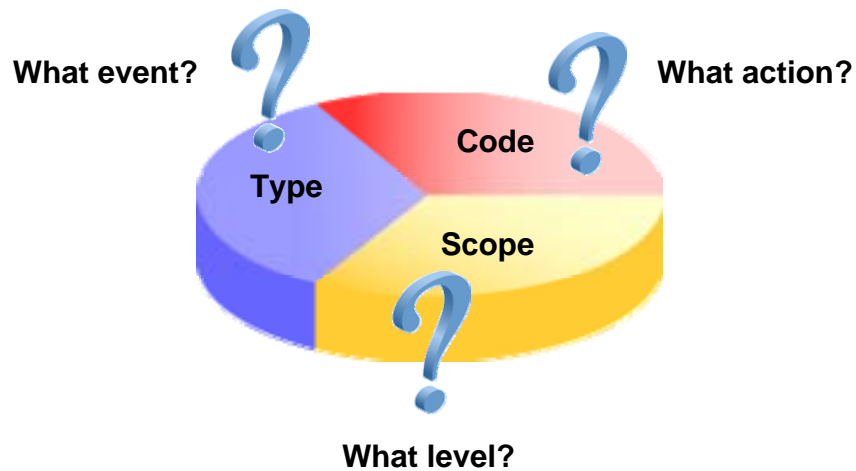
Triggers may be categorized based on their functions:

Category	Fires	Examples
Block processing	In response to events related to record management in a block.	When-Create-Record When-Clear-Block When-Database-Record When-Remove-Record
Interface event	In response to events that occur in the form interface	Key-[all] When-Button-Pressed When-[Checkbox List Radio]-Changed When-Image-[Activated Pressed] When-Timer-Expired When-Window-[Activated Deactivated Closed Resized]
Master-detail	To enforce coordination between records in a detail block and the master record in a master block.	On-Check-Delete-Master On-Clear-Details On-Populate-Details
Message-handling	In response to default messaging events	On-Error On-Message
Navigational	In response to navigational events	Pre-[Form Block Record Text-Item] Post-[Form Block Record Text-Item] When-New-[Form Block Record Item]-Instance
Query-time	Just before and just after the operator or the application executes a query in a block	Pre-Query Post-Query
Validation	When Forms validates data after the user enters data and navigates out of the item or record	When-Validate-[Item Record]

You can also categorize triggers based on their names:

Category	Description
When-Event	Point at which Forms default processing may be <i>augmented</i> with additional tasks or operations
On-Event	Point at which Forms default processing may be <i>replaced</i>
Pre-Event	Point just prior to the occurrence of either a When-event or an On-event; use to prepare objects or data for the upcoming event
Post-Event	Point just following the occurrence of either a When-event or an On-event; use to validate or perform auditing tasks based on the prior event
Key Triggers	Fires when the operator presses a specific key or key-sequence.

Defining Trigger Components



ORACLE

13-6

Copyright © 2004, Oracle. All rights reserved.

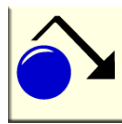
Trigger Components

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

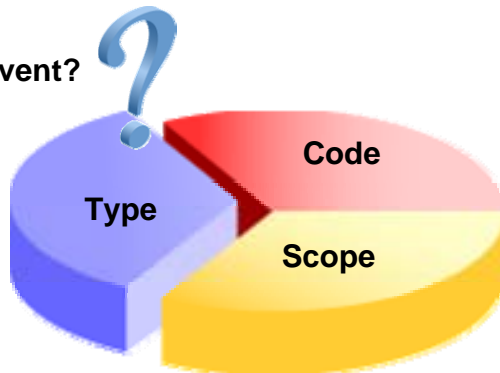
- **Trigger type:** Defines the specific event that will cause the trigger to fire
- **Trigger code:** The body of PL/SQL that defines the actions of the trigger
- **Trigger scope:** The level in a form module at which the trigger is defined—determining the scope of events that will be detected by the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

Trigger Type

- Pre-
- Post-
- When-
- On-
- Key-
- User-named



What event?



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. 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.

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 Forms validates an item.
- Pre-Query fires before Forms issues a query for a block.

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

Forms Builder supports user-named triggers as well as the standard built-in triggers. A user-named trigger is one that is named by the designer. These triggers fire only if called upon by another trigger or program unit using built-in code features.

Trigger Type

(User-named) KEY-CLRBLK KEY-CLRFRM KEY-CLRREC KEY-COMMIT KEY-CQUERY KEY-CREREC KEY-DELREC KEY-DOWN KEY-DUP-ITEM KEY-DUPREC KEY-EDIT KEY-ENTER KEY-ENTQRY KEY-EXEQRY KEY-EXIT KEY-F0 KEY-F1 KEY-F2 KEY-F3 KEY-F4 KEY-F5 KEY-F6 KEY-F7 KEY-F8 KEY-F9 KEY-HELP KEY-LISTVAL KEY-MENU KEY-NEXT-ITEM KEY-NXTBLK	KEY-NXTKEY KEY-NXTREC KEY-NXTSET KEY-OTHERS KEY-PREV-ITEM KEY-PRINT KEY-PRVBLK KEY-PRVREC KEY-SCRDOWN KEY-SCRUP KEY-UP KEY-UPREC ON-CHECK-DELETE-MASTER ON-CHECK-UNIQUE ON-CLOSE ON-COLUMN-SECURITY ON-COMMIT ON-COUNT ON-DELETE ON-FETCH ON-INSERT ON-LOCK ON-LOGON ON-LOGOUT ON-MESSAGE ON-POPULATE-DETAILS ON-ROLLBACK ON-SAVEPOINT ON-SELECT ON-SEQUENCE-NUMBER	ON-UPDATE POST-BLOCK POST-CHANGE POST-DATABASE-COMMIT POST-DELETE POST-FORM POST-FORMS-COMMIT POST-INSERT POST-LOGON POST-LOGOUT POST-QUERY POST-RECORD POST-SELECT POST-TEXT-ITEM POST-UPDATE PRE-BLOCK PRE-COMMIT PRE-DELETE PRE-INSERT PRE-LOGON PRE-LOGOUT PRE-POPUP-MENU PRE-QUERY PRE-RECORD PRE-SELECT PRE-TEXT-ITEM PRE-UPDATE WHEN-BUTTON-PRESSED WHEN-CHECKBOX-CHANGED WHEN-CLEAR-BLOCK	WHEN-CREATE-RECORD WHEN-CUSTOM-ITEM-EVENT WHEN-DATABASE-RECORD WHEN-FORM-NAVIGATE WHEN-IMAGE-ACTIVATED WHEN-IMAGE-PRESSED WHEN-LIST-ACTIVATED WHEN-LIST-CHANGED WHEN-MOUSE-CLICK WHEN-MOUSE-DOUBLECLICK WHEN-MOUSE-DOWN WHEN-MOUSE-ENTER WHEN-MOUSE-LEAVE WHEN-MOUSE-MOVE WHEN-MOUSE-UP WHEN-NEW-BLOCK-INSTANCE WHEN-NEW-ITEM-INSTANCE WHEN-NEW-RECORD-INSTANCE WHEN-RADIO-CHANGED WHEN-REMOVE-RECORD WHEN-TAB-PAGE-CHANGED WHEN-TIMER-EXPIRED WHEN-TREE-NODE-ACTIVATED WHEN-TREE-NODE-EXPANDED WHEN-TREE-NODE-SELECTED WHEN-VALIDATE-ITEM WHEN-VALIDATE-RECORD WHEN-WINDOW-ACTIVATED WHEN-WINDOW-CLOSED WHEN-WINDOW-DEACTIVATED WHEN-WINDOW-RESIZED
---	--	--	--

Forms Builder Trigger Types

ORACLE

13-8

Copyright © 2004, Oracle. All rights reserved.

Trigger Type (continued)

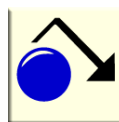
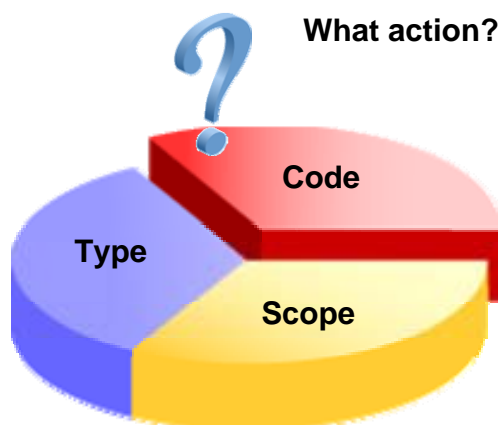
The first part of a trigger name (before the first hyphen) follows a standard convention; this helps you to understand the general nature of the trigger type, and plan the types to use.

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 just before the action named in the trigger type (for example, before a query is executed)
Post-	Fires just after the action named in the trigger type (for example, after a query is executed)
When-	Fires in addition to standard processing (used to augment functionality)

Instructor Note

When-Mouse-Move/Enter/Leave triggers, although still in existence, are ignored when running on the Web due to the amount of network traffic that would be generated.

Trigger Code



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

ORACLE

13-9

Copyright © 2004, Oracle. All rights reserved.

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

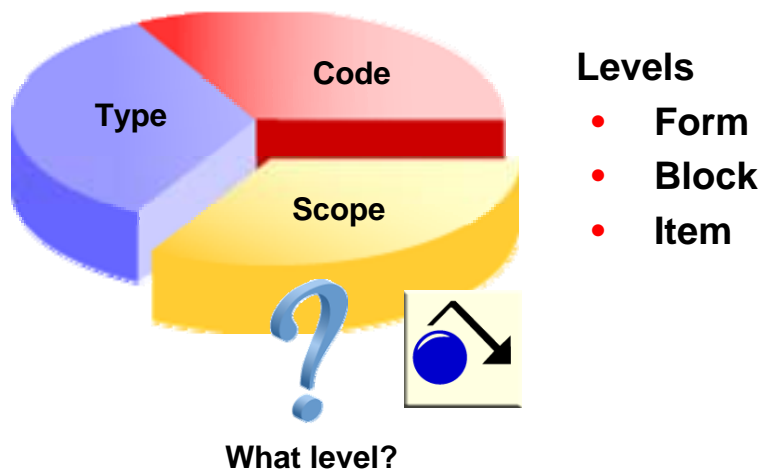
Statements that you write in a trigger can be constructed as follows:

- 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
- Calls to user-named subprograms (procedures and functions) in the form, a library, or the database
- Calls to built-in subprograms and package subprograms

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

- `INSERT`, `UPDATE`, and `DELETE` statements can be placed in transactional triggers. These triggers fire during the commit process.
- Transaction control statements (`COMMIT`, `ROLLBACK`, `SAVEPOINT`) should not be included directly as SQL trigger statements. These actions are carried out automatically by Forms as a result of either commands or built-in procedures that you issue. If included in triggers, these commands are redirected to be handled by Forms. For example, `COMMIT` will issue a `COMMIT_FORM`.

Trigger Scope



ORACLE

13-10

Copyright © 2004, Oracle. All rights reserved.

Trigger Scope

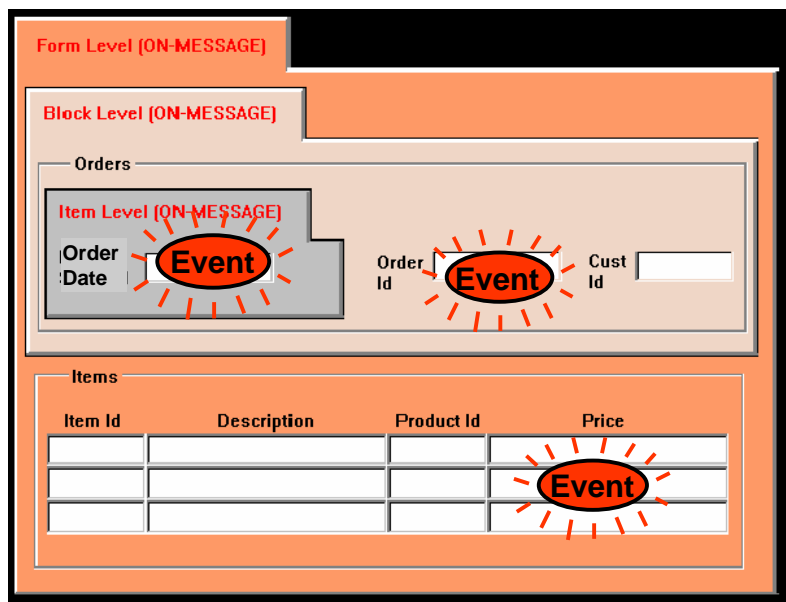
The scope of a trigger is determined by its position in the form object hierarchy, that is, the type of object under which you create the trigger. There are three possible levels:

- **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 fire only when this block is the current block.
- **Item level:** The trigger belongs to an individual item and can fire only 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.

Trigger Scope



ORACLE

13-11

Copyright © 2004, Oracle. All rights reserved.

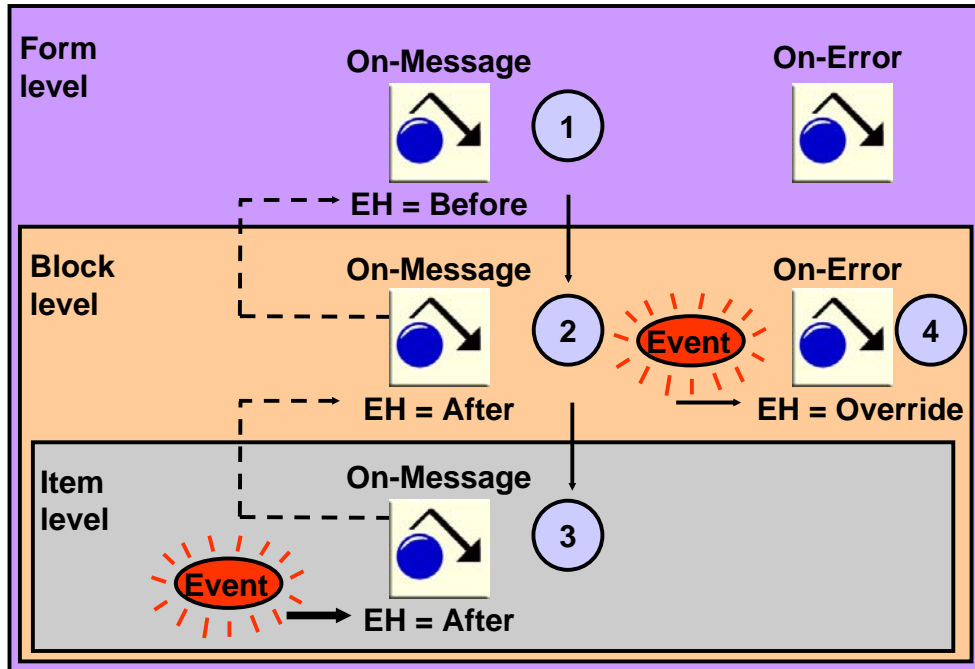
Trigger Scope (continued)

Consider the example in the diagram above:

- When the cursor is in the `Order_Date` item, a message fires the On-Message trigger of the `Order_Date` 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 Forms displays a message. It is recommended that you code On-Message triggers only at the form level.

Specifying Execution Hierarchy



ORACLE

13-12

Copyright © 2004, Oracle. All rights reserved.

Execution Hierarchy

As already stated, when there is more than one trigger of the same type, Forms 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.

The diagram above shows how setting EH affects the firing order of triggers:

1. Fires first
2. Fires second
3. Fires third
4. Fires independently

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

Execution Hierarchy (continued)

Execution hierarchy (EH) is a trigger property that specifies how the current trigger code should execute if there is a trigger with the same name defined at a higher level in the object hierarchy. Setting EH for form level triggers has no effect, since there is no higher level trigger.

Settings for execution hierarchy are:

- **Override:** Only the trigger most specific to the cursor location fires. This is the default.
- **After:** The trigger fires *after* firing the same trigger, if any, at the next highest level.
- **Before:** The trigger fires *before* firing the same trigger, if any, at the next highest level.

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.

Instructor Note

Demonstration

Open the form `EH_DEMO.fmb`, which is based on the `CUSTOMERS` table. Point out that there are On-Error triggers defined at form level, block level, and item level (for `CUSTOMERS.customer_id`). The triggers just display a message indicating the level of the trigger that fires. Run the form and insert text into `CUSTOMERS.customer_id`, then navigate to the next item. This causes the On-Error trigger to fire only for the item level.

Set EH property to “before” for block level and to “after” for item level. Run the form and cause the error as before. This causes the On-Error triggers to fire according to the EH settings: first block level, then form level, and finally item level.

Point out to students that setting EH at form level would not affect the execution order. Also inform them that it is recommended to define On-Error triggers only at form level, but it is used here just to illustrate trigger scope.

Summary

In this lesson, you should have learned that:

- **Triggers are event-activated program units**
- **You can categorize triggers based on function or name to help you understand how they work**
- **Trigger components are:**
 - **Type: Defines the event that fires the trigger**
 - **Code: The actions a trigger performs**
 - **Scope: Specifies the level (form, block, or item) at which the trigger is defined**
- **The Execution Hierarchy trigger property alters the firing sequence of a trigger**

ORACLE

13-14

Copyright © 2004, Oracle. All rights reserved.

Summary

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

- Triggers are event-activated program units.
- You can categorize triggers based on:
 - Function: Block-processing, Interface event, Master-detail, Message-handling, Navigational, Query-time, Transactional, Validation
 - Name: When-, On-, Pre-, Post-, Key-
- Trigger components include:
 - The trigger type that defines the event that fires the trigger
 - The trigger code that consists of a PL/SQL anonymous bloc
 - The trigger scope that 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 the Execution Hierarchy trigger property.

Instructor Note

[Ask questions to test understanding of the lesson.](#)

14

Producing Triggers

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	40 minutes	Lecture
	30 minutes	Practice
	70 minutes	Total

Objectives

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

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

ORACLE

14-2

Copyright © 2004, Oracle. All rights reserved.

Introduction

Overview

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

Creating Triggers in Forms Builder

To produce a trigger:

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

ORACLE

14-3

Copyright © 2004, Oracle. All rights reserved.

Defining Triggers in Forms Builder

Using Smart Triggers

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

Using the Trigger LOV

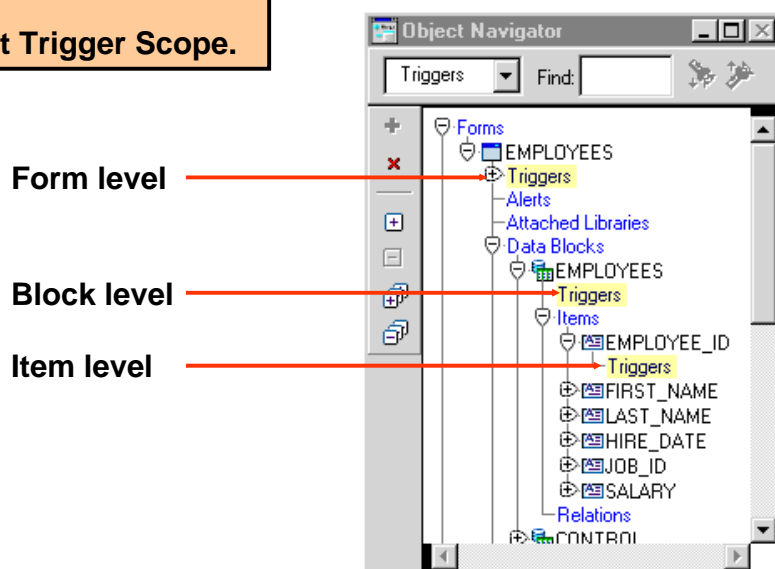
Using Smart Triggers is the easiest way to create a new trigger, but you can also do it by displaying the Trigger LOV, a list of all Forms Builder triggers. You can invoke the Trigger LOV from the Object Navigator, the Layout Editor, the PL/SQL Editor, or the Smart Triggers LOV.

How to Create a Trigger

Use the steps on the following pages to create a trigger.

Creating a Trigger

**Step One:
Select Trigger Scope.**



ORACLE

14-4

Copyright © 2004, Oracle. All rights reserved.

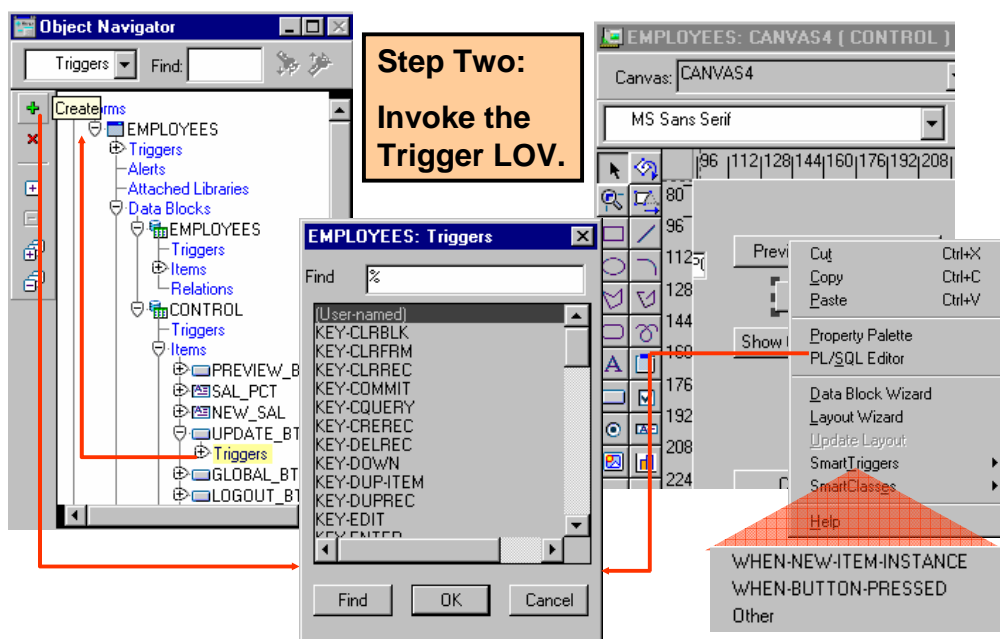
Creating a Trigger

Step One: Select Trigger Scope

In the Object Navigator, select the Triggers node of the form, block, or item that will own the trigger. Alternatively, you can select an item in the Layout Editor if you are defining an item level trigger.

A mistake often made by inexperienced Forms developers is to define a trigger at the wrong scope. For example, if you want a message to display when the cursor enters an item, you should code a When-New-Item-Instance trigger at the item level for that item. If you mistakenly put the trigger at the block or form level, the trigger will fire whenever the operator navigates to any item in the block or form.

Creating a Trigger



14-5

Copyright © 2004, Oracle. All rights reserved.

ORACLE

Creating a Trigger (continued)

Step Two: Invoke the Trigger LOV

Once the trigger scope is selected, there are several ways to invoke the Trigger LOV:

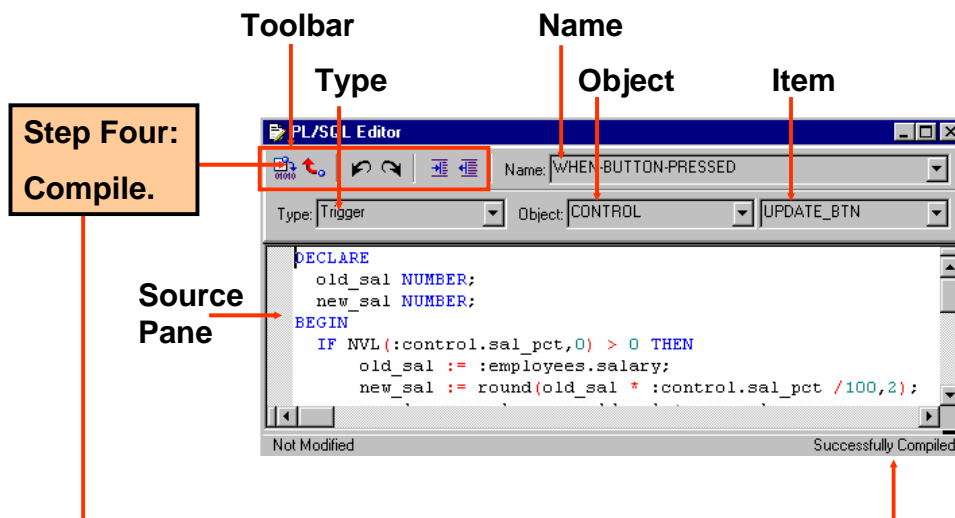
- **Smart Triggers:** Right-click to display the pop-up menu. Select Smart Triggers. This invokes the Smart Triggers LOV displaying a list of common triggers for the selected object. If you do not see the desired trigger, select Other to display the Trigger LOV with a full list of triggers.
- **Create:** If you are in the Object Navigator with a Triggers node highlighted, select Edit > Create from the menu, or click Create in the toolbar. This invokes the Trigger LOV.
- **PL/SQL Editor:** Right click to display the pop-up menu. Select PL/SQL Editor.
 - If there is no trigger defined for the object, this invokes the Trigger LOV.
 - If there are already triggers defined for the object, the name and code of the first one appear in the editor. To define an additional trigger for the item, click Create in the Object Navigator toolbar to invoke the Trigger LOV.

Once the Trigger LOV is invoked, select the trigger type.

Creating a Trigger

Step Three:

Use the PL/SQL Editor to define the trigger code.



Step Four:
Compile.

**Source
Pane**

ORACLE

14-6

Copyright © 2004, Oracle. All rights reserved.

Creating a Trigger (continued)

Step Three: Use the PL/SQL Editor to Define the Trigger Code

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.

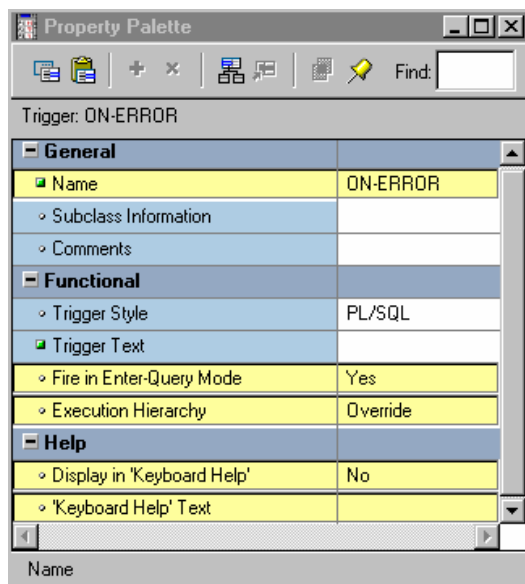
In Forms Builder, the PL/SQL Editor has the following specific trigger components:

- **Name:** Trigger name; pop-up list enables you to switch to a different trigger.
- **Type:** The type is set to Trigger. (The other types are Program Unit and Menu Item Code, but these are not valid for triggers.)
- **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. The Item trigger component is not labeled.
- **Source pane:** Where trigger code is entered or modified
- **Toolbar:** Buttons to compile, revert, undo, redo, indent, or outdent the code

Step Four: Compile

Click the Compile icon in the PL/SQL Editor to compile the trigger. This displays immediate feedback in the form of compilation error messages, which you can correct. If the trigger compiles correctly, a message displays in the lower right corner of the editor.

Setting Trigger Properties



ORACLE

14-7

Copyright © 2004, Oracle. All rights reserved.

Setting Trigger Properties

You can set the following trigger properties to affect the behavior of triggers:

General

- **Name:** Specifies the internal name of the trigger

Functional

- **Fire in Enter Query Mode:** Specify whether the trigger can fire when an event occurs in Enter Query as well as Normal mode.
- **Execution Hierarchy:** Use to change the default order of trigger firing when multiple triggers of the same name are defined at different levels.

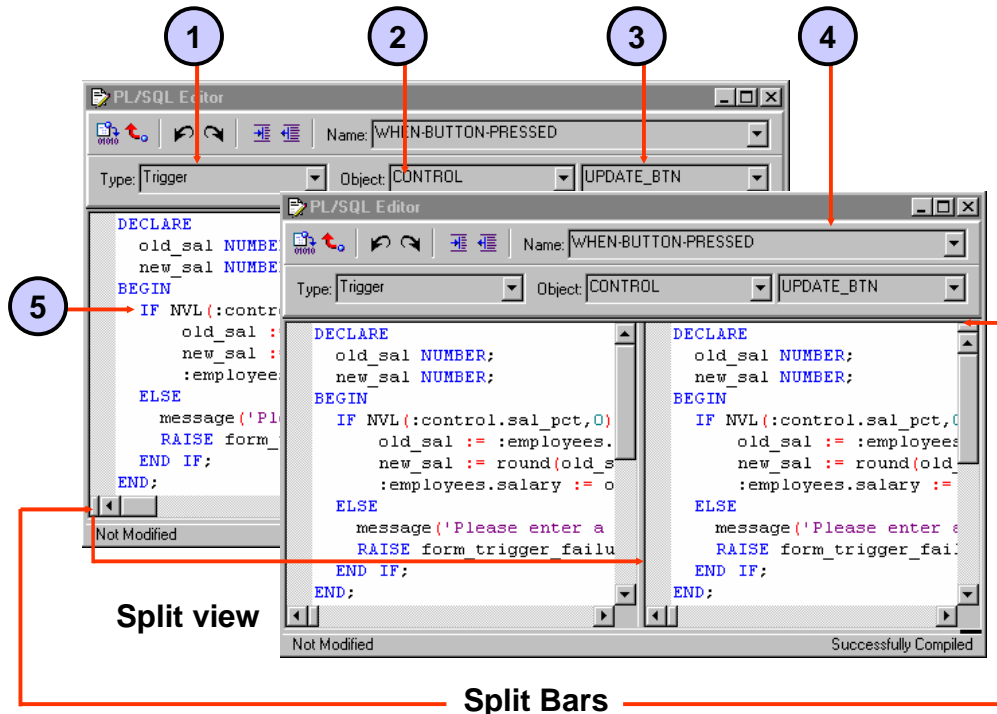
Help (these properties valid only for Key triggers)

- **Display in “Keyboard Help”:** Specify whether you want the name or description to appear in the Show Keys window.
- **“Keyboard Help” Text:** Description to replace the default key description.

Instructor Note

Execution hierarchy is explained in Lesson 12. Trigger Style is now a read-only property for which PL/SQL is the only valid value.

PL/SQL Editor Features



ORACLE

14-8

Copyright © 2004, Oracle. All rights reserved.

PL/SQL Editor Features

The PL/SQL Editor provides the following features:

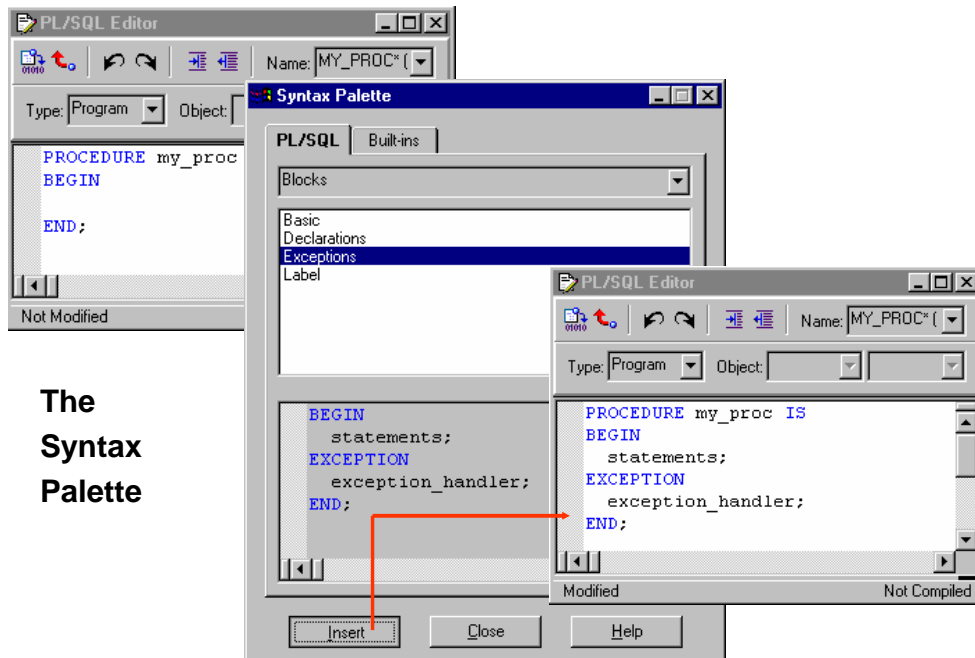
- Automatic Formatting and Coloring of PL/SQL Code
- Automatic Indenting and Color Syntax highlighting
- Drag and Drop text Manipulation
- Unlimited Undo/Redo
- Multiple Split Views

You can create up to four separate views of the current program unit in the PL/SQL Editor by using split bars. Place the cursor over the split bar; it changes to a double-headed arrow. Left-click and drag the split bar to the desired location. To remove the split, drag it back to its original location.

Trigger Components of the PL/SQL Editor

1. **Type:** Set to Trigger
2. **Object:** Enables you to set scope to Form Level or a specified block
3. **Item:** Enables you to switch between items (if item-level trigger) to access other triggers
4. **Name:** Trigger name; enables you to switch to another existing trigger
5. **Source Pane:** Where trigger code is entered or modified

PL/SQL Editor Features



The
Syntax
Palette

ORACLE

14-9

Copyright © 2004, Oracle. All rights reserved.

PL/SQL Editor Features (continued)

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 Tools > Syntax Palette from the menu system.

Global search and replace: Enables you to search for text across multiple program units without opening individual instances of the Program Unit Editor. Replace every occurrence of the search string or selected occurrences only.

Invoke the Find and Replace in Program Units dialog box by selecting Edit > Find and Replace PL/SQL from the menu system.

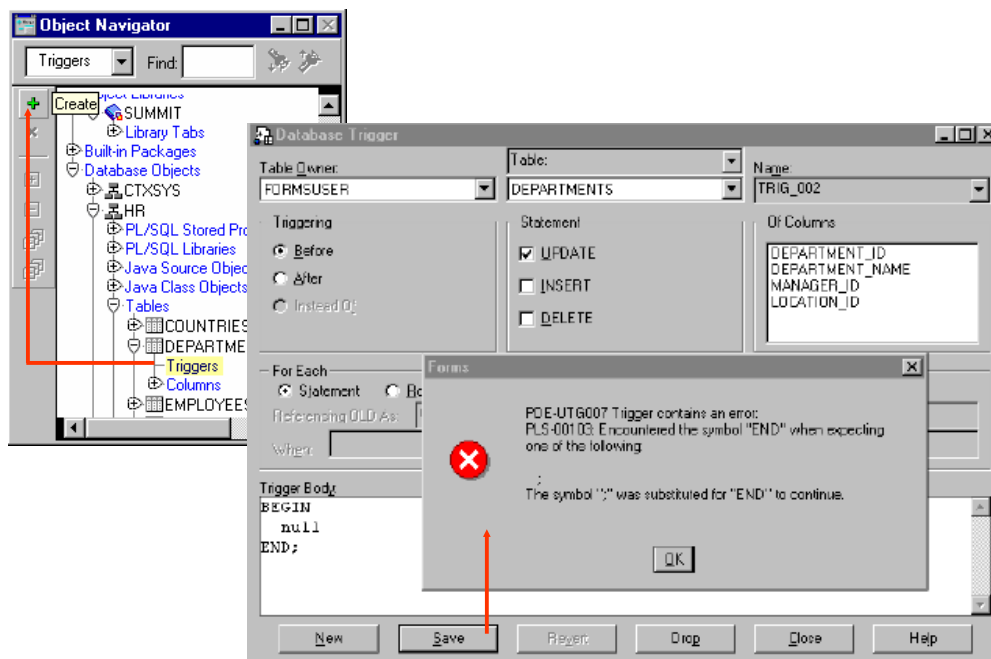
Things to Remember About the PL/SQL Editor

- New or changed text in triggers remains uncompiled until you click Compile.
- 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.

Instructor Note

Demonstrate both the Syntax Palette and the Global Search and Replace in Forms Builder.

The Database Trigger Editor



ORACLE

14-10

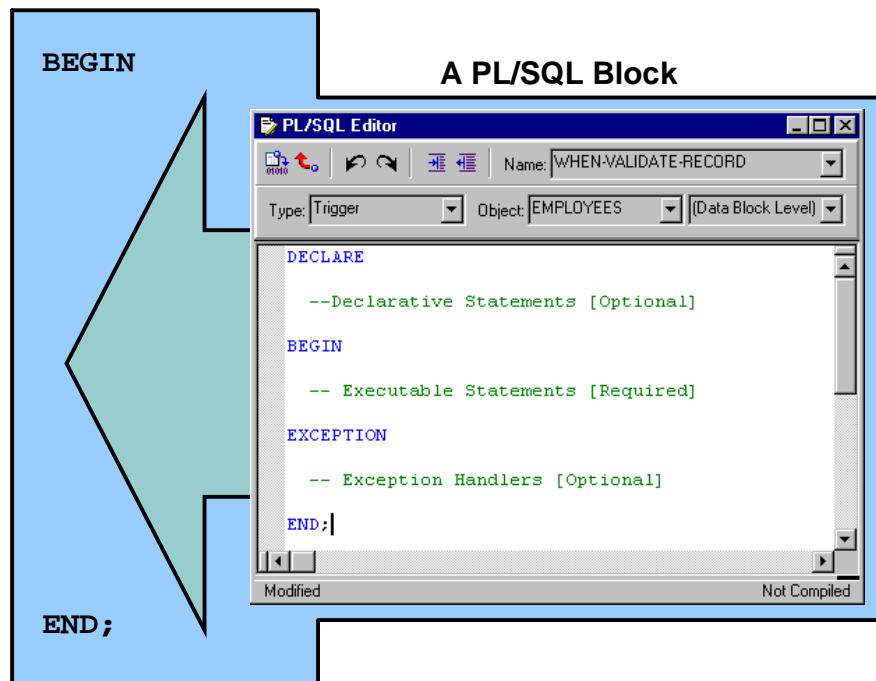
Copyright © 2004, Oracle. All rights reserved.

The 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 Edit > Create, or click Create on the toolbar. The Database Trigger Editor appears.
6. In the Database Trigger Editor, click New.
7. Define and save the desired program units.

Writing Trigger Code



Writing Trigger Code

The code text of a Forms 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.

Instructor Note

Explain the general structure of a trigger code.

The examples that follow are mainly aimed at showing possible structures; their detailed content is not very important at this stage.

Writing Trigger Code (continued)

Examples

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

```
IF :ORDER_ITEMS.unit_price IS NULL THEN
  :ORDER_ITEMS.unit_price := :PRODUCTS.list_price;
END IF;
calculate_total; -- User-named procedure
```

2. 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
(:ORDER_ITEMS.product_id,:ORDER_ITEMS.quantity);
MESSAGE('Discount: ' || TO_CHAR(vn_discount));
END;
```

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

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

Using Variables in Triggers

- **PL/SQL variables must be declared in a trigger or defined in a package**

```
PL/SQL Editor
Name: WHEN-VALIDATE-RECORD
Type: Trigger
Object: EMPLOYEES
(Data Block Level)

DECLARE
  my_var VARCHAR2(30);
BEGIN
  my_var := 'This is a PL/SQL variable';
  :block_name.item_name := 'This is a Forms Builder variable';
END;
```

- **Forms Builder variables**
 - Are not formally declared in PL/SQL
 - Need a colon (:) prefix in reference

ORACLE

14-13

Copyright © 2004, Oracle. All rights reserved.

Using Variables in Triggers

In triggers and subprograms, Forms 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.
- **Forms Builder variables:** Variable types maintained by the Forms 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). Forms Builder variables are not formally declared in a DECLARE section, and can exist outside the scope of a PL/SQL block.

Forms Builder Variables

Variable Type	Purpose	Syntax
Items	Presentation and user interaction	:block_name.item_name
Global variable	Session-wide character variable	:GLOBAL.variable_name
System variables	Form status and control	:SYSTEM.variable_name
Parameters	Passing values in and out of module	:PARAMETER.name

ORACLE

14-14

Copyright © 2004, Oracle. All rights reserved.

Forms Builder Variables

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

Instructor Note

Explain syntax of above variables with reference to examples on next page. What kind of object is OK_TO_LEAVE_BLOCK? Point out that more specific examples and how to manage these variables will be covered in later lessons.

Forms Builder Variables (continued)

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

Examples

1. References to items should be prefixed by the name of the owning Forms 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;
```

2. 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;
```

3. References to system variables must be prefixed by the word System, and the contents must be in uppercase ('NORMAL', not 'normal'):

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

4. Parameters defined at design-time have the prefix parameter:

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

Initializing Global Variables with Default Value

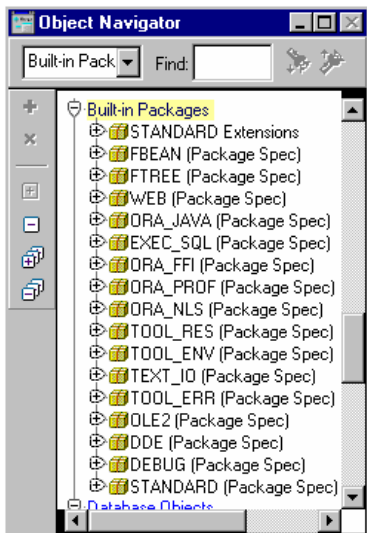
You can use the DEFAULT_VALUE built-in to assign a value to a global variable. Forms Builder creates the global variable if it does not exist. If the value of the indicated variable is not null, DEFAULT_VALUE does nothing. The following example creates a global variable named country and initializes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

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

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.

Adding Functionality with Built-In Subprograms



Built-ins belong to either:

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

ORACLE

14-16

Copyright © 2004, Oracle. All rights reserved.

Adding Functionality with Built-In Subprograms

Forms 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.

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

- **Standard Extensions package:** These built-ins are integrated into the Standard PL/SQL command set in Forms Builder. You can call them directly, without any package prefix. You can use more than one hundred standard built-ins.
Example: EXECUTE_QUERY ;
- **Other Forms 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.
Example: ORA_JAVA.CLEAR_EXCEPTION ;

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

Adding Functionality with Built-In Subprograms (continued)

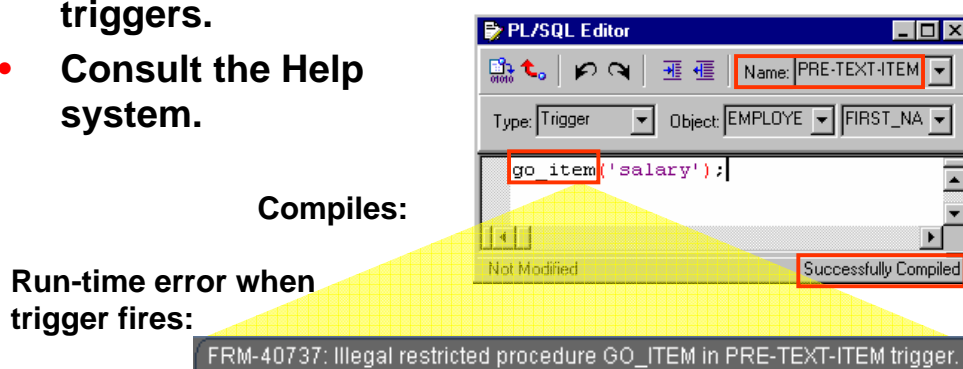
In addition to the standard extensions, Forms Builder provides the following packages:

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
FBEAN	Provides built-ins to interact with client-side Java beans
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_JAVA	Enables you to call Java procedures 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 to and from files
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
WEB	Provides built-ins for the Web environment

Note: Some of these packages, such as OLE2, ORA_FFI, and TEXT_IO, function on the application server side, not on the client side. For example, TOOL_ENV enables you to get and set environment variables on the application server. To interact with the client, you would need to provide similar functionality in a JavaBean.

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.**



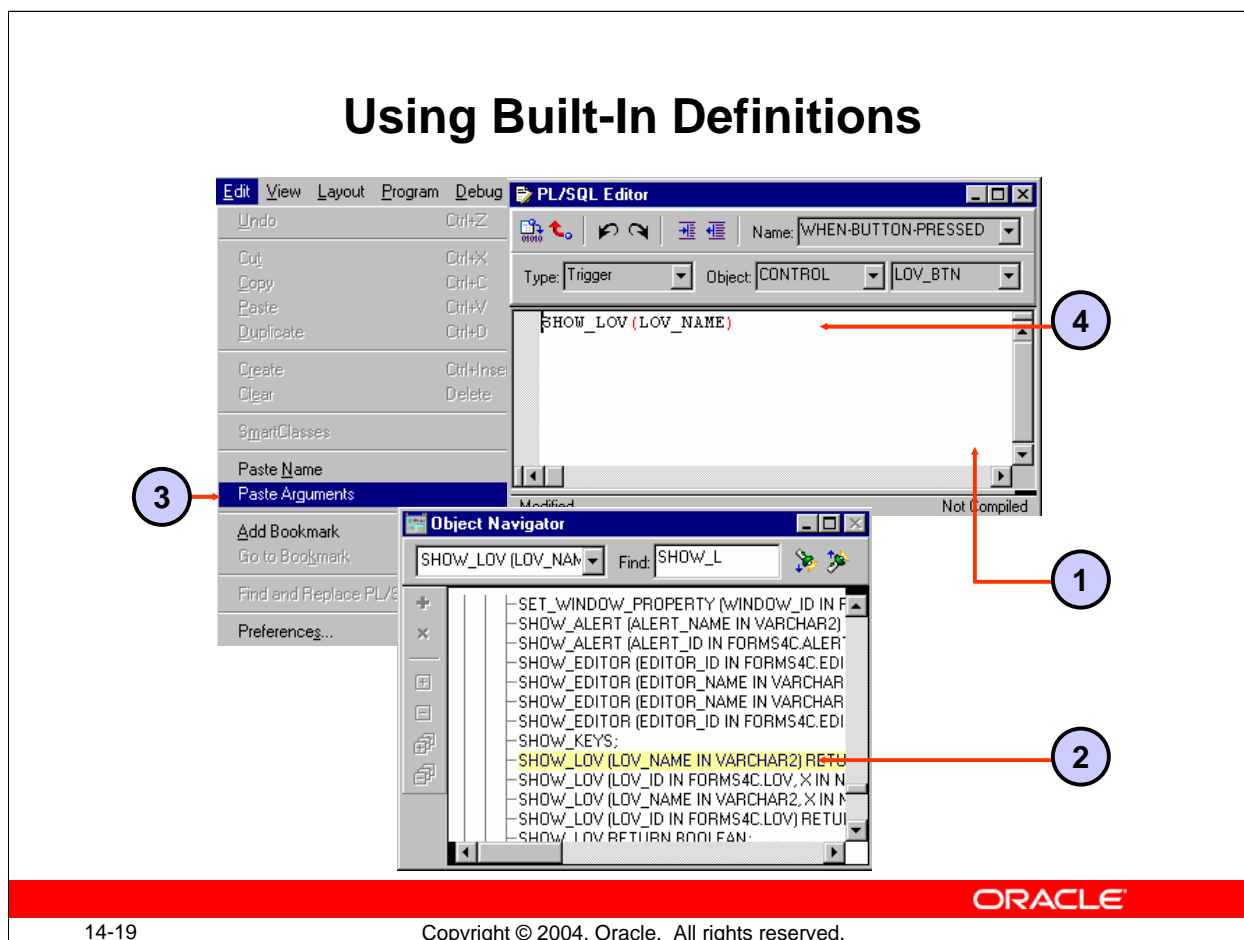
Limits of Use

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.

Calling a restricted built-in from a navigational trigger compiles successfully but causes a run-time error. For example, you can compile a Pre-Text-Item trigger that calls the restricted `GO_ITEM` built-in, but when the trigger fires at run-time it produces the FRM-40737 error.

Using Built-In Definitions



14-19

Copyright © 2004, Oracle. All rights reserved.

ORACLE

Using Built-In Definitions

When you are writing a trigger or a program unit, the Forms 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 Edit > Paste Name or Edit > Paste Arguments from the menus (Paste Arguments includes both the built-in name and its arguments).
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.

Using Built-In Definitions (continued)

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');
    . . .
```

Instructor Note

Point out the overloads in the list within the Object Navigator. You can use the find feature in the Object Navigator to bypass this overload in the list. Point out how you can locate the built-in in Help. Highlight an example using a built-in. Copy it, then close Help and paste in the PL/SQL Editor.

Demonstration: Use `SHOW_LOV` to demonstrate how you can paste prototype syntax for a built-in from the Object Navigator into a trigger or program unit.

Useful Built-Ins

- `EDIT_TEXTITEM`
- `ENTER_QUERY, EXECUTE_QUERY`
- `EXIT_FORM`
- `GET_ITEM_PROPERTY, SET_ITEM_PROPERTY`
- `GO_BLOCK, GO_ITEM`
- `MESSAGE`
- `SHOW_ALERT, SHOW_EDITOR, SHOW_LOV`
- `SHOW_VIEW, HIDE_VIEW`

ORACLE

14-21

Copyright © 2004, Oracle. All rights reserved.

Useful Built-Ins

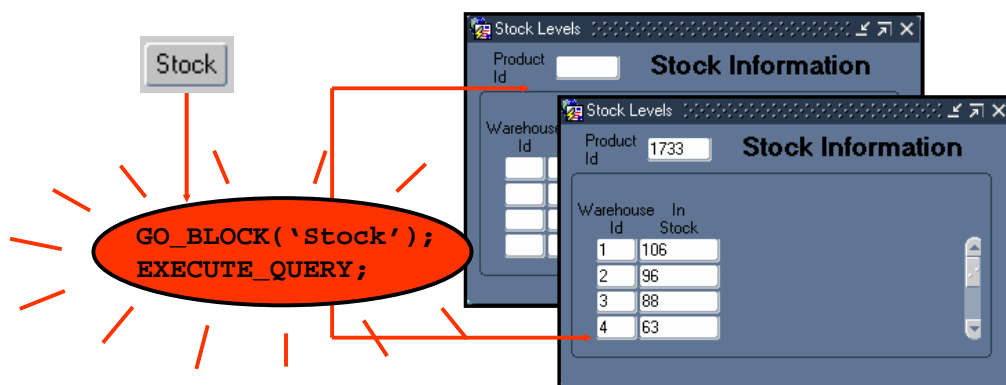
The table on the next page describes some built-ins that you can use in triggers to add functionality to items. They are discussed in later lessons.

Useful Built-Ins (continued)

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 Forms 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, Forms Builder prompts the operator to commit them before continuing EXECUTE_QUERY processing.
EXIT_FORM procedure	If in normal mode, exits current form; if in ENTER-QUERY mode, cancels query
GET_ITEM_PROPERTY function	Returns specified property values for the indicated 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 that indicates 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.

Using Triggers: 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



ORACLE

14-23

Copyright © 2004, Oracle. All rights reserved.

Using Triggers

When-Button-Pressed Trigger

This trigger fires when the user clicks 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

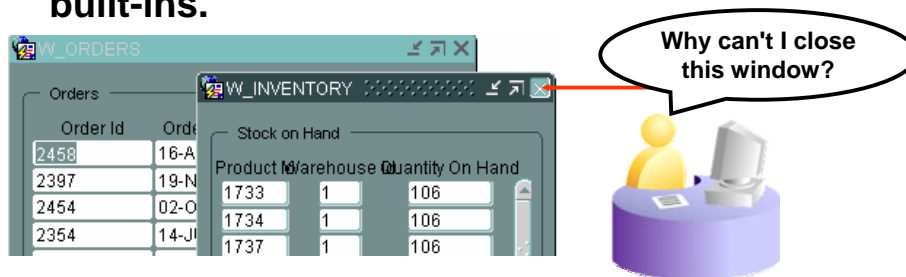
Example

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

```
GO_BLOCK('INVENTORIES');  
EXECUTE_QUERY;
```

Using Triggers: 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.



ORACLE

14-24

Copyright © 2004, Oracle. All rights reserved.

Using Triggers

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. Forms Builder does not close the window when the operator issues a window manager-specific close command; it only fires the 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 WIN_INVENTORY window by setting the VISIBLE property to FALSE.

```
GO_ITEM( 'ORDERS.ORDER_ID' );  
SET_WINDOW_PROPERTY( 'WIN_INVENTORY', VISIBLE, PROPERTY_FALSE );
```

Summary

In this lesson, you should have learned that:

- **You can use the PL/SQL Editor to write trigger code**
- **Trigger code has three sections:**
 - Declaration section (optional)
 - Executable statements section (required)
 - Exception handlers section (optional)
- **You can add functionality by calling built-in subprograms from triggers**
- **Restricted built-ins are not allowed in triggers that fire while navigation is occurring**

ORACLE

14-25

Copyright © 2004, Oracle. All rights reserved.

Summary

- To produce a trigger:
 - Select a scope in the Object Navigator.
 - Create a trigger and select a Name from the LOV, or use the SmartTriggers menu option.
 - Define code in the PL/SQL Editor.
 - Compile.
- 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.

Summary

- **The `When-Button-Pressed` trigger fires when the user presses a button**
- **The `When-Window-Closed` trigger fires when the user closes a window**

ORACLE

14-26

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- 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.

Practice 14 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 applications**
- **Using built-ins to display and hide the Help stack canvas**

ORACLE

14-27

Copyright © 2004, Oracle. All rights reserved.

Practice 14 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 the application
- Using built-ins to display and hide the Help stacked canvas

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

Practice 14

1. In the CUSTGXX form, write a trigger to display the Account_Mgr_Lov when the Account_Mgr_Lov_Button is selected. 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. Test to see that the LOV is invoked when you press the Account_Mgr_Lov_Button and that the form exits when you close the Customer Information window.
4. In the ORDGXX form, write a trigger to display the Products_Lov when the Product_Lov_Button is selected and that the form exits when you press the Exit_Button.
5. Write a trigger that exits the form when the Exit_Button is selected.
6. Save, compile, and run the form. Test to see that the LOV is invoked when you press the Product_Lov_Button.
7. Create a When-Button-Pressed trigger on CONTROL.Show_Help_Button that uses the SHOW_VIEW built-in to display the 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.
9. Create a When-Button-Pressed trigger on CONTROL.Stock_Button that uses the GO_BLOCK built-in to display the INVENTORIES block, and EXECUTE_QUERY to automatically execute a query.
10. Write a form-level When-Window-Closed trigger to hide the WIN_INVENTORY window if the user attempts to close it, and to exit the form if the user attempts to close the WIN_ORDER window.
Hint: Use the system variable :SYSTEM.TRIGGER_BLOCK to determine what block the cursor is in when the trigger fires.
11. Save and compile the form. Click Run Form to run the form and test the changes. The stacked canvas, CV_HELP, is displayed 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.

15

Debugging Triggers

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	40 minutes	Lecture
	25 minutes	Practice
	65 minutes	Total

Objectives

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

- **Describe the components of the Debug Console**
- **Use the Run Form Debug button to run a form module in debug mode**
- **Debug PL/SQL code**

ORACLE

15-2

Copyright © 2004, Oracle. All rights reserved.

Objectives

As you begin to add code to your form by writing triggers, you will probably find that you sometimes obtain incorrect or unexpected results. In a large application, it may be difficult to determine the source of the problem.

You can use Forms Builder's integrated PL/SQL Debugger to locate and correct coding errors. This lesson explains how to debug your PL/SQL code.

Instructor Note

The PL/SQL Debugger integrated into Forms Builder also enables remote debugging. You can attach the debugger to a form that is already running by supplying the host name and port number for that application. However, in this course we present information to enable developers to debug the triggers they are writing, so this lesson discusses using the debugger on a form run from Forms Builder.

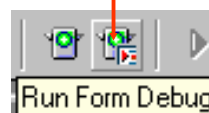
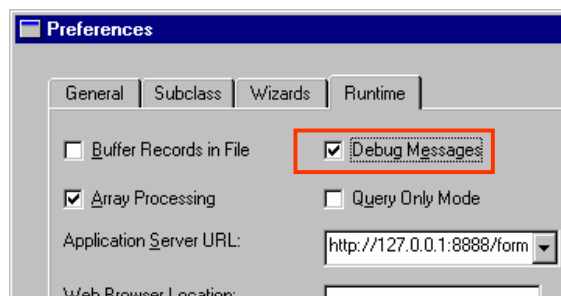
The Debugging Process

Monitor and debug triggers by:

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

```
DECLARE
  xyz VARCHAR2(30) := 'Hi there';
BEGIN
  message (xz);
END;
```

Error 201 at line 4, column 10
identifier 'xz' must be declared
Error 0 at line 4, column 2
Statement ignored



The Debugging Process

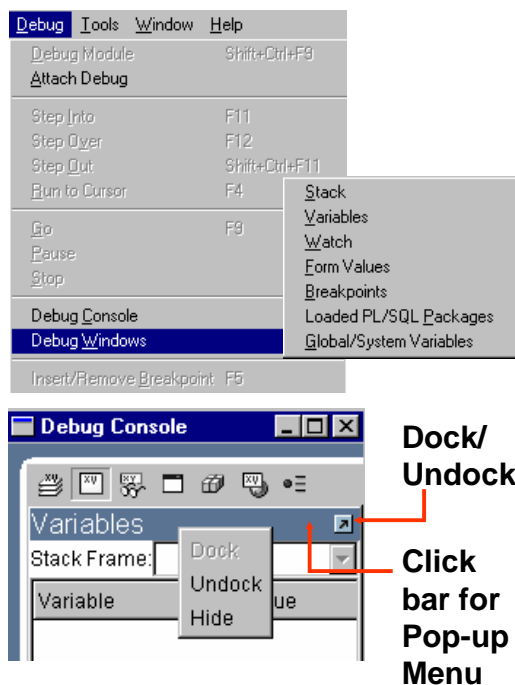
With Forms Builder you can monitor and debug triggers in several ways:

- **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 to see whether certain triggers are firing, their origin and level, and the time at which they fire. If an error occurs, you can look at the last trigger that fired to narrow the scope of the source of the error.
- **Invoking the PL/SQL Debugger:** You invoke the debugger from Forms Builder by clicking Run Form Debug on the toolbar.

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 modify variables as the form is running, which allows you to test how various changes in form item values and variable values will affect your application.

The Debug Console

- **Stack**
- **Variables**
- **Watch**
- **Form Values**
- **PL/SQL Packages**
- **Global and System Variables**
- **Breakpoints**



ORACLE

15-4

Copyright © 2004, Oracle. All rights reserved.

The Debug Console

The debug console is a workspace in Forms Builder that enables you to see various aspects of the running form and its environment. You display the debug console by selecting Debug > Debug Console from the menu. It displays automatically when you encounter a breakpoint in a form that is running in debug mode.

Within the console, you can display several types of panels: Stack, Variables, Watch, Form Values, Breakpoints, PL/SQL Packages, and Global and System Variables. You can resize the debug console and any of the panels displayed within it.

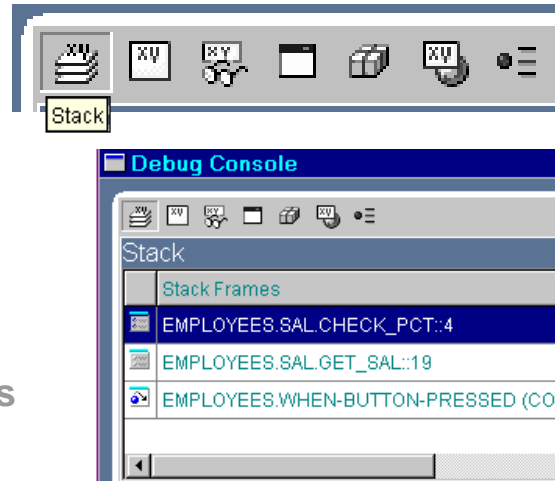
You choose which panels to display or close by clicking the toggle buttons on the toolbar of the Debug Console, or by selecting Debug > Debug Windows from the menu. As you show and hide panels within the console, the panels automatically expand and contract to fill the console.

You can undock any of the panels to display them outside the Debug Console by clicking the upward pointing arrow in the top right of the panel; you redock the panel by clicking its downward pointing arrow.

If you right click the area beside the dock/undock arrow, you will see a pop-up menu enabling you to hide, dock, or undock the panel.

The Debug Console: Stack Panel

- Stack
- Variables
- Watch
- Form Values
- PL/SQL Packages
- Global and System Variables
- Breakpoints



ORACLE

15-5

Copyright © 2004, Oracle. All rights reserved.

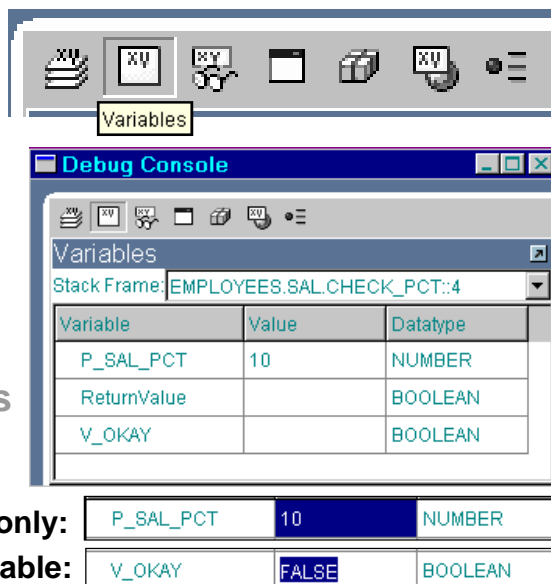
The Debug Console: Stack Panel

The call stack represents the chain of subprograms starting from the initial entry point down to the currently executing subprogram. The program currently executing in the example above is `EMPLOYEES.SAL.CHECK_PCT`, which was called by `EMPLOYEES.SAL.GET_SAL`. This program was called from a When-Button-Pressed trigger, the initial entry point of the current call stack. As you can see, frames are listed in the reverse order in which the subprograms were executed. The earliest frame is at the bottom of the stack, while the latest frame is at the top of the stack.

A stack frame contains information about the corresponding subprogram, including the module name, the package name if any, the subprogram name, and the next statement that is to be executed. For example, `EMPLOYEES.SAL.GET_SAL: : 19` indicates that line 19 of subprogram `GET_SAL` in the `SAL` package contained in the `EMPLOYEES` module will be executed when the application returns to that subprogram. When that occurs, the `EMPLOYEES.SAL.CHECK_PCT` frame will get pushed off the stack because the `EMPLOYEES.SAL.CHECK_PCT` subprogram has finished executing.

The Debug Console: Variables Panel

- Stack
- Variables
- Watch
- Form Values
- PL/SQL Packages
- Global and System Variables
- Breakpoints



The Debug Console: Variables Panel

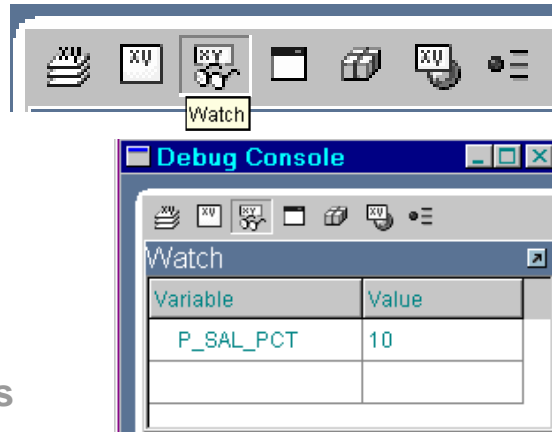
The variables panel displays the variables of the current stack frame, along with their values. There is a pop-up list from which you can select the stack frame whose variables you want to view. You can also switch the variables shown in the variable panel by clicking a different stack frame in the stack panel if it is open as well. This does not change the order of execution of program statements, but only the information that is displayed in the debug console.

A feature of the debugger is that when the form is suspended you can change the variable values by clicking into the value column and entering a new value. When the program continues, it will use the new value that you have entered, so that you can test the effect of changing a value on the final result.

Some variables, such as parameters, cannot be modified. When you click into a read-only variable, the entire cell is highlighted. Clicking into a modifiable variable will highlight only the value, not the entire cell.

The Debug Console: Watch Panel

- Stack
- Variables
- **Watch**
- Form Values
- PL/SQL Packages
- Global and System Variables
- Breakpoints



ORACLE

15-7

Copyright © 2004, Oracle. All rights reserved.

The Debug Console: Watch Panel

A running application may have dozens of variables that can be displayed in various panels in the debug console, but there may be very few that you need to monitor. The Watch panel provides a central place where you can keep track of any valid variables that you specify. Only variables which resolve to a scalar value are valid. Stored package variables are not valid.

Once a variable is displayed in your watch list, when execution is next suspended at a different location, the variable values in the list are updated as needed if they are available in the current execution context. If a variable is not defined in the currently executing subprogram, ##### displays in the cell instead of a value.

To add a variable to the Watch panel, perform the following steps:

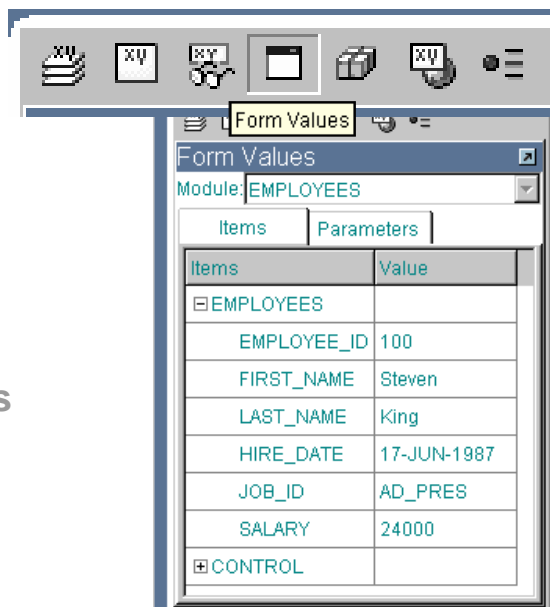
1. Open the window where the variable is displayed.
2. Select the variable that you want to add to the Watch panel.
3. Right-click the selection and choose Add to Watch from the pop-up menu.

To delete an item from the Watch panel:

- Select it in the Watch panel, right-click, and choose Remove from the pop-up menu.
- To clear the entire Watch panel, choose Remove All.

The Debug Console: Form Values Panel

- Stack
- Variables
- Watch
- **Form Values**
- PL/SQL Packages
- Global and System Variables
- Breakpoints



ORACLE

15-8

Copyright © 2004, Oracle. All rights reserved.

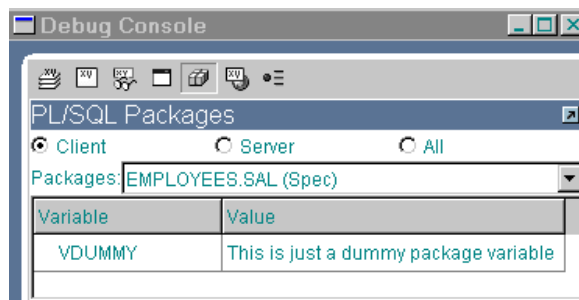
The Debug Console: Form Values Panel

You can use the form values panel to display the values of all items and parameters in modules that are currently running. You switch between a view of items and a view of parameters by clicking the corresponding tabs in the panel.

You can change the values of modifiable items to test the effects of such changes. If the value is read-only, such as display item values, the entire cell is highlighted when you try to edit it. If the value is modifiable, only the value in the cell is highlighted when you select it.

The Debug Console: PL/SQL Packages Panel

- Stack
- Variables
- Watch
- Form Values
- **PL/SQL Packages**
- Global and System Variables
- Breakpoints



ORACLE

15-9

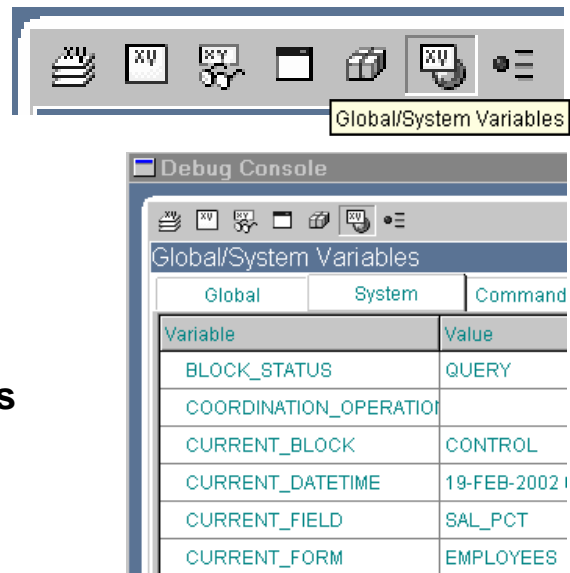
Copyright © 2004, Oracle. All rights reserved.

The Debug Console: PL/SQL Packages Panel

Use the PL/SQL Packages panel to browse and examine PL/SQL packages that have been instantiated. Both package specification and package body global variables are listed. You can view packages only while the runform process is currently executing PL/SQL. You can also select an option button to determine which packages are displayed: Client, Server, or All.

The Debug Console: Global/System Variables Panel

- Stack
- Variables
- Watch
- Form Values
- Loaded PL/SQL Packages
- **Global and System Variables**
- Breakpoints



ORACLE

15-10

Copyright © 2004, Oracle. All rights reserved.

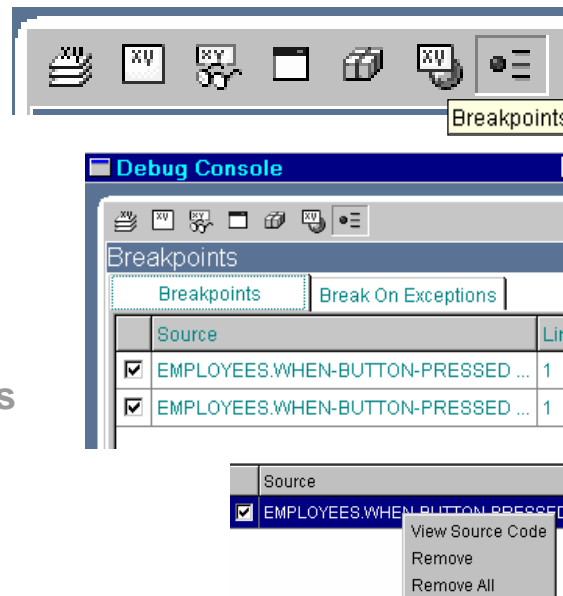
The Debug Console: Global/System Variables Panel

Use the Global/System Variables panel to display the current system, global, and command line variables and their values. You can switch among these types of variables by clicking the corresponding tabs in the panel.

Command line variables and most system variables are read-only. The only modifiable system variables are `DATE_THRESHOLD`, `EFFECTIVE_DATE`, `MESSAGE_LEVEL`, and `SUPPRESS_WORKING`. You can modify global variables, and the new values will be used subsequently by the running application.

The Debug Console: Breakpoints Panel

- Stack
- Variables
- Watch
- Form Values
- Loaded PL/SQL Packages
- Global and System Variables
- **Breakpoints**



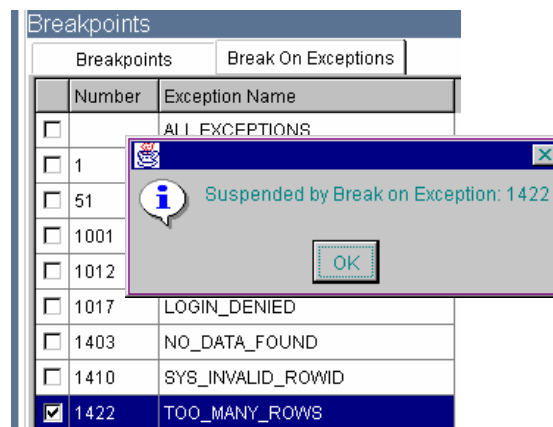
The Debug Console: Breakpoints Panel

The Breakpoints panel consists of two tabs:

- Breakpoints tab displays any breakpoints set in the code during the current Forms Builder session, in order of breakpoint creation.
Display includes:
 - Name of trigger or program unit
 - Line number where breakpoint is set
 - A check box to temporarily enable or disable the breakpoint.
- Enables navigation to source code where breakpoint is set:
 - By double-clicking the breakpoint name
 - By highlighting (single-clicking) it, then choosing View Source Code from the pop-up menu (right-click in Windows). From this pop-up menu, you can also remove the breakpoint or remove all breakpoints.

The Debug Console

- Stack
- Variables
- Watch
- Form Values
- Loaded PL/SQL Packages
- Global and System Variables
- **Breakpoints**



ORACLE

15-12

Copyright © 2004, Oracle. All rights reserved.

The Debug Console: Breakpoints Panel (continued)

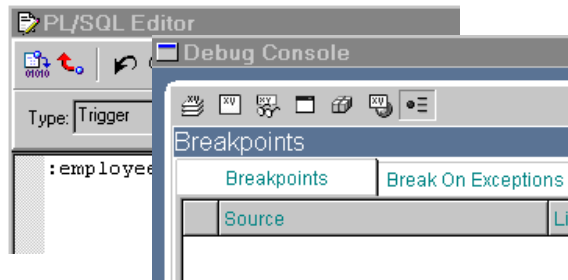
- The Break On Exceptions tab displays a list of frequently used system exceptions that you can use during debugging. The display includes:
 - Exception name
 - Associated ORA- error number
 - Check box where you set or unset the breakpoint

Setting Breakpoints in Client Code

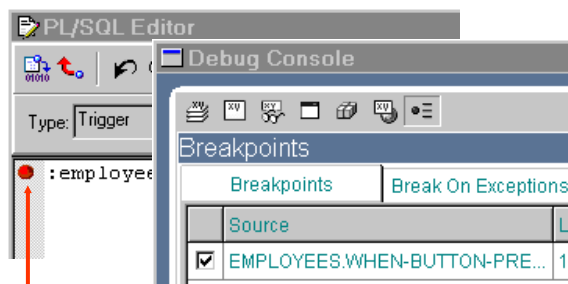
Breakpoints:

- Suspend form execution
- Return control to the debugger
- Remain in effect for the Forms Builder session
- May be enabled and disabled
- Are set in the PL/SQL Editor on executable lines of code

Before setting breakpoint:



After setting breakpoint:



ORACLE

15-13

Copyright © 2004, Oracle. All rights reserved.

Setting Breakpoints in Client Code

You set breakpoints in code so that the form running in debug mode will be suspended when a breakpoint is encountered and control will return to the Forms Builder debugger, allowing you to monitor or change the environment at that point. When you set a breakpoint, it remains set until you exit the Forms Builder session. However, you can disable and enable breakpoints as needed by unchecking and rechecking the check box in the Breakpoints panel of the Debug Console.

You can set breakpoints only on executable lines of code, such as assignments or subprogram calls. There are three ways to set a breakpoint:

- By double-clicking to the left of a line of code in the PL/SQL Editor
- By right-clicking a line of code and selecting Insert/Remove Breakpoint.
- By choosing Debug > Insert/Remove breakpoint from the main menu.

Performing the same action again unsets the breakpoint. You can also remove one or all breakpoints from the pop-up menu in the Breakpoint panel, as described previously.

Setting Breakpoints in Stored Code

- **Can set on stored program units:**
 - Expand Database Objects node
 - Expand <schema> node
 - Expand PL/SQL Stored Program Units node
 - Double-click program unit
 - Set breakpoint in PL/SQL Editor
- **Cannot set on database triggers or stored PL/SQL libraries**
- **Compile with debug information**

ORACLE

15-14

Copyright © 2004, Oracle. All rights reserved.

Setting Breakpoints in Stored Code

If you are connected to the database, you can set breakpoints in stored packages, procedures, and functions just as you do in client-side programs. To do so:

1. Expand the Database Objects node.
2. Navigate to the stored subprogram.
3. Open it in the PL/SQL Editor.

You cannot set breakpoints in database triggers or stored PL/SQL libraries.

If the program unit does not appear in a stack frame, or if you are not able to see it as you step through its code, it has not been compiled with debug information included. There are three methods to compile the stored program unit with debug information:

- Create the stored procedure in Forms Builder, which creates it with debugging information.
- Use `ALTER SESSION SET PLSQL_DEBUG=TRUE` before creating the stored procedure.
- Manually recompile an existing PL/SQL program unit using:
`ALTER PROCEDURE <schema.procedure> COMPILE DEBUG`

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.**

ORACLE

15-15

Copyright © 2004, Oracle. All rights reserved.

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

Run Form
Debug



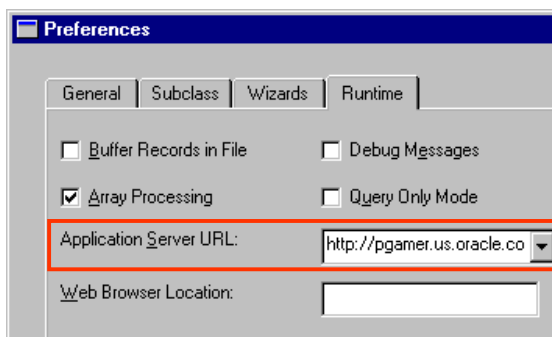
(Compiles automatically)

Contains source
code and
executable run file



(Runs automatically)

Runs Form in
Debug Mode on
Server specified
in Runtime
Preferences



ORACLE

15-16

Copyright © 2004, Oracle. All rights reserved.

Running a Form in Debug Mode

The Run Form Debug button in Forms Builder runs the form in debug mode. When a breakpoint is encountered and control passes to the Debugger in Forms Builder, you can use the debug commands to resume execution or step through the code in a variety of ways to see how each line affects the application and the environment, as you will see shortly.

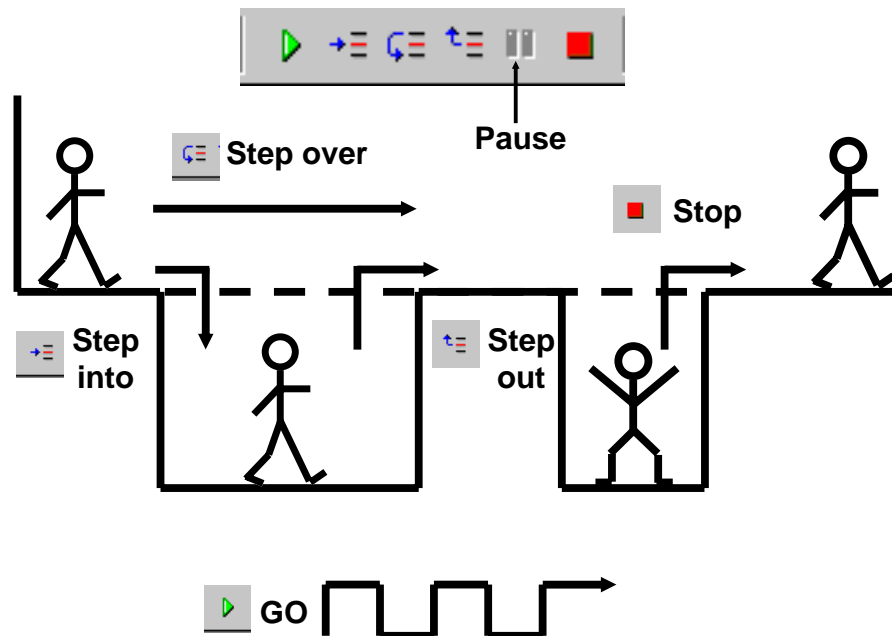
As when you run a form from Forms Builder with the Run Form button, the Run Form Debug button runs the form in a three-tier environment. It takes its settings from the Preferences window that you access by choosing Edit > Preferences from the main menu and clicking the Runtime tab.

You enter the URL for the application server that you want to run the form, which runs in your default browser unless you specify a different browser in the Web Browser Location text box. You can use a named configuration, if desired, with the `config` parameter.

Example of Application Server URL:

`http://mymachine:8889/forms90/f90servlet?config=test`
where `test` is a named section in the Forms Web configuration file (`formsweb.cfg` by default) that specifies settings to use.

Stepping Through Code



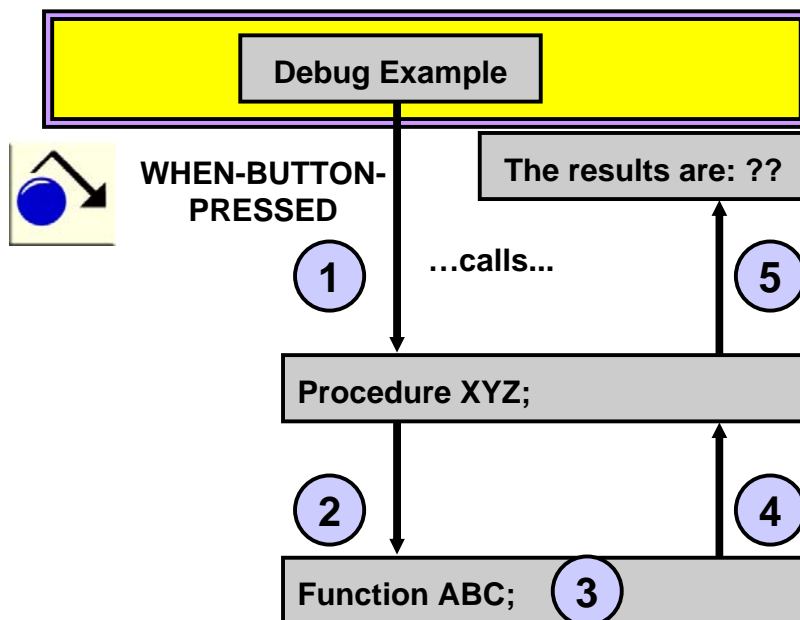
Stepping Through Code

Once the program encounters a breakpoint, the PL/SQL Debugger enables you to step through program units in a variety of ways in order to examine the environment as the program progresses, using the following buttons:

- **Step Into:** Executes the next statement
- **Step Over:** Executes the next statement without stepping into a nested subprogram
- **Step Out:** Completes the nested subprogram and steps to the next executable statement in the calling program
- **Go:** Resumes execution until the program terminates normally or is interrupted by the next breakpoint
- **Pause:** Pauses the execution of running PL/SQL code to enable you to examine the environment. For example, you could check variable values.
- **Stop:** Terminates debugging and program execution completely; the Debug Console and any opened debug panels close and application execution terminates.

Another command, available from the Debug menu, is *Run to Cursor*. When you insert the mouse cursor on a line of code in the PL/SQL Editor (by clicking on it), the *Run to Cursor* command executes all code up to that line, then stops and marks that line as the next executable line of code.

Debug Example



Debug 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.

Debug Example (continued)

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_num3 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;
```

Instructor Note

Demonstration

Open the form `DebugDemo.fmb` in Forms Builder to show the students how to use the debugger.

- First run the form normally. When you click Debug Example 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.
- Set a breakpoint in the When-Button-Pressed trigger for `Block1.push_button1` on the executable line that calls the procedure.
- Click Run Form Debug to run the `DebugDemo` form in Debug mode.
- Click Debug Example in the form. The program stops at the breakpoint.
- The XYZ procedure now displays in the PL/SQL Editor, with “=>” to mark current position at the beginning of the executable code.
- If not already displayed, bring up the Debug Console and demonstrate the Breakpoints, Variables, and Stack panels.
- Click Step Into in the Debugger to advance into the XYZ procedure.
- Examine the Stack values for the `xyz_param` and `v_results` parameters (as well as system variables). Everything looks normal in the xyz procedure.
- Click Step Into to enter the ABC function. Step through each of the opening assignment statements. Find the problem in the code (`v_num6` is incorrectly set to 8 instead of 6).
- Before proceeding, in the Variables panel, change the value of the variable to 6. Then, click Go.
- Return to the form to see that the correct result of 128 is now obtained. Point out to students that to correct the problem permanently, the PL/SQL would need to be corrected.

In previous versions of Forms, you could change PL/SQL code on the fly in the debugger. However, now that the debugger is integrated into Forms Builder, and the form itself is running in a three-tier environment, this is no longer possible.

If a more complex demo is desired, open and run `EMPLOYEES.fmb`:

- The various buttons in this form enable you to demonstrate the Package and Break on Exceptions windows.
- The Cause Exception button in the form causes a 1422 exception, so select that as the exception on which to break.

Summary

In this lesson, you should have learned that:

- The Debug Console consists of panes to view the call stack, program variables, a user-defined watch list, Form values, loaded PL/SQL packages, global and system variables, and breakpoints
- You use the Run Debug button to run a form module in debug mode within Forms Builder
- You can set breakpoints in the PL/SQL Editor by double-clicking to the left of an executable line of code
- The debug buttons in the Forms Builder toolbar enable you to step through code in various ways

ORACLE

15-20

Copyright © 2004, Oracle. All rights reserved.

Summary

To debug a form from within Forms Builder, you can:

- Click Run Form Debug (compiles and runs the form automatically)
- Set breakpoints in the code
- Use various panes of the Debug Console to view aspects of the running form
- Use the debug-related toolbar buttons to step through the code

Instructor Note (continued)

- The code error in the form is that the Update Salary button does not seem to change the salary. Set a breakpoint in the When-Button-Pressed trigger for that button. The actual error occurs in the `sal.get_sal` package procedure, which is called by the When-Button-Pressed trigger of the Update Salary button. The lines:

```
IF p_action = 'UPDATE' THEN
    v_new_sal := p_old_sal;
END IF;
```

should be corrected to:

```
IF p_action = 'UPDATE' THEN
    v_new_sal := p_old_sal + v_new_sal;
END IF;
```

Practice 15 Overview

This practice covers the following topics:

- **Running a form in debug mode from Forms Builder**
- **Setting breakpoints**
- **Stepping through code**
- **Viewing variable values while form is running**

ORACLE

15-21

Copyright © 2004, Oracle. All rights reserved.

Practice 15 Overview

In this practice session, you will run a form in debug mode from within Forms Builder, set a breakpoint, and step through code, looking at variable values as the form runs.

Note: For solutions to this practice, see Practice 15 in Appendix A, “Practice Solutions.”

Practice 15

1. Open your CUSTGXX.FMB file. In this form, create a procedure that is called List_Of_Values. Import code from the pr15_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 = TRUE 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 of CONTROL.Account_Mgr_LOV_Button in order to call this procedure. Misspell the parameter to pass the LOV name.
3. Compile your form and click Run Form to run it. Press the LOV button for the Account Manager. Notice that the LOV does not display, and you receive a message that 'You have just cancelled the List of Values'.
4. Now click Run Form Debug. Set a breakpoint in your When-Button-Pressed trigger, and investigate the call stack. Try stepping through the code to monitor its progress. Look at the Variables panel to see the value of the parameters you passed to the procedure, and the value of the p_lov variable in the procedure. How would this information help you to figure out where the code was in error?

16

Adding Functionality to Items

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	40 minutes	Lecture
	40 minutes	Practice
	80 minutes	Total

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**

ORACLE

16-2

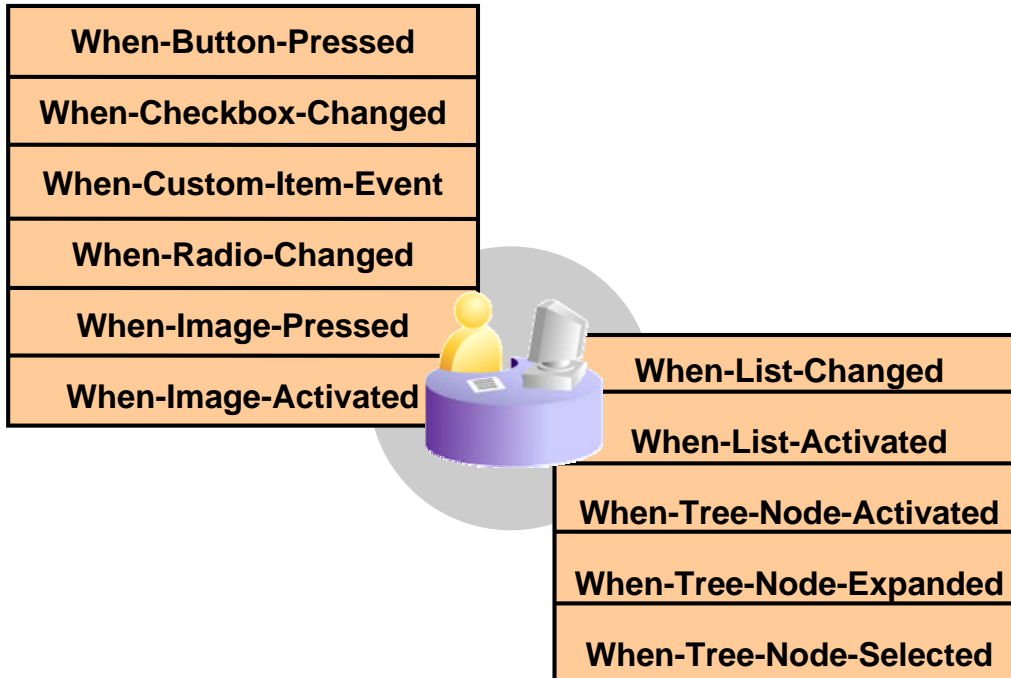
Copyright © 2004, Oracle. All rights reserved.

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



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)

Instructor Note

The [When-Image-Pressed](#) trigger fires when the user clicks the image. The [When-Image-Activated](#) trigger fires when the user double-clicks the image. The mouse-event triggers, [When-Mouse-Click](#) and [When-Mouse-DoubleClick](#), may also exist within the scope of the objects discussed here. In this case, the mouse-event triggers fire after the item interaction trigger, when the mouse is used by the user.

Item Interaction Triggers (continued)

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 clicks with mouse or uses function key to select
When-Checkbox-Changed	User changes check box state by clicking or by pressing a function key
When-Custom-Item-Event	User selects or changes the value of a JavaBean component
When-Radio-Changed	User selects different button, or deselects current button, in a radio group
When-Image-Pressed	User 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
When-Tree-Node-Activated	User double-clicks a node or presses [Enter] when a node is selected
When-Tree-Node-Expanded	User expands or collapses a node
When-Tree-Node-Selected	User selects or deselects a node

Coding Item Interaction Triggers

- **Valid commands:**
 - **SELECT statements**
 - **Standard PL/SQL constructs**
 - **All built-in subprograms**
- **Do not fire during:**
 - **Navigation**
 - **Validation (use When-Validate-“object” to code actions to take place during validation)**

ORACLE

16-5

Copyright © 2004, Oracle. All rights reserved.

Command Types in Item Interaction Triggers

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

Although Forms allows you to use DML (INSERT, UPDATE, or DELETE) statements in any trigger, it is best to use them in commit triggers only. Otherwise the DML statements are not included in the administration kept by Forms concerning commit processing. This may lead to unexpected and unwanted results. You learn about commit triggers in Lesson 21.

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 (continued)

Example of When-Radio-Changed

When-Radio-Changed trigger on :CUSTOMERS.Credit_Limit. When the user changes the credit limit, this trigger immediately confirms whether the customer has outstanding orders exceeding the new credit limit. If so, a message warns the user.

```
DECLARE
  n NUMBER;
  v_unpaid_orders NUMBER;
BEGIN
  SELECT SUM(nvl(unit_price,0)*nvl(quantity,0))
    INTO v_unpaid_orders
    FROM orders o, order_items i
    WHERE o.customer_id = :customers.customer_id
    AND o.order_id = i.order_id
  -- Unpaid credit orders have status between 4 and 9
  AND (o.order_status > 3 AND o.order_status < 10);
  IF v_unpaid_orders > :customers.credit_limit THEN
    n := SHOW_ALERT('credit_limit_alert');
  END IF;
END;
```

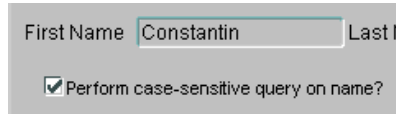
Note: Displaying alerts is discussed in the next lesson.

Instructor Note

Demonstration

- Use the `customers.fmb` file to show the When-Radio-Changed trigger on `Credit_Limit`. Run the `customer.fmb` file to show the functionality of the trigger.
- Use Customer ID 101, because this customer has some outstanding credit orders. Click the Account Information tab and change the customer's credit limit to Low. An alert displays to warn that the new credit limit has been exceeded. You can click the Orders button to show the customer's outstanding orders.
Note: The data included in the common schema, used as a basis for the examples and practices in this course, may not fit with what would exist in the real world. For example, most customers with outstanding credit orders actually exceed their set credit limits.
- Explain the actions in the trigger code.
- We will discuss using `SELECT` statements in triggers in the lesson on query triggers.

Interacting with Check Boxes



First Name Last Name
 Perform case-sensitive query on name?

When-Checkbox-Changed

```
IF CHECKBOX_CHECKED('CONTROL.case_sensitive') THEN
  SET_ITEM_PROPERTY('CUSTOMERS.cust_first_name',
    CASE_INSENSITIVE_QUERY, PROPERTY_FALSE);
  SET_ITEM_PROPERTY('CUSTOMERS.cust_last_name',
    CASE_INSENSITIVE_QUERY, PROPERTY_FALSE);
ELSE
  SET_ITEM_PROPERTY('CUSTOMERS.cust_first_name',
    CASE_INSENSITIVE_QUERY, PROPERTY_TRUE);
  SET_ITEM_PROPERTY('CUSTOMERS.cust_last_name',
    CASE_INSENSITIVE_QUERY, PROPERTY_TRUE);
END IF;
```

ORACLE

16-7

Copyright © 2004, Oracle. 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

The `When-Checkbox-Changed` trigger (shown in the slide) on the `:CONTROL.Case_Sensitive` item enables a query to be executed without regard to case if the box is not checked.

Changing List Items at Run Time

Triggers:

- **When-List-Changed**
- **When-List-Activated**

Built-ins:

- **ADD_LIST_ELEMENT**
- **DELETE_LIST_ELEMENT**

	Index
Excellent	1
Good	2
Poor	3

List Items

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

With Forms 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	Number identifying the element position in the list (top is 1)
Label	The name of the element
Value	The new value for the element

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

At run time, when the block contains queried or changed records Forms may not allow you to add or delete elements from a list item.

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

ORACLE

16-9

Copyright © 2004, Oracle. All rights reserved.

Defining Functionality for Noninput Items

Displaying LOVs from Buttons

If you have attached an 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 an LOV. The button has two advantages:

- It is convenient alternative for accessing the LOV.
- It displays an 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`.

Defining Functionality for Noninput Items (continued)

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 )
```

You should note 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.

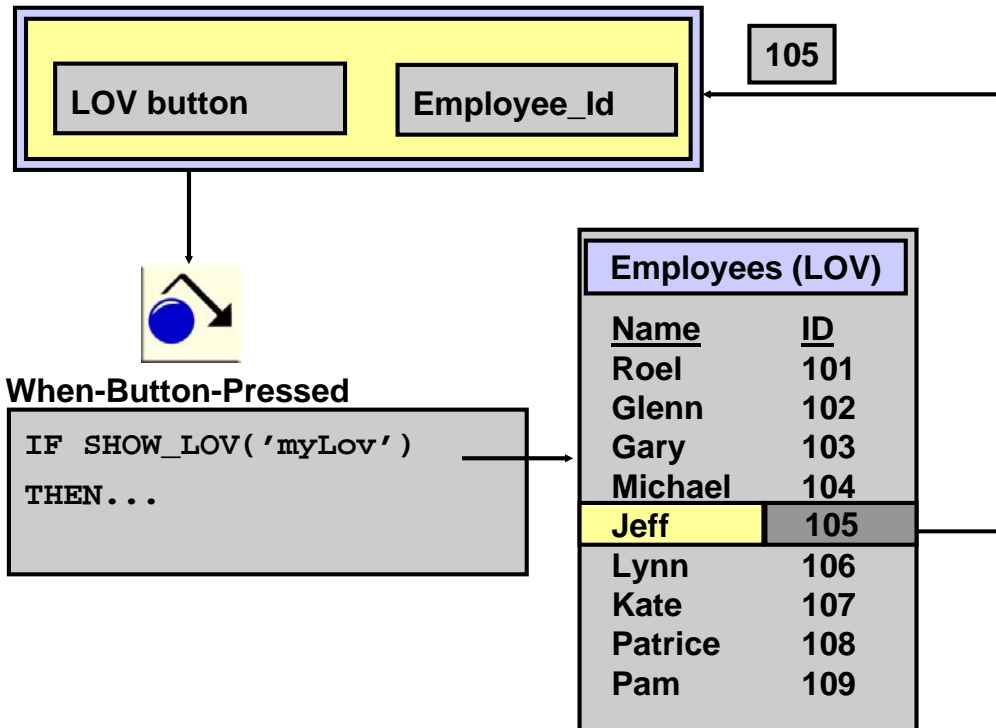
Instructor Note

Use `LIST_VALUES` with one button for each item that has an LOV.

Use `SHOW_LOV` with a single button for the whole form.

In both cases, set the `Keyboard Navigable` and `Mouse Navigate` properties of the button to `No`.

LOVs and Buttons



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.

Using the `SHOW_LOV` Function (continued)

- 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 redisplay until they do so.

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

Populating Image Items

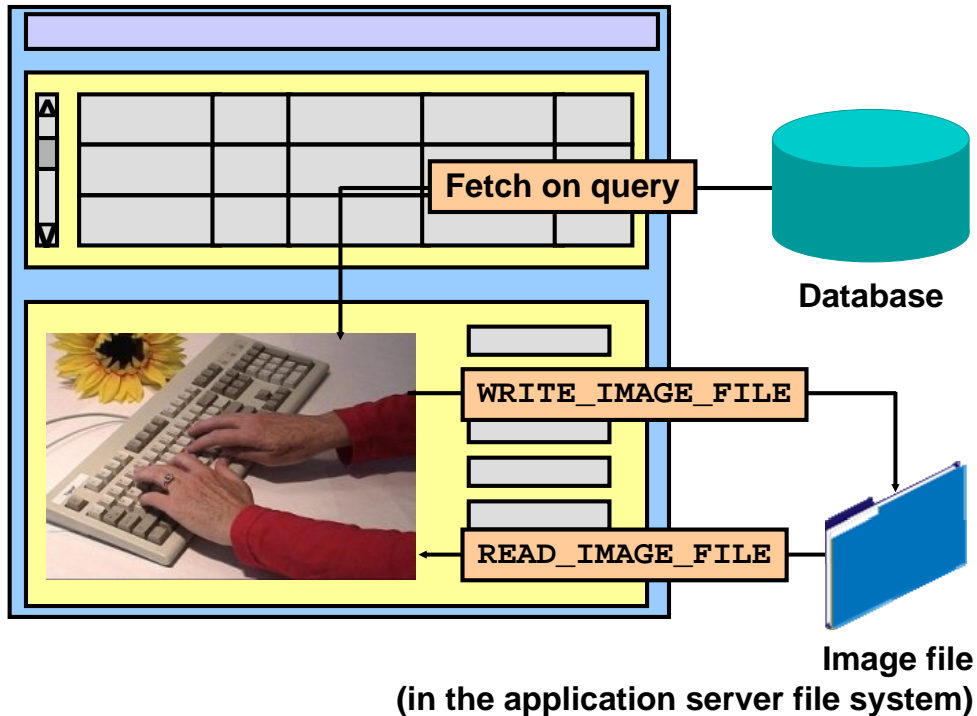


Image Items

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 or BLOB 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)

Note: The READ_IMAGE_FILE built-in procedure loads an image file from the application server file system. If you need to load an image file from the file system on the client, use a JavaBean.

Image Items (continued)

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');
```

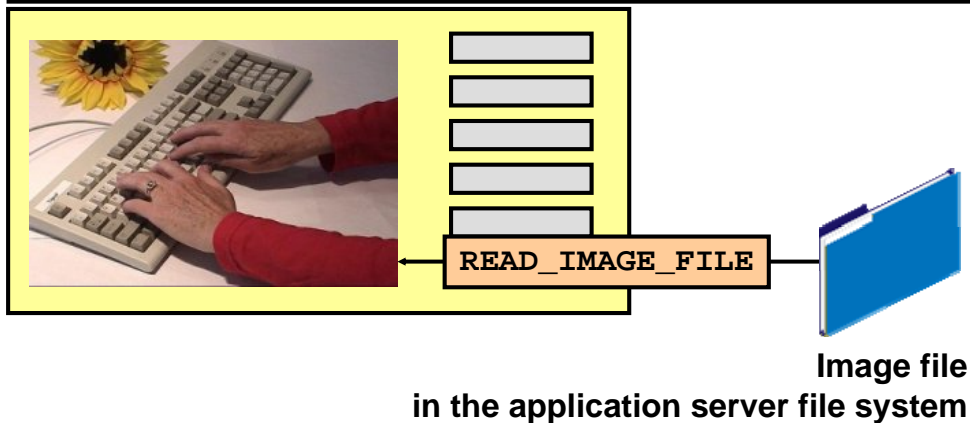
Parameter	Description
filename	Image file name (without a specified path, default path is assumed)
filetype	File type of the image (You can use ANY as a value, but it is recommended to set a specific file type for better performance. Refer to online Help for file types.)
item_name	Name of the image item (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.
- The `WRITE_IMAGE_FILE` built-in procedure writes an image file to the application server file system. If you need to write an image file to the file system on the client, use a `JavaBean`.

Loading the Right Image

```
READ_IMAGE_FILE  
(TO_CHAR(:ORDER_ITEMS.product_id) || '.JPG',  
'JPEG', 'ORDER_ITEMS.product_image' );
```



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:

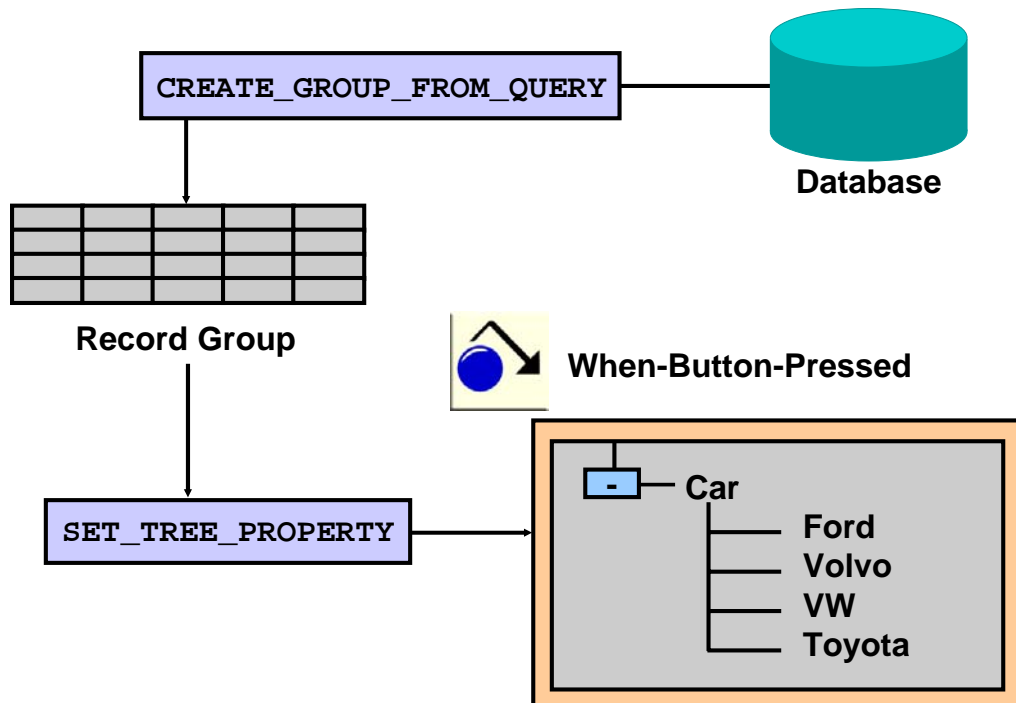
`<product id>.jpg`.

```
READ_IMAGE_FILE(TO_CHAR(:ORDER_ITEMS.product_id) || '.jpg',  
'JPEG', 'ORDER_ITEMS.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` or `BLOB` column when you save changes in the form. You can use this technique to populate a table with images.

Populating Hierarchical Trees



ORACLE

16-16

Copyright © 2004, Oracle. All rights reserved.

Populating Hierarchical Trees

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

The FTREE Package

The FTREE package contains built-ins and constants to interact with hierarchical tree items in a form. To utilize the built-ins and constants, you must precede their names with the name of the package.

Instructor Note

This example uses a record group to populate the tree. Developers can also use a query to populate a hierarchical tree. The example used in the following slides is contained in the `HTreeDemo.fmb` file. The `HTreeDemo2.fmb` file contains a more complete example, using the data stored in the tree nodes to display details about a specific employee.

Populating Hierarchical Trees (continued)

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. A variable holding the Item_id is also valid for this argument.
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 a 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.

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-Button-Pressed 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, last_name, NULL, to_char(employee_id) '
    ||' from employees ' ||
    'connect by prior employee_id = manager_id ' ||
    'start with job_id = ''AD_PRES''');
    v_ignore := Populate_Group(rg_emps);
    Ftree.Set_Tree_Property(htree, Ftree.RECORD_GROUP, rg_emps);
END;
```

Displaying Hierarchical Trees

When-Button-Pressed

```
rg_emps := create_group_from_query('rg_emps',
  'select 1, level, last_name, NULL,
  to_char(employee_id) ' ||
  'from employees ' ||
  'connect by prior employee_id = manager_id ' ||
  'start with job_id = ''AD_PRES''');

v_ignore := populate_group(rg_emps);

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

ORACLE

16-18

Copyright © 2004, Oracle. All rights reserved.

Displaying Hierarchical Trees

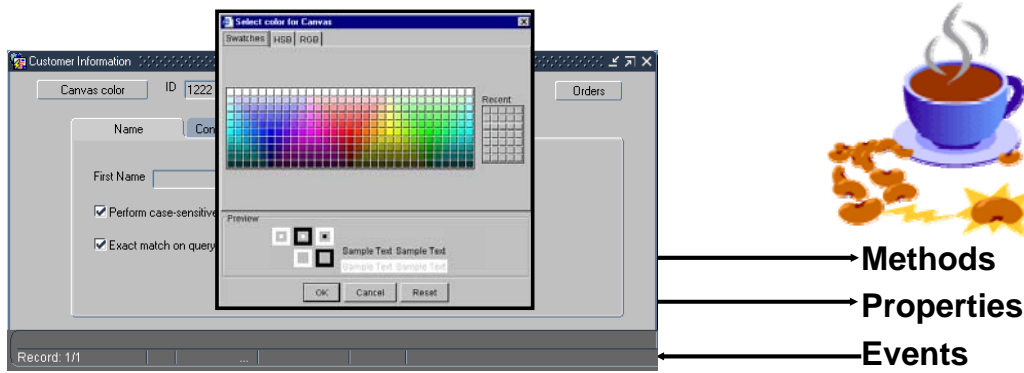
The Record Group or Query

The columns in a record group or query that are used to populate a hierarchical tree are:

- **Initial state:** 0 (not expandable), 1 (expanded), or -1 (collapsed)
- **Node tree depth:** Use LEVEL pseudocolumn
- **Label for the node:** What the user sees
- **Icon for the node:** Picture displayed, if any
- **Data:** Actual value of the node

Interacting with JavaBeans

- **Tell Forms about the bean: Register**
- **Communication from Forms to JavaBean:**
 - **Invoke Methods**
 - **Get/Set Properties**
- **Communication from JavaBean to Forms: Events**



Interacting with JavaBeans

In Lesson 10, you learned how to add a JavaBean to a form using the Bean Area item. The bean that you add to a form may have a visible component on the form itself, such as a Calendar bean that has its own button to invoke the bean. However, JavaBeans (such as the ColorPicker bean) do not always have visible component, so you may need to create a button or other mechanism to invoke the bean.

Regardless of whether the bean is visible in the bean area, there must be some communication between the run-time form and the Java classes that comprise the bean. First, the form must be made aware of the bean, either by setting its Implementation Class property at design time or by registering the bean and its events at run time. Once the form knows about the bean, the form communicates to the bean by:

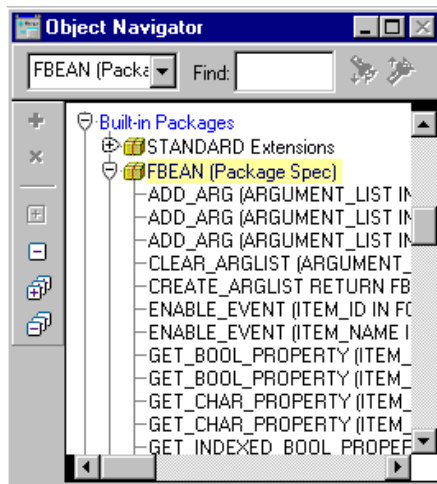
- Invoking the methods of the bean
- Getting and setting properties of the bean

The bean communicates to the form by:

- Sending an event, such as the fact that the user selected a date or color
- Sending a list containing information needed by the form, such as what date or color was selected
- Returning a value from an invoked method

Interacting with JavaBeans

The **FBEAN** package provides built-ins to:



- Register the bean
- Invoke methods of the bean
- Get and set properties on the bean
- Subscribe to bean events

ORACLE

16-20

Copyright © 2004, Oracle. All rights reserved.

Interacting with JavaBeans (continued)

The **FBEAN** package

The **FBEAN** package contains Forms built-ins that enable you to code interactions with JavaBeans in PL/SQL, eliminating the need to know Java in order to communicate with the bean.

Many of the built-ins take some of the same arguments:

- **Item Name or Item Id (obtained with the **FIND_ITEM** built-in):** The first argument for most of the **FBEAN** built-ins, referred to on the next page simply as **ITEM**.
- **Item Instance:** A reference to which instance of the item should contain the bean. This is applicable where the Bean Area is part of a multi-row block and more than one instance of the Bean Area is displayed. This is referred to on the next page as **INSTANCE**. You can use the value **ALL_ROWS** (or **FBEAN.ALL_ROWS**) for the Item Instance value to indicate that command should apply to all of the instances of this Bean Area in the block.
Note: This refers to the UI instance of the Bean Area, not the row number in the block. For example, in a block with 5 rows displayed and 100 rows queried, there will be 5 instances of the bean numbered 1 through 5, not 100 instances.
- **Value:** Can accept **BOOLEAN**, **VARCHAR2**, or **NUMBER** datatypes.

Interacting with JavaBeans (continued)

The FBEAN package (continued)

Some of the built-ins in the FBEAN package are:

- `GET_PROPERTY (ITEM , INSTANCE , PROPERTY_NAME)` (returns VARCHAR2):
Function that retrieves the value of the specified property
- `SET_PROPERTY (ITEM , INSTANCE , PROPERTY_NAME , VALUE)`:
Sets the specified property of the bean to the value indicated
- `INVOKE (ITEM , INSTANCE , METHOD_NAME [, ARGUMENTS])`:
Invokes a method on the bean, optionally passing arguments to the method
- `REGISTER_BEAN (ITEM , INSTANCE , BEAN_CLASS)`:
Registers the bean with the form at run time, making all its exposed attributes and methods available for the form's bean item (The last argument is the full class name of the bean, such as 'oracle.forms.demos.beans.ColorPicker'.)
- `ENABLE_EVENT (ITEM , INSTANCE , EVENT_LISTENER_NAME , SUBSCRIBE)`; the last argument is a BOOLEAN indicating whether to subscribe (TRUE) or unsubscribe (FALSE) to the event.

Remember to precede calls to any of these built-ins with the package name and a dot, such as `FBEAN.GET_PROPERTY (...)`. You can pass arguments to these built-ins as either a delimited string or as an argument list.

Deploying the Bean

Because the bean itself is a Java class or set of Java class files separate from the form module, you need to know where to put these files. You can locate these either:

- On the middle-tier server, either in the directory structure referenced by the form applet's CODEBASE parameter or in the server's CLASSPATH. CODEBASE is by default the `forms90\java` subdirectory of ORACLE_HOME.
- If using JInitiator, in a JAR file in the middle-tier server's CODEBASE directory, and included in the ARCHIVE parameter so that the JAR file is downloaded to and cached on the client. For example:

```
archive_jini=f90all_jinit.jar,colorpicker.jar
```

(The CODEBASE and ARCHIVE parameters are set in the `formsweb.cfg` file.)

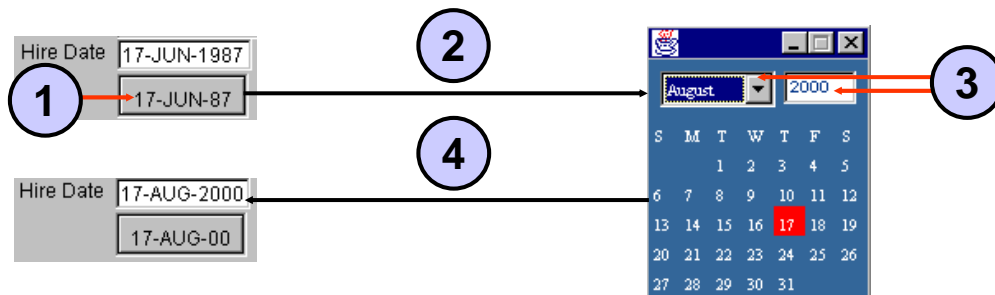
Instructor Note

When the JavaBean is registered, its properties are given a name that is derived from the JavaBean setters and getters for that property. For example, the Juggler JavaBean has a *rate* property exposed through the bean methods *getAnimationRate* and *setAnimationRate*. Once the bean is registered with `FBEAN.REGISTER_BEAN`, the custom property *animationRate* is made available for that Bean Area item in the form.

To see all the methods and properties of the JavaBean, you can use `FBEAN.SET_LOGGING_MODE` to make the information appear on the Java Console.

Interacting with JavaBeans

- **Register a listener for the event:**
`FBEAN.ENABLE_EVENT('MyBeanArea',1,'mouseListener', true);`
- **When an event occurs on the bean:**
 - The When-Custom-Item-Event trigger fires.
 - The name and information are sent to Forms in:
`:SYSTEM.CUSTOM_ITEM_EVENT`
`:SYSTEM.CUSTOM_ITEM_EVENT_PARAMETERS`



ORACLE

16-22

Copyright © 2004, Oracle. All rights reserved.

Interacting with JavaBeans (continued)

Responding to Events

When a user interacts with a JavaBean at run time, it usually causes an event to occur. You can use `FBEAN.ENABLE_EVENT` to register a listener for the event, so that when the event occurs Forms will fire the When-Custom-Item-Event trigger. In this trigger, you can code a response to the event. The `:SYSTEM.CUSTOM_ITEM_EVENT` and `:SYSTEM.CUSTOM_EVENT_PARAMETERS` variables contain the name of the event and information the bean is sending to the form.

A typical interaction that could occur includes the following steps:

1. The user clicks the bean area for a Calendar bean. This bean area has a visible component on the form that looks like a button. The label is set to the hire date for an employee.
2. The Calendar bean is invoked and displays a calendar initially set to the employee's hire date.
3. The user changes the date on the bean by picking a new month and year, then clicking on a day, which initiates the `DateChanged` event.
4. The When-Custom-Item-Event trigger obtains the changed date and assigns it back to the employee `hire_date` item, also changing the label on the bean area "button".

Interacting with JavaBeans (continued)

Coding a When-Custom-Item-Event Trigger

In a When-Custom-Item-Event trigger to respond to JavaBeans events, you code the action that you want to take place when an event occurs.

For example, when a user selects a date from the Calendar bean, the DateChange event takes place and the When-Custom-Item-Event trigger fires. In the code for the When-Custom-Item-Event trigger on the Calendar bean area item, you need to obtain the name of the event. If it is the DateChange event, you must obtain the new date and assign it to a form item, such as the employee's hire date. You can use the system variables containing the event and parameter information:

```
declare
    hBeanEventDetails ParamList;
    eventName varchar2(80);
    paramType number;
    eventType varchar2(80);
    newDateVal varchar2(80);
    newDate date := null;
begin
    hBeanEventDetails := get_parameter_list
        (:system.custom_item_event_parameters);
    eventName := :system.custom_item_event;
    if(eventName = 'DateChange') then
        get_parameter_attr(hBeanEventDetails,
            'DateValue', ParamType, newDateVal);
        newDate := to_date(newDateVal, 'DD.MM.YYYY');
    end if;
    :employees.hire_date := newDate;
end;
```

The above example is for a bean that uses hand-coded integration. If you use the FBEEAN package to integrate the bean, the name of the value passed back to the form is always called 'DATA'.

For example:

```
get_parameter_attr(:system.custom_item_event_parameters,
    'DATA', paramType, eventData);
```

(where paramType and eventData are PL/SQL variables you declare in the When-Custom-Item-Event trigger, like paramType and newDateVal in the preceding example).

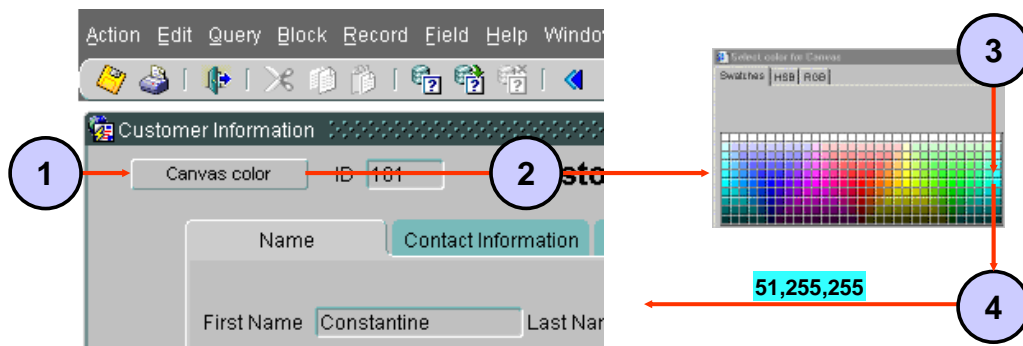
Note: There are many examples of JavaBeans in the Forms Demos that you can download from OTN:

http://otn.oracle.com/sample_code/products/forms/index.html

Interacting with JavaBeans

The JavaBean may:

- Not have a visible component
- Not communicate via events
- Return a value to the form when invoked (use like a function)



ORACLE

16-24

Copyright © 2004, Oracle. All rights reserved.

Interacting with JavaBeans (continued)

Getting Values from JavaBeans without Events

Not all information from JavaBeans is obtained via events. For example, a JavaBean may return a value when one of its methods is invoked. This value may be assigned to a PL/SQL variable or Forms item, similarly to the way a function returns a value.

An example of this is the ColorPicker bean that you added to the Customer form in Lesson 9. It contains a single method that returns a value, and also has no visible component in the bean area of the form. To invoke the bean and obtain a value from it, you can use a Forms push button and trigger with code similar to the following:

```
vcNewColor := FBean.Invoke_char(hColorPicker,  
1,'showColorPicker','"Select color for canvas"');
```

The INVOKE_CHAR built-in is used to call a method that returns a VARCHAR2 value.

1. User clicks button to invoke the bean.
2. Color Picker component displays.
3. User selects a color.
4. Color value (RGB values in comma-separated list) is returned to the vcNewColor variable. The code can then use the color value to set the canvas color.

Summary

In this lesson, you should have learned that:

- **You can use triggers to supplement the functionality of:**
 - **Input items:**
When-[Checkbox | Radio]-Changed
When-List-[Changed | Activated]
 - **Noninput items:**
When-Button-Pressed
When-Image-[Pressed | Activated]
When-Tree-Node-[Activated | Expanded | Selected]
When-Custom-Item-Event

ORACLE

16-25

Copyright © 2004, Oracle. All rights reserved.

Summary

In this lesson, you should have 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.

Summary

- **You can call useful built-ins from triggers:**
 - `CHECKBOX_CHECKED`
 - `[ADD | DELETE]_LIST_ELEMENT`
 - `SHOW_LOV`
 - `[READ | WRITE]_IMAGE_FILE`
 - `FTREE: POPULATE_TREE, ADD_TREE_DATA, [GET | SET]_TREE_PROPERTY`
 - `FBEAN: [GET | SET]_PROPERTY, INVOKE, REGISTER_BEAN, ENABLE_EVENT`

ORACLE

16-26

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- There are built-ins for check boxes, LOV control, list item control, image file reading, hierarchical tree manipulation, interaction with JavaBeans, and so on.

Practice 16 Overview

This practice covers the following topics:

- Writing a trigger to check whether the customer's credit limit has been exceeded
- Creating a toolbar button to display and hide product images
- Coding a button to enable users to choose a canvas color for a form

ORACLE

16-27

Copyright © 2004, Oracle. All rights reserved.

Practice 16 Overview

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

- Writing a trigger to check whether the customer's credit limit has been exceeded
- Coding a button to enable users to choose a canvas color for a form
- Creating a toolbar button to display and hide product images

Note: For solutions to this practice, see Practice 16 in Appendix A, "Practice Solutions."

Instructor Note

Be sure the students understand that the code they will import for this practice will only toggle the image item. The code to display an image in the image item will be implemented in a later practice.

Practice 16

1. In the CUSTGXX form, write a trigger that fires when the credit limit changes. The trigger should display a message warning the user if a customer's outstanding credit orders (those with an order status between 4 and 9) exceed the new credit limit. You can import the `pr16_1.txt` file.
2. Click Run Form to run the form and test the functionality.
Hint: Most customers who have outstanding credit orders exceed the credit limits, so you should receive the warning for most customers. (If you wish to see a list of customers and their outstanding credit orders, run the `CreditOrders.sql` script in SQL*Plus.) Customer 120 has outstanding credit orders of less than \$500, so you shouldn't receive a warning when changing this customer's credit limit.
3. Begin to implement a JavaBean for the ColorPicker bean area on the CONTROL block that will enable a user to choose a color from a color picker.
Create a button on the CV_CUSTOMER canvas to enable the user to change the canvas color using the ColorPicker bean.
Set the following properties on the button:
 Label: Canvas Color Mouse Navigate: No
 Keyboard Navigable: No Background color: white
The button should call a procedure named PickColor, with the imported text from the `pr16_3.txt` file.
The bean will not function at this point, but you will write the code to instantiate it in Practice 20.
4. Save and compile the form. You will not be able to test the Color button yet, because the bean does not function until you instantiate it in Practice 20.
5. In the ORDGXX form CONTROL block, create a new button called Image_Button and position it on the toolbar. Set the Label property to Image Off.
6. Import the `pr16_6.txt` file 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.
7. Save and compile the form. Click Run Form to run the form and test the functionality.
Note: The image will not display in the image item at this point; you will add code to populate the image item in Practice 20.

17

Run Time Messages and Alerts

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	45 minutes	Lecture
	20 minutes	Practice
	65 minutes	Total

Objectives

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

- **Describe the default messaging behavior of a form**
- **Handle run-time failure of built-in subprograms**
- **Identify the different types of Forms messages**
- **Control system messages**
- **Create and control alerts**
- **Handle database server errors**

ORACLE

17-2

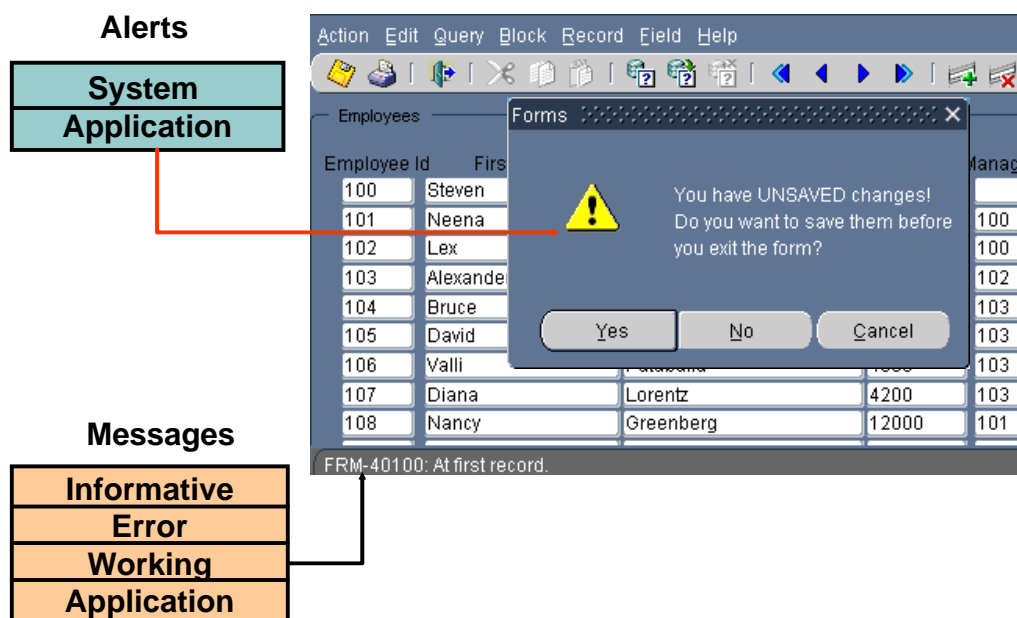
Copyright © 2004, Oracle. All rights reserved.

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.

Run-Time Messages and Alerts Overview



ORACLE

17-3

Copyright © 2004, Oracle. All rights reserved.

Run-Time Messages and Alerts Overview

Forms 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.

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

Run-Time Messages and Alerts Overview (continued)

- **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.

Instructor Note

You can point out to students that the error messages can be used to look up information in MetaLink to help resolve errors.

The URL for MetaLink is <http://metalink.oracle.com>. Click MetaLink Search and search on the error number, such as FRM-40735. You can further refine your search by selecting the product, Oracle Forms.

Detecting Run-Time Errors

- **FORM_SUCCESS**
 - **TRUE: Action successful**
 - **FALSE: Error/Fatal error occurred**
- **FORM_FAILURE**
 - **TRUE: A nonfatal error occurred**
 - **FALSE: Action successful or a fatal error occurred**
- **FORM_FATAL**
 - **TRUE: A fatal error occurred**
 - **FALSE: Action successful or a nonfatal error occurred**

ORACLE

17-5

Copyright © 2004, Oracle. 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 INVENTORIES block, and performs a query there.

```
GO_BLOCK( ' INVENTORIES ' ) ;  
EXECUTE_QUERY ;
```

If the GO_BLOCK built-in procedure fails because the INVENTORIES 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-Ins and Handling Errors (continued)

Built-In functions for detecting success and failure

Forms 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 . . .`
`OR IF NOT FORM_SUCCESS THEN . . .`
- **What went wrong?**
 - `ERROR_CODE, ERROR_TEXT, ERROR_TYPE`
 - `MESSAGE_CODE, MESSAGE_TEXT, MESSAGE_TYPE`

ORACLE

17-7

Copyright © 2004, Oracle. 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( ' INVENTORIES' );  
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.

Errors and Built-Ins (continued)

Note: The program unit `CHECK_PACKAGE_FAILURE`, which is written when you build master-detail blocks, may be called to fail a trigger if the last action was unsuccessful.

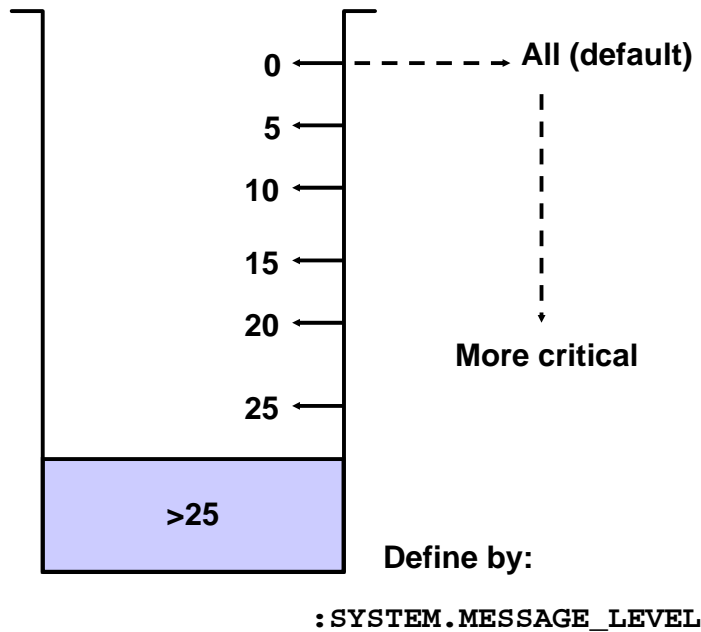
Built-in Functions to Determine the Error

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
<code>ERROR_CODE</code>	Error number (datatype NUMBER)
<code>ERROR_TEXT</code>	Error description (datatype CHAR)
<code>ERROR_TYPE</code>	FRM=Forms Builder error, ORA=Oracle error (datatype CHAR)

These built-ins are explained in detail later in this lesson.

Message Severity Levels



ORACLE

17-9

Copyright © 2004, Oracle. 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. Forms 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 cannot be suppressed

Controlling System Messages (continued)

Suppressing messages according to their severity (continued)

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`. Forms 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';
```

ORACLE

17-11

Copyright © 2004, Oracle. 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';
```

Example of Suppressing Messages (continued)

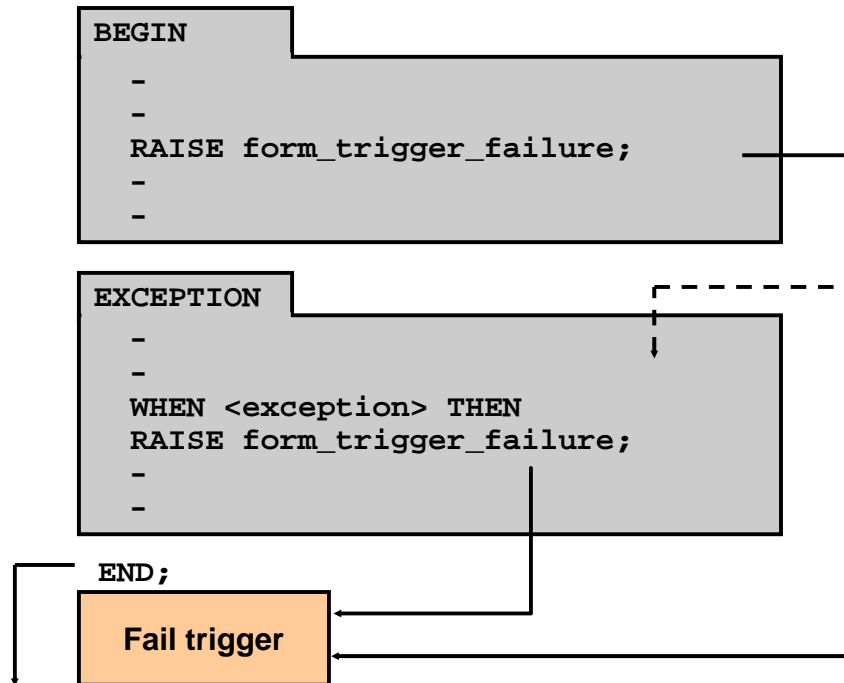
Suppressing working messages

Working messages are displayed when the Forms is busy processing an action. For example, while querying you receive the message: *Working . . .* 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.

The FORM_TRIGGER_FAILURE Exception



ORACLE

17-13

Copyright © 2004, Oracle. All rights reserved.

The FORM_TRIGGER_FAILURE Exception

Triggers fail only when one of the following occurs:

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

This exception is defined and handled by Forms 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, Forms 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.

The FORM_TRIGGER_FAILURE Exception (continued)

Example

This example adds an action to the exception handler of the When-Validate-Item trigger for the Customer_ID item. It raises an exception to fail the trigger when the message is sent, and therefore traps the user in the Customer_ID item:

```
SELECT cust_first_name || ' ' || cust_last_name
       INTO :ORDERS.customer_name
       FROM CUSTOMERS
       WHERE customer_id = :ORDERS.customer_id;
EXCEPTION
WHEN no_data_found THEN
    MESSAGE('Customer with this ID not found');
    RAISE form_trigger_failure;
```

Instructor Note

The broken line in the diagram represents possible route into the EXCEPTION section if the designer has coded an exception handler for FORM_TRIGGER_FAILURE or OTHERS.

Triggers for Intercepting System Messages

- **On-Error:**
 - Fires when a system error message is issued
 - Is used to trap Forms 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

ORACLE

17-15

Copyright © 2004, Oracle. 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 Forms 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.

Triggers for Intercepting System Messages (continued)

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;
```

Instructor Note

Remind students that On- triggers fire in place of the usual processing.

Handling Informative Messages

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

ORACLE

17-17

Copyright © 2004, Oracle. 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
<code>MESSAGE_CODE</code>	Number of informative message that would have displayed (data type NUMBER)
<code>MESSAGE_TEXT</code>	Text of informative message that would have displayed (datatype CHAR)
<code>MESSAGE_TYPE</code>	FRM=Forms Builder message ORA=Oracle server message NULL=No message issued yet in this session (datatype CHAR)

On-Message Trigger (continued)

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) and the “Query caused no records to be retrieved. Re-enter.” message (40301):.

```
IF MESSAGE_CODE in (40350,40301) THEN
  MESSAGE('No Orders found-check your search values');
ELSE
  MESSAGE(MESSAGE_TYPE || '-' || TO_CHAR(MESSAGE_CODE)
    || ':' || MESSAGE_TEXT);
END IF;
```

Setting Alert Properties

Title	This is the Title
Message	Alert Message (Maximum 200 characters) Can appear on multiple lines
Alert Style	Caution
Button 1 Label	Label 1
Button 2 Label	Label 2
Button 3 Label	Label 3
Default Alert Button	Button 1

Alert Styles:

- Caution
- Stop
- Note

ORACLE

17-19

Copyright © 2004, Oracle. All rights reserved.

Setting Alert Properties Example

The slide shows a generic example of an alert, showing all three icons and buttons that can be defined.

1	Title
2	Message
3	Alert Style (Caution, Stop, Note)
4	Button1 label
5	Button2 label
6	Button3 label
7	Default Alert Button

Instructor Note

Create an alert that you can display in your application during a later demonstration. You may want to define more than one button so that you can show the testing of user response, as in the Delete Record example that follows.

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 perform 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

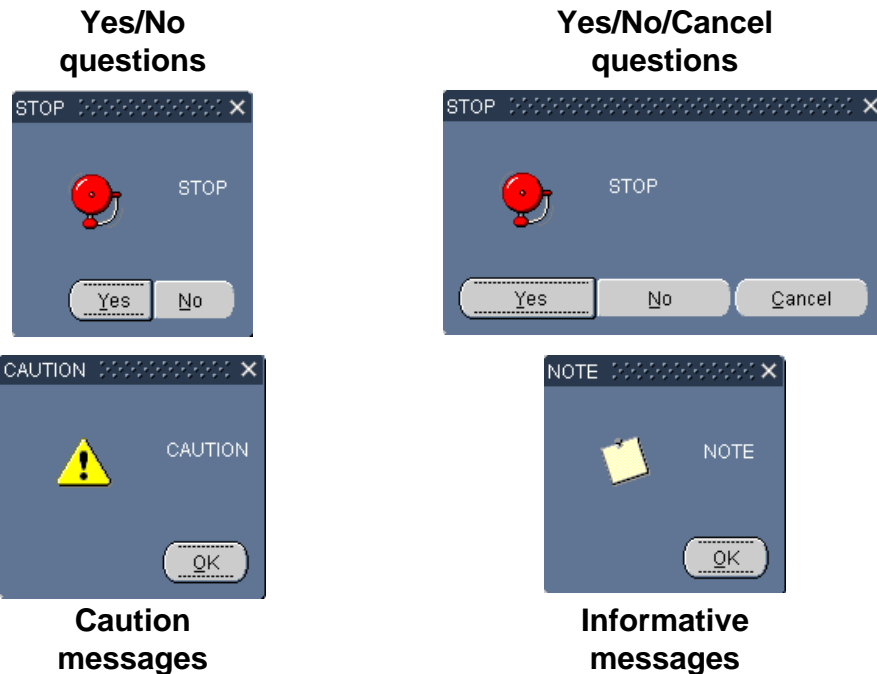
Like other objects you create at design-time, alerts are created from the Object Navigator.

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

The properties that you can specify for an alert include the following:

Property	Description
Name	Name for this object
Title	Title that displays on alert
Alert Style	Symbol that displays on alert: Stop, Caution, or Note
Button1, Button2, Button3 Labels	Labels for each of the three possible buttons (Null indicates that the button will not be displayed)
Default Alert Button	Specifies which button is selected if user presses [Enter]
Message	Message that will appear in the alert – can be multiple lines, but maximum of 200 characters

Planning Alerts



ORACLE

17-21

Copyright © 2004, Oracle. 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. A single alert can be used for displaying many messages, providing that the available buttons are suitable for responding to the messages.

Create an alert for each combination of:

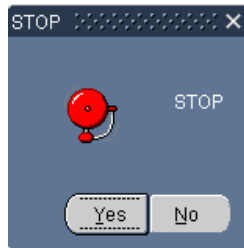
- Alert style required
- Set of available buttons (and labels) for operator response

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.

Instructor Note

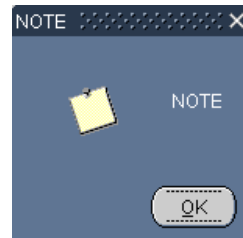
Because the properties for alert title, message, and labels for buttons can be set dynamically at run time, a module should require no more than three generic alerts.

Controlling Alerts



SET_ALERT_PROPERTY

SET_ALERT_BUTTON_PROPERTY



ORACLE

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 Forms Builder (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 single quoted string, if required.)

Controlling Alerts at Run Time (continued)

SET_ALERT_BUTTON_PROPERTY procedure

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

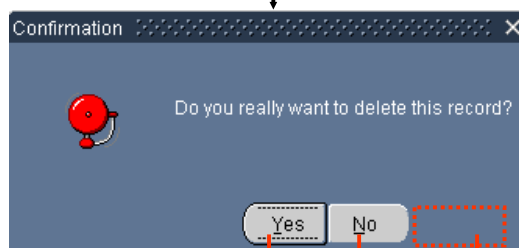
```
SET_ALERT_BUTTON_PROPERTY('alert_name', button,  
property, 'value')
```

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

SHOW_ALERT Function

```
IF SHOW_ALERT('del_Check')=ALERT_BUTTON1 THEN
```

```
. . .
```



Alert_Button1

Alert_Button2

Alert_Button3

ORACLE

17-24

Copyright © 2004, Oracle. 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 clicked in response to the alert. These numbers correspond to the values of three PL/SQL constants, which are predefined by the Forms Builder:

If the number equals...	The operator selected...
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 clicked by comparing the returned value against the corresponding constants.

SHOW_ALERT Function (continued)

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;
```

Instructor Note

Demonstration

Write a trigger that displays your earlier alert.

Open the `Show_Alert.fmb` file that demonstrates creating a generic alert. Run the form.

1. Set title, message, and button labels to any value you want.
2. Click the Show the Alert button.
The button number that you clicked appears in the text item.
3. Show the underlying code in this form.

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;
```

ORACLE

17-26

Copyright © 2004, Oracle. All rights reserved.

Directing Errors to an Alert

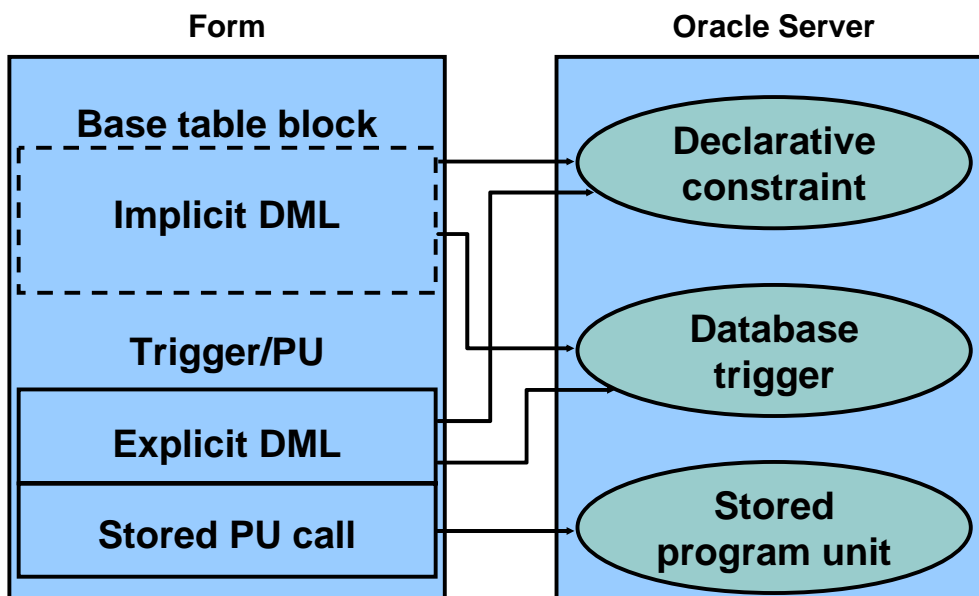
You may want to display errors automatically in an alert, through an On-Error trigger. The built-in functions that return error information, such as `ERROR_TEXT`, can be used in the `SET_ALERT_PROPERTY` procedure, to construct the alert message for display.

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;
```

Note: If you want the trigger to fail, include a call to `RAISE form_trigger_failure`.

Causes of Oracle Server Errors



ORACLE

17-27

Copyright © 2004, Oracle. All rights reserved.

Handling Errors Raised by the Oracle Database Server

Oracle server errors can occur for many different reasons, such as violating a declarative constraint or encountering a stored program unit error. You should know how to handle errors that may occur in different situations.

Causes of Oracle Server Errors

Cause	Error Message
Declarative constraint	Causes predefined error message
Database trigger	Error message specified in RAISE_APPLICATION_ERROR
Stored program unit	Error message specified in RAISE_APPLICATION_ERROR

Instructor Note

Demonstration: Use `ServerSide.fmb` to show the handling of errors originating from a trigger, from a stored procedure, and from the database server. Click the **HELP** button in the form to get instructions on running the demonstration. You can also find instructions in `ServerSideDemo.txt`.

Handling Errors Raised by the Oracle Database Server (continued)

Types of DML statements

Declarative-constraint violations and firing of database triggers are in turn caused by DML statements. For error-handling purposes, you must distinguish between the following two types of DML statements:

Type	Description
Implicit DML	DML statements that are associated with base table blocks. Implicit DML is also called base table DML. By default, Forms constructs and issues these DML statements.
Explicit DML	DML statements that a developer explicitly codes in triggers or program units.

FRM-Error messages caused by implicit DML errors

If an implicit DML statement causes an Oracle server error, Forms displays one of these FRM-error messages:

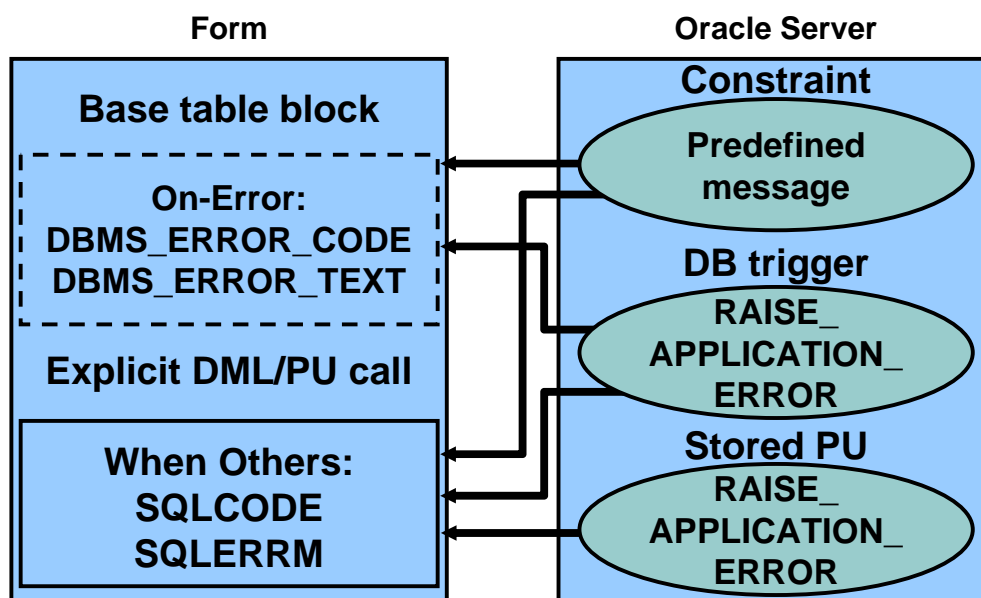
- FRM-40508: ORACLE error: unable to INSERT record.
- FRM-40509: ORACLE error: unable to UPDATE record.
- FRM-40510: ORACLE error: unable to DELETE record.

You can use `ERROR_CODE` to trap these errors in an On-Error trigger and then use `DBMS_ERROR_CODE` and `DBMS_ERROR_TEXT` to determine the ORA-error code and message.

FRM-Error messages with Web-deployed Forms

Users may receive a generic FRM-99999 error. You can obtain meaningful information about this error from the JInitiator Control Panel.

Trapping Server Errors



ORACLE

17-29

Copyright © 2004, Oracle. All rights reserved.

How to Trap Different Types of Oracle Database Server Errors

Type	Error Handling
Implicit DML	Use the Forms built-ins DBMS_ERROR_CODE and DBMS_ERROR_TEXT in an On-Error trigger.
Explicit DML	Use the PL/SQL functions SQLCODE and SQLERRM in a WHEN OTHERS exception handler of the trigger or program that issued the DML statements.
Stored program unit	Use the PL/SQL functions SQLCODE and SQLERRM in a WHEN OTHERS exception handler of the trigger or program that called the stored program unit.

Note: Declarative-constraint violations and database triggers may be caused by both implicit DML and explicit DML. Stored program units are always called explicitly from a trigger or program unit.

Technical Note

The values of DBMS_ERROR_CODE and DBMS_ERROR_TEXT are the same as what a user would see after selecting Help > Display Error; the values are not automatically reset following successful execution.

Summary

In this lesson, you should have learned that:

- **Forms displays messages at run time to inform the operator of events that occur in the session.**
- **You can use `FORM_SUCCESS` to test for run-time failure of built-ins.**
- **There are four types of Forms messages:**
 - Informative
 - Error
 - Working
 - Application

ORACLE

17-30

Copyright © 2004, Oracle. All rights reserved.

Summary

In this lesson, you should have learned 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.

- Test for failure of built-ins by using the `FORM_SUCCESS` built-in function or other built-in functions.
- Forms messages can be either Informative, Error, Working, or Application messages.

Summary

- **You can control system messages with built-ins and triggers:**
 - MESSAGE_LEVEL
 - SUPPRESS_WORKING
 - On-[Error | Message] triggers
 - [ERROR | MESSAGE]_[CODE | TEXT | TYPE]
- **Types of alerts: Stop, Caution, Note**
- **Alert built-ins:**
 - SHOW_ALERT
 - SET_ALERT_PROPERTY
 - SET_ALERT_BUTTON_PROPERTY

ORACLE

17-31

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- 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 and On-Message triggers intercept system error messages and informative system messages.
- You can use the built-ins ERROR_CODE, ERROR_TEXT, ERROR_TYPE, MESSAGE_CODE, MESSAGE_TEXT, or MESSAGE_TYPE to obtain information about the number, text, and type of errors and messages.
- Alert types: Stop, Caution, and Note
- Up to three buttons are available for response (NULL indicates no button).
- Display alerts and change alert messages at run time with SHOW_ALERT and SET_ALERT_PROPERTY.

Summary

- **Handle database server errors:**
 - **Implicit DML: Use `DBMS_ERROR_CODE` and `DBMS_ERROR_TEXT` in On-Error trigger**
 - **Explicit DML: Use `SQLCODE` and `SQLERRM` in `WHEN OTHERS` exception handler**

ORACLE

17-32

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- Intercept and handle server-side errors.

Practice 17 Overview

This practice covers the following topics:

- **Using an alert to inform the operator that the customer's credit limit has been exceeded**
- **Using a generic alert to ask the operator to confirm that the form should terminate**

ORACLE

17-33

Copyright © 2004, Oracle. All rights reserved.

Practice 17 Overview

In this practice, you create some alerts. These include a general alert for questions and a specific alert that is customized for credit limit.

- Using an alert to inform the operator that the customer's credit limit has been exceeded
- Using a generic alert to ask the operator to confirm that the form should terminate

Note: For solutions to this practice, see Practice 17 in Appendix A, "Practice Solutions."

Practice 17

1. Create an alert in CUSTGXX called Credit_Limit_Alert with one OK button. The message should read “This customer's current orders exceed the new credit limit”.
2. Alter the When-Radio-Changed trigger on Credit_Limit to show the Credit_Limit_Alert instead of the message when a customer’s credit limit is exceeded.
3. Save and compile the form. Click Run Form to run the form and test the changes.
4. Create a generic alert in ORDGXX called Question_Alert that allows Yes and No replies.
5. Alter the When-Button-Pressed trigger on CONTROL.Exit_Button to use the Question_Alert to ask the operator to confirm that the form should terminate. (You can import the text from pr17_4.txt.)
6. Save and compile the form. Click Run Form to run the form and test the changes.

18

Query Triggers

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	45 minutes	Lecture
	30 minutes	Practice
	75 minutes	Total

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 screen query conditions**
- **Write triggers to supplement query results**
- **Control trigger action based on the form's query status**

ORACLE

18-2

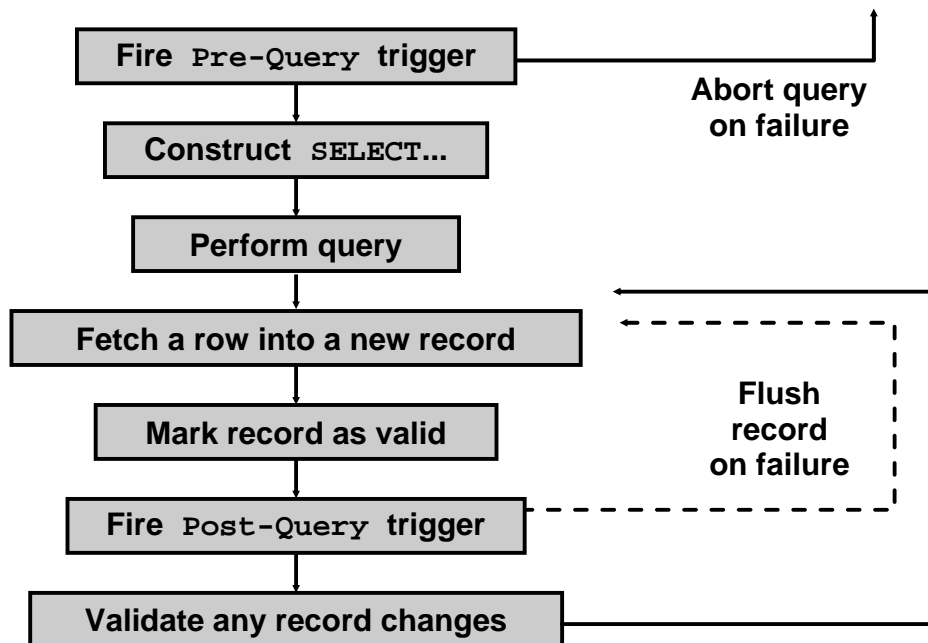
Copyright © 2004, Oracle. All rights reserved.

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 Overview



Query Processing Overview

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 Forms processing of records, or possibly abandon a query before it is even executed, if the required conditions are not suitable.

Forms 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, Forms fires the Pre-Query trigger if defined.
2. If the Pre-Query succeeds, Forms constructs the query SELECT statement, based on any existing criteria in the block (either entered by the operator or by the Pre-Query).

Query Processing Overview (continued)

3. The query is performed.
4. Forms 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. Forms fires the `Post-Query` trigger. If it fails, this record is flushed from the block.
7. Forms performs item and record validation if the record has changed (due to a trigger).
8. Steps 4 through 7 are repeated for any remaining records of this query.

SELECT Statements Issued During Query Processing

```
SELECT      base_column, ..., ROWID
INTO        :base_item, ..., :ROWID
FROM        base_table
WHERE       (default_where_clause OR
            onetime_where_clause)

            AND      (example_record_conditions)
            AND      (query_where_conditions)
ORDER BY    default_order_by_clause |
            query_where_order_by
```

Slightly different for COUNT

ORACLE

18-5

Copyright © 2004, Oracle. All rights reserved.

SELECT Statements Issued During Query Processing

If you have not altered default query processing, Forms 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 OR onetime_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 OR onetime_where_clause)
AND        (example_record_conditions)
AND        (query_where_conditions)
ORDER BY    default_order_by_clause | query_where_order_by
```

SELECT Statements Issued During Query Processing (continued)

Note: The vertical bar (|) in the ORDER BY clause indicates that either of the two possibilities can be present. Forms 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 in the example record and press Count Query.

WHERE Clause

- **Four sources for the WHERE clause:**
 - WHERE Clause block property
 - ONETIME_WHERE block property
 - Example Record
 - Query/Where dialog box
- **WHERE clauses are combined by the AND operator, except that WHERE and ONETIME_WHERE are combined with the OR operator.**

ORACLE

18-7

Copyright © 2004, Oracle. All rights reserved.

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.

Four sources for the WHERE clause

- The WHERE Clause block property (set in Forms Builder, or by setting the DEFAULT_WHERE_CLAUSE property programmatically)
- The ONETIME_WHERE block property (set programmatically)
- 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. If the WHERE clause and the ONETIME_WHERE clause are present, only one is used: the ONETIME_WHERE clause for the first execution of the query, and the WHERE clause for subsequent executions.

ONETIME_WHERE Property

The screenshot shows the PL/SQL Editor with the following trigger code:

```

Name: WHEN-BUTTON-PRESSED
Type: Trigger
Object: CONTROL
BLOCK: STOCK_BUTTON

SET_BLOCK_PROPERTY('INVENTORIES', ONETIME_WHERE,
'product_id='||:ORDER_ITEMS.product_id);
GO_BLOCK('INVENTORIES');
EXECUTE_QUERY;
    
```

The main form window shows the 'Stock' button and a table of items. The 'Product Id' field is set to 2322. Below the main form, two 'Inventory' windows are shown. The first window shows inventory for product 2322, and the second window shows inventory for product 1733. Red boxes highlight the product IDs in both windows, and red arrows point from the trigger code to these boxes.

Initially shows restricted query

2nd Execute_Query not restricted

ORACLE

18-8

Copyright © 2004, Oracle. All rights reserved.

WHERE and ORDER BY Clauses (continued)

ONETIME_WHERE property:

For instances where you want to restrict the query only once, you can programmatically set the ONETIME_WHERE property on a block. This specifies a WHERE clause for the block that will be in effect only for the first query issued on the block after setting that property.

Example:

From the ORDER_ITEMS block, you want to display the INVENTORIES block, which is on a separate window. When the block is initially displayed, it should present information about the stock of the product selected in the ORDER_ITEMS block. However, once this initial information is displayed, you want users to be able to query the stock of any product. You accomplish this by coding the Stock button to set the ONETIME_WHERE property:

```

Set_Block_Property('INVENTORIES', ONETIME_WHERE,
'product_id = '||:ORDER_ITEMS.PRODUCT_ID);
Go_block('INVENTORIES');
Execute_Query;
    
```

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**

ORACLE

18-9

Copyright © 2004, Oracle. All rights reserved.

WHERE and ORDER BY Clauses (continued)

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 use the SET_BLOCK_PROPERTY built-in to change the WHERE Clause and ORDER BY Clause block properties at run time.

Writing Query Triggers: Pre-Query Trigger

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

```
IF TO_CHAR(:ORDERS.ORDER_ID) ||
   TO_CHAR(:ORDERS.CUSTOMER_ID)
IS NULL THEN
    MESSAGE('You must query by
    Order ID or Customer ID');
    RAISE form_trigger_failure;
END IF;
```

ORACLE

18-10

Copyright © 2004, Oracle. All rights reserved.

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, just before Forms executes the query. You can use Pre-Query to:

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

Example

The Pre-Query trigger on the ORDERS block shown above permits queries only if there is a restriction on either the Order ID or Customer ID. This prevents very large queries.

Note: Pre-Query is useful for assigning values passed from other Oracle Forms Developer modules, so that the query is related to data elsewhere in the session. You will learn how to do this in a later lesson.

Instructor Note

With a restricted query, this trigger code can cause an error. For example, entering >10 for the ID item causes an error because the TO_CHAR in the trigger code around the ID item. Using NAME_IN in the code ensures that the correct datatype is used.

Writing Query Triggers: Post-Query Trigger

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

```
SELECT    COUNT(order_id)
INTO      :ORDERS.lineitem_count
FROM      ORDER_ITEMS
WHERE     order_id = :ORDERS.order_id;
```

ORACLE

18-11

Copyright © 2004, Oracle. All rights reserved.

Writing Query Triggers: 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 fires only 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

The Post-Query trigger on the ORDERS block, shown above, 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.

Writing Query Triggers: Using `SELECT` Statements in Triggers

- **Forms 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.**

ORACLE

18-12

Copyright © 2004, Oracle. All rights reserved.

Writing Query Triggers: 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 Forms 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**

ORACLE

18-13

Copyright © 2004, Oracle. All rights reserved.

Query Array Processing

The default behavior of Forms 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.

Forms 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 Edit > 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 Processing (continued)

Query Array Size Property: This property specifies the maximum number of records that Forms should fetch from the database at one time. If set to zero, the query array size defaults to the number of records displayed in the block.

A size of 1 provides the fastest perceived response time, because Forms 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 Triggers for Enter-Query Mode

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

ORACLE

18-15

Copyright © 2004, Oracle. 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 Forms fires a trigger if the associated event occurs in Enter-Query mode. Not all triggers can do this; consult Forms 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.

Coding Triggers for Enter-Query Mode (continued)

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 should 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;
```

To create a trigger that fires in Enter-Query mode, perform the following steps:

- In the Object Navigator, select a trigger.
- In the Property Palette, set the Fire in Enter-Query Mode property to Yes.

Instructor Note

The following triggers may fire in Enter-Query mode:

- Key-
- On-Error
- On-Message
- When- triggers, except:
 - When-Database-Record
 - When-Image-Activated
 - When-New-Block-Instance
 - When-New-Form-Instance
 - When-Create-Record
 - When-Remove-Record
 - When-Validate-Record
 - When-Validate-Item

Coding Triggers 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.**

ORACLE

18-17

Copyright © 2004, Oracle. All rights reserved.

Coding Triggers for Enter-Query Mode (continued)

Finding out the current mode

When a trigger fires 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 <code>:SYSTEM.MODE</code>	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 Forms is currently performing a fetch. (For example, this value always occurs in a Post-Query trigger.)

Coding Triggers for Enter-Query Mode (continued)

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 Forms 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

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	

ORACLE

18-19

Copyright © 2004, Oracle. 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 call “Do-the-right-thing” built-ins from transactional triggers to augment default query processing. The “Do-the-right-thing” built-ins perform the same actions as the default processing would. You can then supplement the default processing with your own code.

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.**

ORACLE

18-20

Copyright © 2004, Oracle. All rights reserved.

Overriding Default Query Processing (continued)

Using 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.

Instructor Note

When the `CREATE_QUERIED_RECORD` built-in is called from an On-Fetch trigger, it creates a record on the block waiting list. The waiting list is an intermediary record buffer that contains records that have been fetched from the data source, but have not yet been placed on the block list of active records. This built-in is included primarily for applications using transactional triggers to run against a data source other than Oracle.

Overriding Default Query Processing (continued)

Transactional Triggers for query processing have the following characteristics:

Trigger	Characteristic
On-Close	Fires when Forms closes a query (It augments, rather than replaces, default processing.)
On-Count	Fires when Forms would usually perform default Count Query processing to determine the number of rows that match the query conditions.
On-Fetch	Fires when Forms performs a fetch for a set of rows (You can use the <code>CREATE_QUERIED_RECORD</code> built-in to create queried records if you want to replace default fetch processing.) The trigger continues to fire until: <ul style="list-style-type: none">• No queried records are created during a single execution of the trigger• The query is closed by the user or by the <code>ABORT_QUERY</code> built-in executed from another trigger• The trigger raises <code>FORM_TRIGGER_FAILURE</code>
Pre-Select	Fires after Forms has constructed the block <code>SELECT</code> statement based on the query conditions, but before it issues this statement
On-Select	Fires when Forms would usually issue the block <code>SELECT</code> statement (The trigger replaces the open cursor, parse, and execute phases of a query.)
Post-Select	Fires after Forms has constructed and issued the block <code>SELECT</code> statement, but before it fetches the records

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`

ORACLE

18-22

Copyright © 2004, Oracle. All rights reserved.

Obtaining Query Information at Run Time

You can 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`. This system variable was discussed previously 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 Forms. 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 Forms has implicitly executed the `SELECT_RECORDS` built-in, the `SELECT` statement contains bind variables (for example, `ORDER_ID = :1`). If used after Forms has implicitly executed the `SELECT_RECORDS` built-in, the `SELECT` statement contains the actual search values (for example, `ORDER_ID=102`). The system variable contains bind variables during the Pre-Select trigger and actual search values during the Post-Select trigger.

Unlike most system variables, `SYSTEM.LAST_QUERY` may contain a mixture of upper and lower case.

Obtaining Query Information at Run Time

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

ORACLE

18-23

Copyright © 2004, Oracle. All rights reserved.

Obtaining Query Information at Run Time (continued)

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

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**

ORACLE

18-24

Copyright © 2004, Oracle. All rights reserved.

Obtaining Query Information at Run Time (continued)

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**

Instructor Note

QUERYABLE determines if the item can be included in a query against the base table of the block to which the item belongs.

QUERY_ONLY specifies that an item can be queried but that it should not be included in any **INSERT** or **UPDATE** statement that Forms issues for the block at run time.

Summary

In this lesson, you should have learned that:

- **Query processing includes the following steps:**
 1. Pre-Query trigger fires
 2. SELECT statement constructed
 3. Query performed
 4. Record fetched into block
 5. Record marked Valid
 6. Post-Query trigger fires
 7. Item and record validation if the record has changed (due to a trigger)
 8. Steps 4 through 7 repeat till all fetched

ORACLE

18-25

Copyright © 2004, Oracle. All rights reserved.

Summary

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

- **The query process:** Prior to beginning the query, the Pre-Query trigger fires once for each query. Then the query statement is constructed and executed. For each record retrieved by the query, the record is fetched into the block and marked as valid, the Post-Query trigger fires for that record, and item and record validation occurs if a trigger has changed the record.

Summary

- **The query triggers, which must be defined at block or form level, are:**
 - **Pre-Query:** Use to screen query conditions (set `ONETIME_WHERE` or `DEFAULT_WHERE` properties, or assign values to use as query criteria)
 - **Post-Query:** Use to supplement query results (populate nonbase table items, perform calculations)
- **You can use transactional triggers to override default query processing.**
- **You can control trigger action based on the form's query status by checking `SYSTEM.MODE` values: `NORMAL`, `ENTER-QUERY`, or `QUERY`**

ORACLE

18-26

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- 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 transactional triggers; to replace the default functionally, use “Do-the-right-thing” built-ins.
- Obtain query information at run-time by using:
 - `SYSTEM.MODE`, `SYSTEM.LAST_QUERY`
 - Some properties of `GET/SET_BLOCK_PROPERTY` and `GET/SET_ITEM_PROPERTY`

Practice 18 Overview

This practice covers the following topics:

- **Populating customer names and sales representative names for each row of the `ORDERS` block**
- **Populating descriptions for each row of the `ORDER_ITEMS` block**
- **Restricting the query on the `INVENTORIES` block for only the first query on that block**
- **Disabling the effects of the Exit button and changing a radio group in Enter-Query mode**
- **Adding two check boxes to enable case-sensitive and exact match query**

ORACLE

18-27

Copyright © 2004, Oracle. All rights reserved.

Practice 18 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 `ORDERS` block
- Populating descriptions for each row of the `ORDER_ITEMS` block
- Restricting the query on the `INVENTORIES` block for only the first query on that block
- Disabling the effect of the Exit button and changing a radio group in Enter-Query mode
- Adding two check boxes to the Customers form to enable case-sensitive and exact match query

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

Practice 18

1. In the ORDGXX form, write a trigger that populates the Customer_Name and the Sales_Rep_Name for every row fetched by a query on the ORDERS block. You can import the text from pr18_1.txt.
2. Write a trigger that populates the Description for every row fetched by a query on the ORDER_ITEMS block. You can import the text from pr18_2.txt.
3. Change the When-Button-Pressed trigger on the Stock_Button in the CONTROL block so that users will be able to execute a second query on the INVENTORIES block that is not restricted to the current Product_ID in the ORDER_ITEMS block. You can import the text from pr18_3.txt.
4. Ensure that the Exit_Button has no effect in Enter-Query mode.
5. Click Run Form to run the form and test it.
6. Open the CUSTGXX form module. Adjust the default query interface. 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. Place the check box on the Name page of the TAB_CUSTOMER canvas. You can import the pr18_6.txt file into the When-Checkbox-Changed trigger. Set the initial value property to Y, and the Checked/Unchecked properties to Y and N. Set the Mouse Navigate property to No.
7. 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.) Set the label to: Exact match on query? Set the initial value property to Y, and the Checked/Unchecked properties to Y and N. Set the Mouse Navigate property to No. You can import the pr18_7.txt file into the Pre-Query Trigger.
8. Ensure that the When-Radio-Changed trigger for the Credit_Limit item does not fire when in Enter-Query mode.
9. Click Run Form to run the form and test the changes.

19

Validation

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	35 minutes	Lecture
	30 minutes	Practice
	65 minutes	Total

Objectives

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

- **Explain the effects of the validation unit upon a form**
- **Control validation:**
 - Using object properties
 - Using triggers
 - Using Pluggable Java Components
- **Describe how Forms tracks validation status**
- **Control when validation occurs**

ORACLE

19-2

Copyright © 2004, Oracle. All rights reserved.

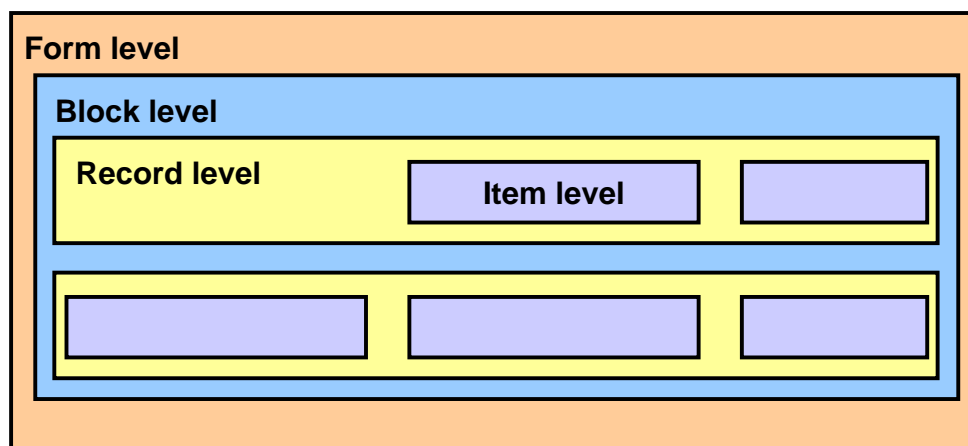
Introduction

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.

The Validation Process

Forms validates at the following levels:



Validation Process

Validation Levels

Forms 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:** Forms 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 Forms 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)

The Validation Process

Validation occurs when:

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

ORACLE

19-4

Copyright © 2004, Oracle. All rights reserved.

Validation Process (continued)

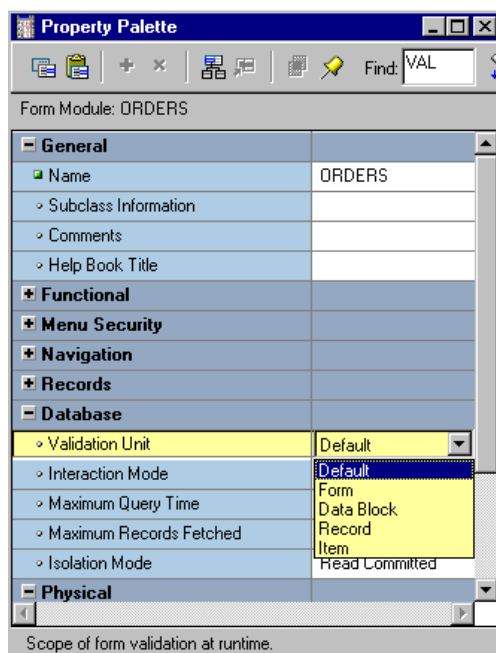
- **Record level:** After leaving a record, Forms checks to see whether the record is valid. If not, then the status of each item in the record is checked, and a When-Validate-Record trigger is then fired, if present. When the record passes these checks, it is set to valid.
- **Block and form level:** 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?

Forms carries out validation for the validation unit under the following conditions:

- The [Enter] key is pressed or the ENTER built-in procedure is run. The purpose of the ENTER built-in is to force validation immediately.
Note: The ENTER action is not necessarily mapped to the key that is physically labeled Enter.
- 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.

Controlling Validation Using Properties: Validation Unit



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 Forms 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 Forms 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.

Using Object Properties to Control Validation (continued)

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.

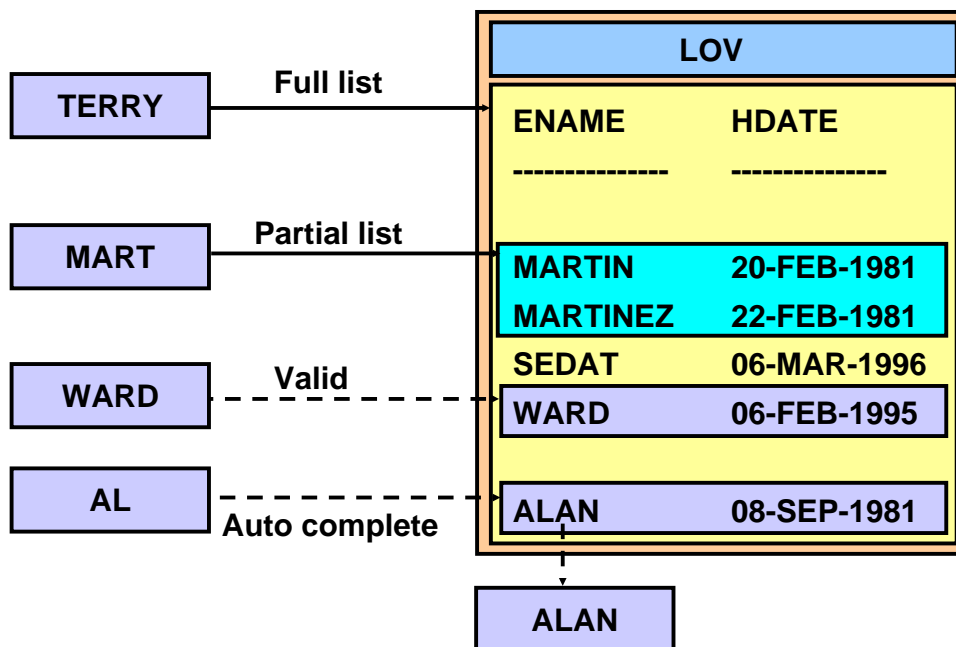
You might set a validation unit above item level under one of the following conditions:

- Validation involves database references, and you want to postpone traffic until the operator has completed a record (record level).
- The application runs on the Internet and you want to improve performance by reducing round trips to the application server.

Instructor Note

Show the Validation Unit property in Forms Builder. You may want to set Validation Unit to a higher level, and demonstrate its effect at run time.

Controlling Validation Using Properties: Validate from List



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.

The Validate from List Property

Do this by setting the Validate from List property to Yes for the item. At validation time, Forms 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, Forms 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.

Using LOVs for Validation (continued)

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.

Instructor Note

Set the Validate from List property on an item to which you have already attached a suitable LOV. Then, build and run the form to show the use of this feature.

Controlling Validation Using Triggers

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

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

ORACLE

19-9

Copyright © 2004, Oracle. All rights reserved.

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

The When-Validate-Item trigger on ORDERS.Order_Date, shown above, ensures that the Order Date is not later than the current (database) date.

Controlling Validation by Using Triggers (continued)

When-Validate-Record Trigger

This trigger fires after standard record-level validation, when the operator has left a new or changed record. Because Forms 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 ORDER_ITEMS warns the operator when a line item for a new credit order causes the customer's credit limit to be exceeded.

```
DECLARE
    cred_limit number;
    n number;
BEGIN
    -- Order status of 4 is new credit order
    IF :orders.order_status = 4 THEN
        SELECT credit_limit INTO cred_limit FROM customers
            WHERE customer_id = :orders.customer_id;
        IF :control.total > cred_limit THEN
            n := show_alert('credit_limit_alert');
        END IF;
    END IF;
END;
```

Note: If you want to stop the operator from navigating out of the item when validation fails, you can raise an exception to fail the trigger:

```
RAISE form_trigger_failure;
```

In the example above, you would include this code immediately after displaying the alert.

Instructor Note

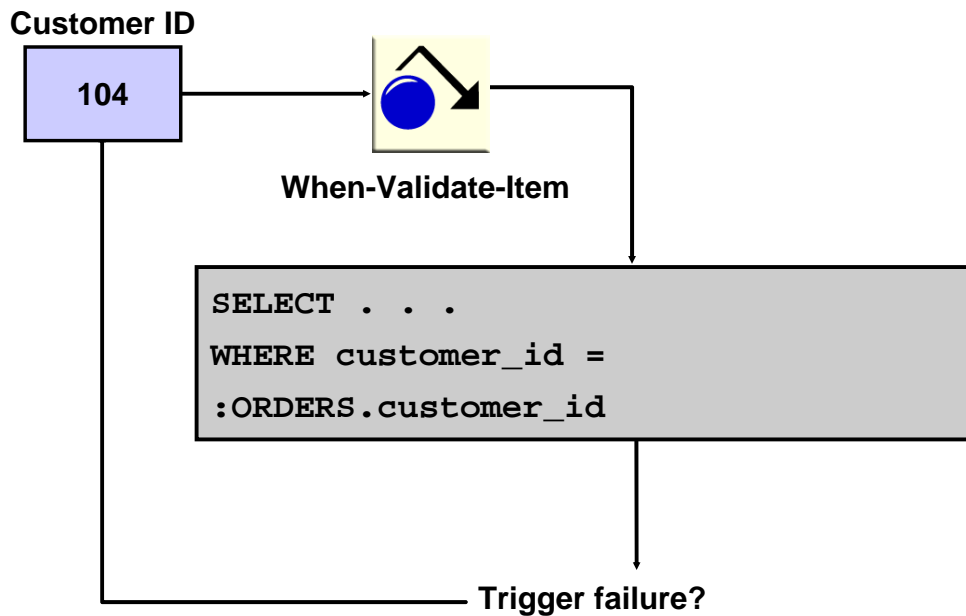
Introduce When-Validate-Record as an additional validation trigger.

Students who have experience with early versions of the product (SQL*Forms V2.3) sometimes ask about using Post-Change as a validation trigger.

Post-Change does not fire if an item is set to Null, and it can fire on population at query time, making it less specific to validation than the When-Validate-*<object>* triggers.

The Post-Change trigger is included only for compatibility with previous versions of Forms Builder. Its use is not recommended in new applications.

Example: Validating User Input



ORACLE

19-11

Copyright © 2004, Oracle. All rights reserved.

Controlling Validation by Using Triggers (continued)

Example: 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 the Name that correspond to the `Customer ID` entered by the user.

```
SELECT cust_first_name || ' ' || cust_last_name
INTO :ORDERS.customer_name
FROM customers
WHERE customer_id = :ORDERS.customer_id;
```

Controlling Validation by Using Triggers (continued)

Example: Validating user input (continued)

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>
```

Using Client-Side Validation

- **Forms validation:**
 - Occurs on middle tier
 - Involves network traffic
- **Client-side validation:**
 - Improves performance
 - Implemented with PJC

Line	Product	Item Id	Id	Description	Unit Price	Quantity
1	2395	32MB Cache /M	123	abcdefg	48	10
2	2289	KB 101.ES			48	20
3	3106	KB 101.EN			48	

Using number datatype

FRM-50016: Legal characters are 0-9 - + E .

Attempt to enter alphabetic characters

Line	Product	Item Id	Id	Description	Unit Price	Quantity
1	2395	32MB Cache /M	123		48	10
2	2289	KB 101.ES			48	20
3	3106	KB 101.EN			48	

Using KeyFilter PJC

Enter a numeric value

ORACLE

19-13

Copyright © 2004, Oracle. All rights reserved.

Using Client-Side Validation

It is common practice to process input to an item using a `When-Validate-Item` trigger. The trigger itself is processed on the Forms Services. Even validation that occurs with a format mask on an item involves a trip to the middle tier. In the first example in the slide, the number data type on the Quantity item is not checked until the operator presses [Enter] to send the input to the Forms Services machine, which returns the FRM-50016 error.

You should consider using Pluggable Java Components (PJC) to replace the default functionality of standard client items, such as text boxes. Then validation of items, such as the date or maximum or minimum values, is contained within an item. This technique opens up opportunities for more complex, application-specific validation, like automatic formatting of input such as telephone numbers with format (XXX) XXX-XXXX. Even a simple numeric format is enforced instantly, not allowing alphabetic keystrokes to be entered into the item.

This validation is performed on the client without involving a network round trip, thus improving performance. In the second example above, the KeyFilter PJC does not allow the operator to enter an alphabetic character into the Quantity item. The only message that is displayed on the message line is the item's Hint.

Using Client-Side Validation (continued)

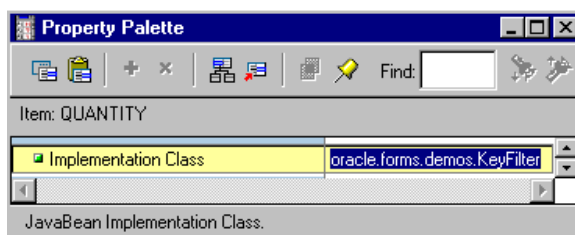
Pluggable Java Components are similar to JavaBeans, and in fact, the two terms are often used interchangeably. Although both are Java components that you can use in a form, there are the following differences between them:

- JavaBeans are implemented in a bean area item, whereas PJC's are implemented in a native Forms item such as a text item or check box.
- PJC's must always have the implementation class specified in the Property Palette, but JavaBeans may be registered at run time with the FBean package.

Using Client-Side Validation

To use a PJC:

1. Set the item's Implementation Class property



2. Set properties for the PJC

```
SET_CUSTOM_PROPERTY('order_items.quantity',  
                    1, 'FILTER_TYPE', 'NUMERIC');
```

ORACLE

19-15

Copyright © 2004, Oracle. All rights reserved.

Using Client-Side Validation (continued)

You implement a PJC to replace an item by setting the item's Implementation Class property to the class of the PJC. You may use the `SET_CUSTOM_PROPERTY` built-in to set properties of the PJC that restrict input or otherwise validate the item. At run time, Forms looks for the Java class contained on the middle tier or in the archive files with the path specified in the Implementation Class for the item. If you open `keyfilter.jar` in WinZip, you find that the path to `KeyFilter.class` is `oracle\forms\demos`.

You deploy the PJC as you would a JavaBean, which was discussed in Lesson 16. You can locate the Java class file:

- On the middle-tier server, either in the directory structure referenced by the form applet's `CODEBASE` parameter or in the server's `CLASSPATH`. `CODEBASE` is by default the `forms90\java` subdirectory of `ORACLE_HOME`.
- If using JInitiator, in a JAR file in the middle-tier server's `CODEBASE` directory, and included in the `ARCHIVE` parameter so that the JAR file is downloaded to and cached on the client. For example:

```
archive_jini=f90all_jinit.jar,keyfilter.jar
```

(The `CODEBASE` and `ARCHIVE` parameters are set in the `formsweb.cfg` file.)

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

ORACLE

19-16

Copyright © 2004, Oracle. All rights reserved.

Tracking Validation Status

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

Tracking Validation Status (continued)

Item Validation Status

Status	Definition
NEW	When a record is created, Forms 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	Forms marks an item as changed under the following conditions: <ul style="list-style-type: none">• 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	Forms marks an item as valid under the following conditions: <ul style="list-style-type: none">• All items in the 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, Forms 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, Forms marks that record as changed.
VALID	Forms marks a record as valid under the following conditions: <ul style="list-style-type: none">• 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 duplicate record inherits the status of its source

Controlling When Validation Occurs with Built-Ins

- **CLEAR_BLOCK, CLEAR_FORM, EXIT_FORM**
- **ENTER**
- **SET_FORM_PROPERTY**
 - (... , VALIDATION)
 - (... , VALIDATION_UNIT)
- **ITEM_IS_VALID** item property
- **VALIDATE (scope)**

ORACLE

19-18

Copyright © 2004, Oracle. All rights reserved.

Controlling When Validation Occurs with Built-Ins

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.

Controlling When Validation Occurs with Built-Ins (continued)

SET_FORM_PROPERTY

You can use this to disable Forms 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(scope)` forces Forms to immediately execute validation processing for the indicated scope.

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

Summary

In this lesson, you should have learned that:

- **The validation unit specifies how much data is entered before validation occurs.**
- **You can control validation using:**
 - **Object properties: Validation Unit (form); Validate from List (item)**
 - **Triggers: When-Validate-Item (item level); When-Validate-Record (block level)**
 - **Pluggable Java Components for client-side validation**

ORACLE

19-20

Copyright © 2004, Oracle. All rights reserved.

Summary

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

- Validation occurs at several levels: Item, Record, Block, and Form.
- Validation happens when:
 - The [Enter] Key is pressed 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.
- The When-Validate-“*object*” triggers supplement standard validation.
- You can use Pluggable Java Components to perform client-side validation.

Summary

- **Forms tracks validation status of items and records, which are either NEW, CHANGED, or VALID.**
- **You can use built-ins to control when validation occurs:**
 - CLEAR_BLOCK
 - CLEAR_FORM
 - EXIT_FORM
 - ENTER
 - ITEM_IS_VALID
 - VALIDATE

ORACLE

19-21

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- Forms tracks validation status internally: NEW, CHANGED, or VALID
- You can use built-ins to control when validation occurs.

Practice 19 Overview

This practice covers the following topics:

- **Validating the Sales Representative item value by using an LOV**
- **Writing a validation trigger to check that online orders are CREDIT orders**
- **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**
- **Restricting user input to numeric characters using a Pluggable Java Component**

ORACLE

19-22

Copyright © 2004, Oracle. All rights reserved.

Practice 19 Overview

In this practice, you introduce additional validation to the CUSTGXX and ORDGXX form modules.

- Validating sales representative item value by using an LOV
- Writing a validation trigger to check that all online orders are CREDIT orders
- 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
- Implementing client-side validation on the item quantity using a Pluggable Java Component

Note: For solutions to this practice, see Practice 19 in Appendix A, “Practice Solutions.”

Practice 19

1. In the CUSTGXX form, cause the Account_Mgr_Lov to be displayed whenever the user enters an Account_Mgr_Id that does not exist in the database.
2. Save and compile the form. Click Run Form to run the form and test the functionality.
3. In the ORDGXX form, write a validation trigger to check that if the Order_Mode is online, the Order_Status indicates a CREDIT order (values between 4 and 10). You can import the text from `pr19_3.txt`.
4. In the ORDGXX 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 data is not found. You can import text from `pr19_4a.txt` and `pr19_4b.txt`.
5. Create another validation trigger on ORDER_ITEMS.Product_Id to derive the name of the product and suggested wholesale price, and write them to the Description item and the Price item. Fail the trigger and display a message if the product is not found. You can import the text from `pr19_5.txt`.
6. Perform client-side validation on the ORDER_ITEMS.Quantity item using a Pluggable Java Component to filter the keystrokes and allow only numeric values. The full path to the PJC class is `oracle.forms.demos.KeyFilter` (this is case sensitive), to be used as the Implementation Class for the item. You will set the filter for the item in the next practice, so the validation is not yet functional.
7. Save and compile the form. Click Run Form to run the form and test the changes. Do not test the validation on the Quantity item because it will not function until after you set the filter on the item in Practice 20.

20

Navigation

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	45 minutes	Lecture
	20 minutes	Practice
	65 minutes	Total

Objectives

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

- **Distinguish between internal and external navigation**
- **Control navigation with properties**
- **Describe and use navigation triggers to control navigation**
- **Use navigation built-ins in triggers**

ORACLE

20-2

Copyright © 2004, Oracle. All rights reserved.

Introduction

Overview

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

Navigation Overview

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

ORACLE

20-3

Copyright © 2004, Oracle. All rights reserved.

Navigation Overview

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

Navigation Overview (continued)

Entering and Leaving Objects

During navigation, Forms 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.

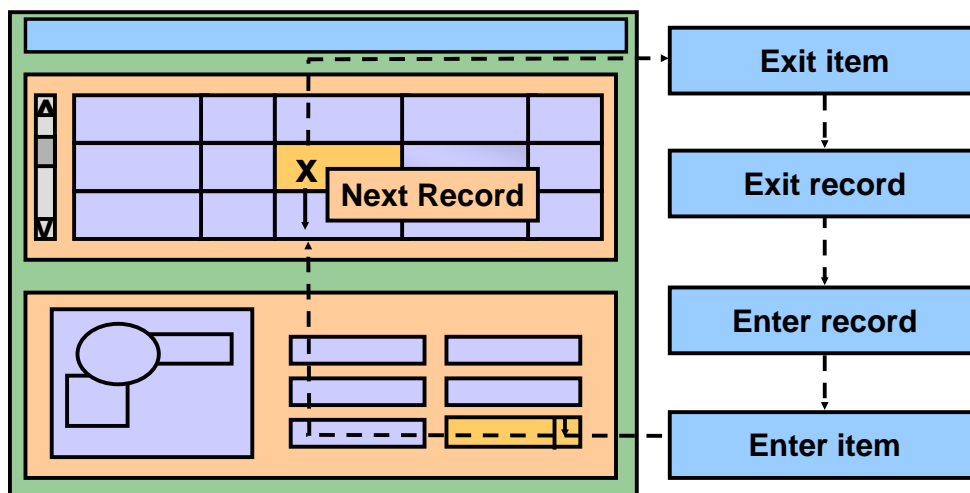
The Cursor and how it Relates to the Navigation Unit

The cursor is a visible, external object that indicates the current input focus. Forms 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, Forms 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 Forms cannot move the navigation unit back to its initial location, it exits the form.

Understanding Internal Navigation



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

Understanding Internal Navigation (continued)

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:

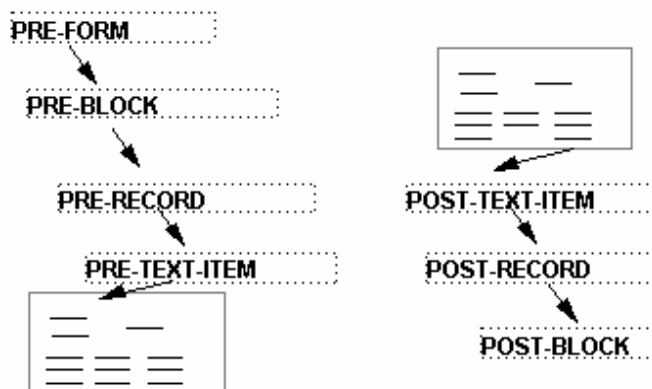
- Exit current item
- Exit current record
- Exit current block

Performance Note

Forms uses smart event bundling: All the events that are triggered by navigation between two objects are delivered as one packet to Forms Services on the middle tier for subsequent processing.

Instructor Note

Draw navigation examples of “on entry” and “on commit.”



Demonstration: Run the form `Navdemo.fmx`, which displays messages as navigational triggers fire. This illustrates the navigation that occurs behind the scenes when a form first opens.

Using Object Properties to Control Navigation

- **Block**
 - Navigation Style
 - Previous Navigation Data Block
 - Next Navigation Data Block
- **Item**
 - Enabled
 - Keyboard Navigable
 - Mouse Navigate
 - Previous Navigation Item
 - Next Navigation Item

ORACLE

20-7

Copyright © 2004, Oracle. All rights reserved.

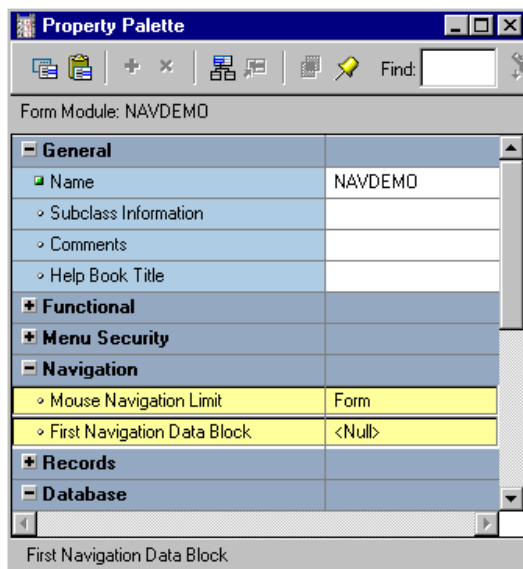
Controlling Navigation Using Object Properties

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:

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

Note: You can use the mouse to navigate to any enabled item regardless of its position in the navigational order.

Using Object Properties to Control Navigation



- **Form module**
 - **Mouse Navigation Limit**
 - **First Navigation Data Block**

ORACLE

20-8

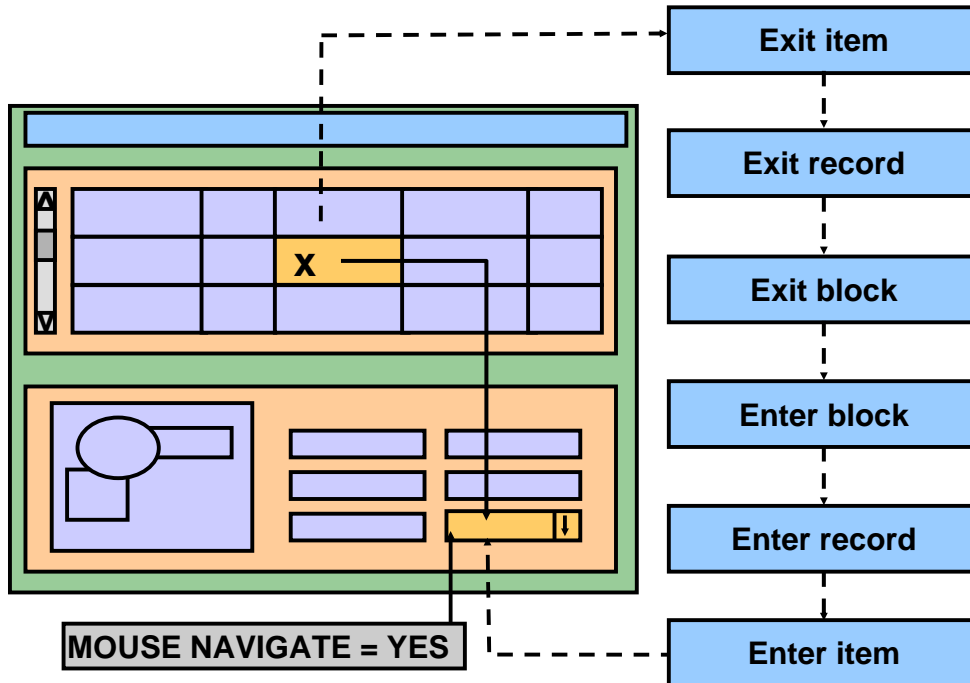
Copyright © 2004, Oracle. All rights reserved.

Controlling Navigation Using Object Properties (continued)

There are two other navigation properties that you can set for the form module: Mouse Navigation Limit and First Navigation Block.

Form Module Property	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 Forms should navigate on form startup (Setting this property does not override the order used for committing.)

Mouse Navigate Property



ORACLE

20-9

Copyright © 2004, Oracle. All rights reserved.

Mouse Navigate Property

The Mouse Navigate property is valid for the following items:

- Push Button
- Check box
- List item
- Radio group
- Hierarchical tree item
- Bean Area Item

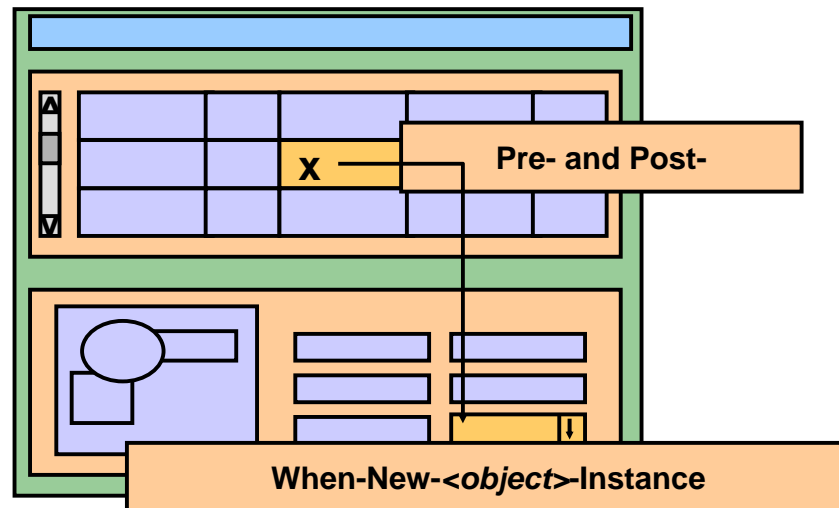
Note: The default setting for the Mouse Navigate property is Yes.

Setting	Use to Ensure That:
Yes	Forms navigates to the new item. (This causes the relevant navigational and validation triggers to fire.)
No	Forms does not navigate to the new item or validate the current item when the user activates the new item with the mouse.

Instructor Note

Point out that with Mouse Navigate set to No, none of the navigation events listed in the above slide occurs.

Writing Navigation Triggers



Writing 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, 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 Navigation Triggers not Fire?

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

Navigation Triggers

Pre- and Post-	When-New-<object>-Instance
Fire during navigation	Fire after navigation
Do not fire if validation unit is higher than trigger object	Fire even when validation unit is higher 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

ORACLE

20-11

Copyright © 2004, Oracle. All rights reserved.

Navigation Triggers

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.

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**

ORACLE

20-12

Copyright © 2004, Oracle. All rights reserved.

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 a form is run, after successful navigation into the 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

SET_<object>_PROPERTY Examples

```
SET FORM PROPERTY (FIRST_NAVIGATION_BLOCK,  
'ORDER_ITEMS');
```

```
SET BLOCK PROPERTY ('ORDERS', ORDER_BY,  
'CUSTOMER_ID');
```

```
SET RECORD PROPERTY (3, 'ORDER_ITEMS', STATUS,  
QUERY_STATUS);
```

```
SET ITEM PROPERTY ('CONTROL.stock_button',  
ICON_NAME, 'stock');
```

ORACLE

20-13

Copyright © 2004, Oracle. All rights reserved.

Initializing Forms Builder Objects

Use the When-New-<object>-Instance triggers, along with the SET_<object>_PROPERTY built-in subprograms to initialize Forms 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('ORDER_ITEMS',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;
```

Initializing Forms Builder Objects (continued)

Example

Register the Color Picker JavaBean into the Control.Colorpicker bean area item when the CUSTOMERS form is run by including the following code in your When-New-Form-Instance trigger:

```
FBean.Register_Bean('control.colorpicker',1,  
                    'oracle.forms.demos.beans.ColorPicker');
```

At run time, Forms looks for the Java class contained on the middle tier or in the archive files with the path specified in the code. If you open `colorpicker.jar` in WinZip, you find that the path to `ColorPicker.class` is `oracle\forms\demos\beans`.

The Pre- and Post-Triggers

- **Pre/Post-Form**
- **Pre/Post-Block**
- **Pre/Post-Record**
- **Pre/Post-Text-Item**

ORACLE

20-15

Copyright © 2004, Oracle. All rights reserved.

Using the Pre- and Post-Triggers

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.

Using the Pre- and Post-Triggers (continued)

Trigger Type	Use to
Pre-Form	<ul style="list-style-type: none">• Validate<ul style="list-style-type: none">– User– Time of day• Initialize control blocks• Call another form to display messages
Post-Form	<ul style="list-style-type: none">• Perform housekeeping, such as erasing global variables• Display messages to user before exit
Pre-Block	<ul style="list-style-type: none">• Authorize access to the block• Set global variables
Post-Block	<ul style="list-style-type: none">• Validate the last record that had input focus• Test a condition and prevent the user from leaving the block
Pre-Record	<ul style="list-style-type: none">• Set global variables
Post-Record	<ul style="list-style-type: none">• Clear global variables• Set a visual attribute for an item as the user scrolls through a set of records• Perform cross field validation
Pre-Text-Item	<ul style="list-style-type: none">• Derive a complex default value• Record the previous value of a text item
Post-Text-Item	<ul style="list-style-type: none">• Calculate or change item values

Post-Block Trigger Example

Disabling Stock button when leaving the `ORDER_ITEMS` block:

```
SET_ITEM_PROPERTY('CONTROL.stock_button',  
enabled, property_false);
```

ORACLE

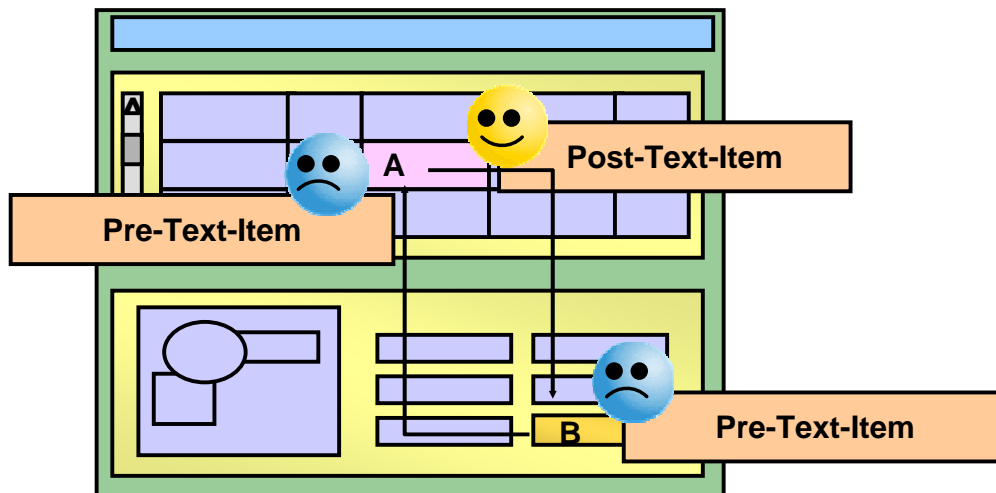
20-17

Copyright © 2004, Oracle. All rights reserved.

Instructor Note

Ask the students what type of trigger you would use to reenable the Stock button.

The Navigation Trap



ORACLE

20-18

Copyright © 2004, Oracle. 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 in the slide 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.

Instructor Note

Demonstration: The `navigation_trap.fmb` file illustrates the navigation trap. When you run the form and navigate to the 2nd item, its Pre-Text-Item trigger fails. Forms is not able to return input focus to the first item because its Pre-Text-Item trigger is set to fail the second time it fires.

Using Navigation Built-Ins in Triggers

GO_FORM
GO_BLOCK
GO_ITEM
GO_RECORD
NEXT_BLOCK
NEXT_ITEM
NEXT_KEY
NEXT_RECORD

NEXT_SET
UP
DOWN
PREVIOUS_BLOCK
PREVIOUS_ITEM
PREVIOUS_RECORD
SCROLL_UP
SCROLL_DOWN

ORACLE

20-19

Copyright © 2004, Oracle. All rights reserved.

Using Navigation Built-Ins in Triggers


You can initiate navigation programmatically by calling the built-in subprograms, such as GO_ITEM and PREVIOUS_BLOCK from triggers.

Built-Ins for Navigation	Function
GO_FORM	Navigates to an open form in a multiple form application
GO_BLOCK/ITEM/RECORD	Navigates to the indicated block, item, or record
NEXT_BLOCK/ITEM/KEY	Navigates to the next enterable block, item, or primary key item
NEXT/PREVIOUS_RECORD	Navigates to the first enterable item in the next or previous record
NEXT_SET	Fetches another set of records from the database and navigates to the first record that the fetch retrieves
UP, DOWN	Navigates to the instance of the current item in the previous/next record
PREVIOUS_BLOCK/ITEM	Navigates to the previous enterable block or item
SCROLL_UP/DOWN	Scrolls the block so that the records above the top visible one or below the bottom visible one display

Using Navigation Built-Ins in Triggers


- **When-New-Item-Instance**

```
IF CHECKBOX_CHECKED('ORDERS.order_mode') --Online
THEN                                     -- order
    ORDERS.order_status := 4; --Credit order
    GO_ITEM('ORDERS.order_status');
END IF;
```



- **Pre-Text-Item**

```
IF CHECKBOX_CHECKED('ORDERS.order_mode') --Online
THEN                                     -- order
    ORDERS.order_status := 4; --Credit order
    GO_ITEM('ORDERS.order_status');
END IF;
```



ORACLE

20-20

Copyright © 2004, Oracle. All rights reserved.

Using Navigation Built-Ins in Triggers (continued)

Calling built-ins from Navigational Triggers

You are not allowed to use a restricted built-in from within a trigger that fires during the navigation process (the Pre- and Post- triggers). This is because restricted built-ins perform some sort of navigation, and so cannot be called until Forms navigation is complete.

You can call restricted built-ins from triggers such as When-New-Item-Instance, because that trigger fires after Forms has moved input focus to the new item.

Instructor Note

Point out that the first example above correctly uses the When-New-Item-Instance trigger, whereas the second incorrectly uses the Pre-Text-Item trigger and contains a restricted built-in.

Summary

In this lesson, you should have learned that:

- **External navigation is visible to the user, while internal navigation occurs behind the scenes.**
- **You can control navigation with properties of the form, block, or item:**
 - **Set in Navigation category of the Property Palette**
 - OR**
 - **Use SET_[FORM | BLOCK | ITEM]_PROPERTY**

ORACLE

20-21

Copyright © 2004, Oracle. 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.

Summary

- **Navigation triggers:**
 - Those that fire during navigation (watch out for the navigation trap):
[Pre | Post] - [Form | Block | Item]
 - Those that fire after navigation:
When-New- [Form | Block | Record | Item] -Instance
- **You can use navigation built-ins in triggers (except for triggers that fire during navigation):**
 - GO_[FORM | BLOCK | RECORD | ITEM]
 - NEXT_[BLOCK | RECORD | ITEM | KEY | SET]
 - UP
 - DOWN
 - PREVIOUS_[BLOCK | RECORD | ITEM]
 - SCROLL_[UP | DOWN]

ORACLE

20-22

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- Navigation triggers:
 - When-New-<object>-Instance
 - Pre- and Post-
- Avoid the navigation trap.
- Navigation built-ins are available.

Practice 20 Overview

This practice covers the following topics:

- Registering the bean area's JavaBean at form startup
- Setting properties on a Pluggable Java Component at form startup
- Executing a query at form startup
- Populating product images when cursor arrives on each record of the `ORDER_ITEMS` block

ORACLE

20-23

Copyright © 2004, Oracle. All rights reserved.

Practice 20 Overview

In this practice, you provide a trigger to automatically perform a query, register a JavaBean, and set properties on a PJC at form startup. Also, you use `When-New-<object>-Instance` triggers to populate the `Product_Image` item as the operator navigates between records in the `ORDGXX` form.

- Executing a query at form startup
- Populating product images when the cursor arrives on each record of the `ORDER_ITEMS` block

Note: For solutions to this practice, see Practice 20 in Appendix A, “Practice Solutions.”

Practice 20

1. When the `ORDGXX` form first opens, set a filter on the `ORDER_ITEMS.Quantity` Pluggable Java Component, and execute a query. You can import the code for the trigger from `pr20_1.txt`.
2. Write a trigger that fires as the cursor arrives in each record of the `ORDER_ITEMS` block to populate the `Product_Image` item with a picture of the product, if one exists. First create a procedure called `get_image` to populate the image, then call that procedure from the appropriate trigger. You can import the code for the procedure from `pr20_2a.txt`.
3. Define the same trigger type and code on the `ORDERS` block.
4. Is there another trigger where you might also want to place this code?
5. Save and compile the form. Click `Run Form` to run the form and test the changes.
6. Notice that you receive an error if the image file does not exist. Code a trigger to gracefully handle the error by populating the image item with a default image called `blank.jpg`. You can import the code from `pr20_6.txt`.
7. The image item has a lot of blank space when the image does not take up the entire area. To make it look better, set its `Background Color` of both the `CONTROL.Product_Image` item and the `CV_ORDER` canvas to the same value, such as `r0g75b75`. Set the `Bevel` for the `Product_Image` item to `None`.
8. Click `Run Form` to run the form again and test the changes.
9. In the `CUSTGXX` form, register the `ColorPicker` bean (making its methods available to Forms) when the form first opens, and also execute a query on the `CUSTOMERS` block. You can import the code from `pr20_9.txt`.
10. Save, compile, and click `Run Form` to run the form and test the `Color` button. You should be able to invoke the `ColorPicker` bean from the `Color` button, now that the bean has been registered at form startup.

21

Transaction Processing

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	90 minutes	Lecture
	30 minutes	Practice
	120 minutes	Total

Objectives

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

- **Explain the process used by Forms to apply changes to the database**
- **Describe the commit sequence of events**
- **Supplement transaction processing**
- **Allocate sequence numbers to records as they are applied to tables**
- **Implement array DML**

ORACLE

21-2

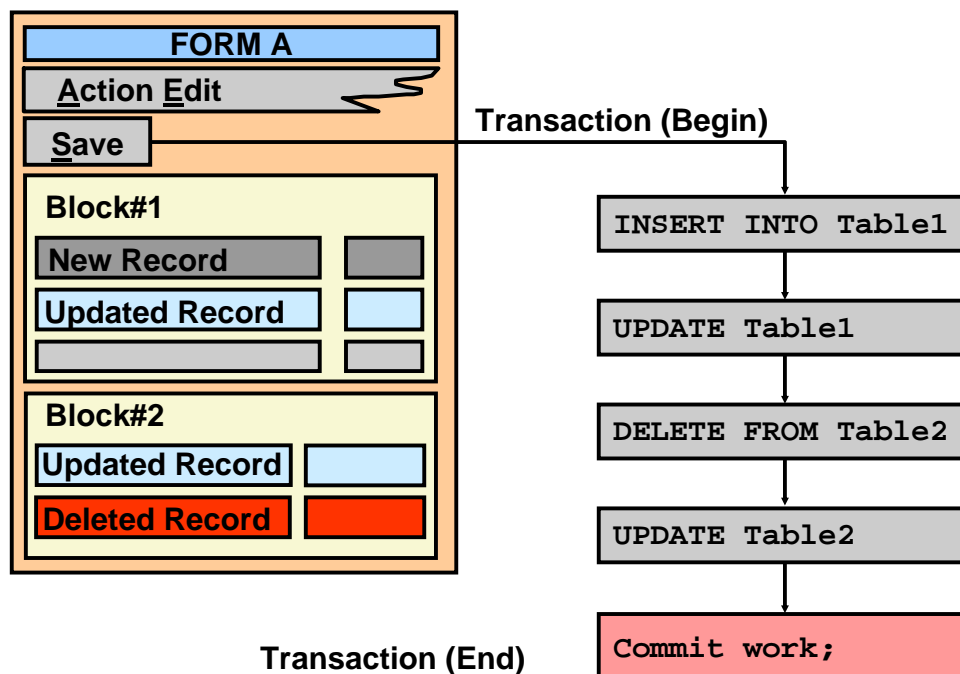
Copyright © 2004, Oracle. All rights reserved.

Introduction

Overview

While applying a user's changes to the database, Forms 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 Overview



Transaction Processing Overview

When Forms 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 Forms 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, Forms commits the transaction, making the changes permanent.

Transaction Processing Overview

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**

ORACLE

21-4

Copyright © 2004, Oracle. All rights reserved.

Transaction Processing Overview (continued)

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 Save 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 Forms. 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 Overview (continued)

Other events related to transactions include rollbacks, savepoints, and locking.

Rollbacks

Forms 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

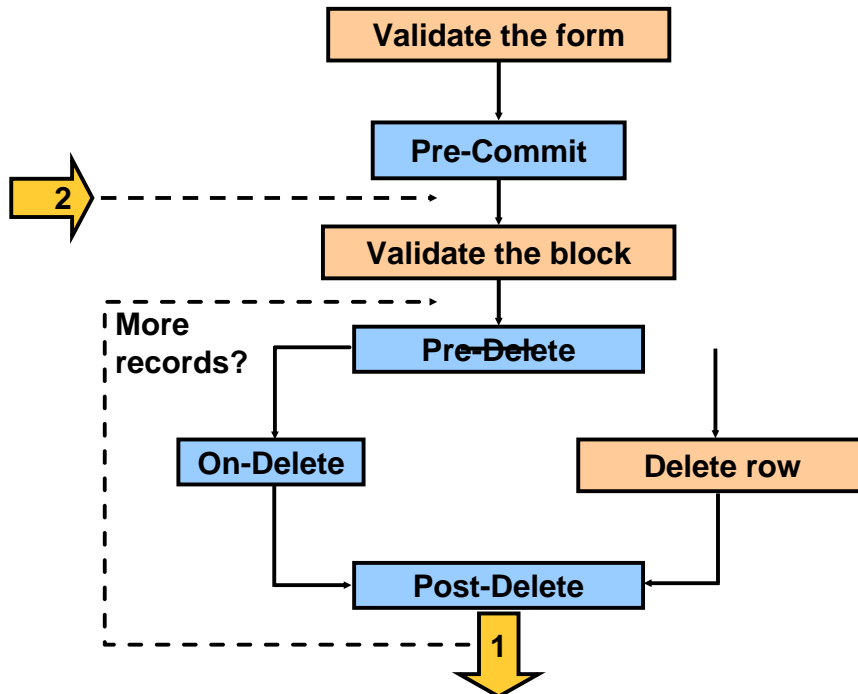
Forms issues savepoints in a transaction automatically, and will roll back to the latest savepoint if certain events occur. Generally, these savepoints are for Forms internal use, but certain built-ins, such as the `EXIT_FORM` built-in procedure, can request a rollback to the latest savepoint by using 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 Forms program unit, Forms treats `COMMIT` as the `COMMIT_FORM` built-in, and `ROLLBACK` as the `CLEAR_FORM` built-in.

The Commit Sequence of Events

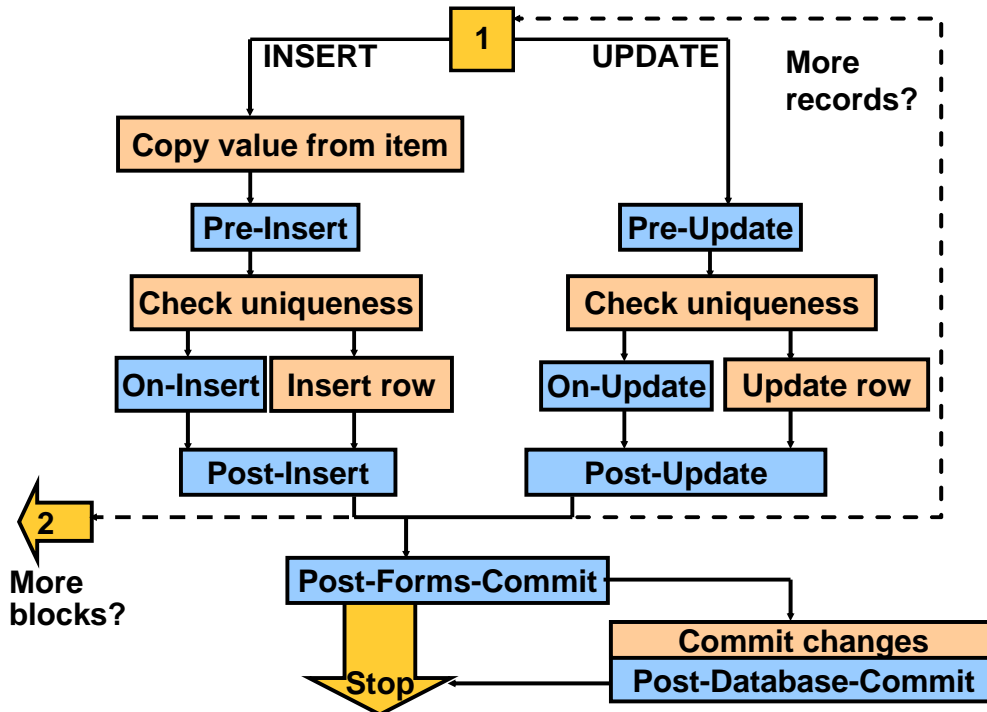


The Commit Sequence of Events

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).
5. Perform the DML:
 - 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.

The Commit Sequence of Events



The Commit Sequence of Events (continued)

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.

6. Fire the Post-Forms-Commit trigger.

If the current operation is COMMIT, then:

7. Issue an SQL-COMMIT statement.
8. 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

ORACLE

21-8

Copyright © 2004, Oracle. All rights reserved.

Characteristics of Commit Triggers

You have already seen when commit triggers fire during the normal flow of commit processing. The table on the next page gives more detailed information regarding the conditions under which these triggers fire.

It is usually unnecessary to code commit triggers, and the potential for coding errors is high. Because of this, use commit triggers only if your application requires special processing at commit time.

One valid use of commit triggers is to distribute DML statements to underlying tables when you are performing DML on a block based on a join view. However, using a database instead-of trigger may eliminate the need to define specialized Forms commit triggers for this purpose.

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 uncommitted posts issued before the savepoint are not rolled back.

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.

ORACLE

21-9

Copyright © 2004, Oracle. All rights reserved.

Characteristics of Commit Triggers (continued)

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 (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 marked for insert, update, or delete when Forms would typically issue its INSERT, UPDATE, or DELETE statement, replacing the default DML statements (Include a call to INSERT_RECORD, UPDATE_RECORD, or DELETE_RECORD built-in to perform 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 post or commit (this is also true for the SQL-COMMIT statement itself.)

Common Uses for Commit Triggers

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

ORACLE

21-10

Copyright © 2004, Oracle. All rights reserved.

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 in the table on the next page.

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 Forms.

Use DML statements in commit triggers only; otherwise the DML statements are not included in the administration kept by Forms concerning commit processing. This may lead to unexpected and unwanted results.

Common Uses for Commit Triggers

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

ORACLE

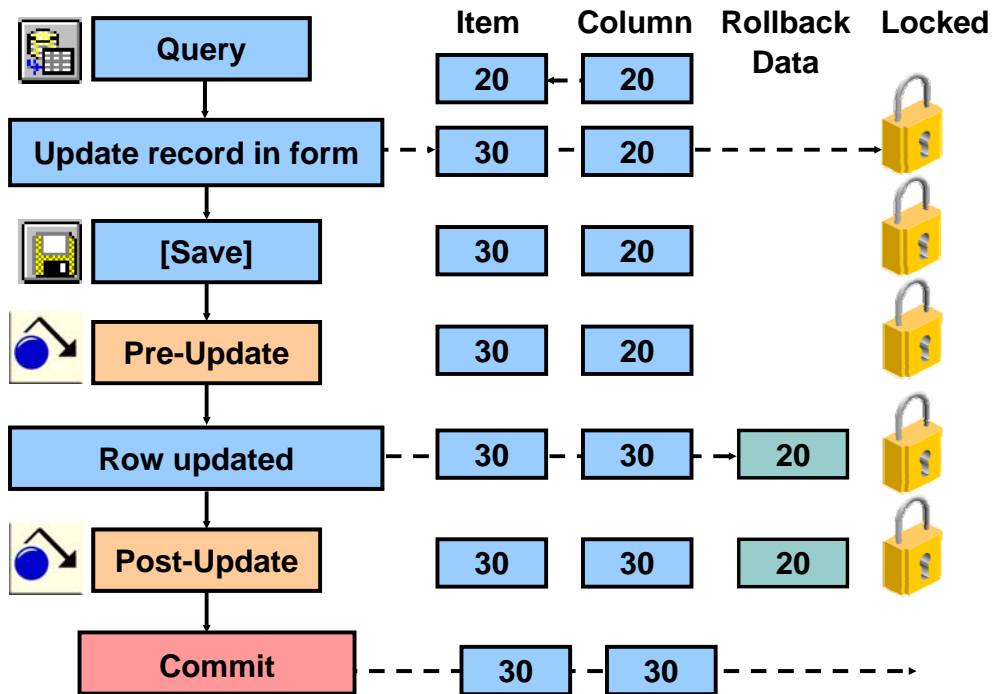
21-11

Copyright © 2004, Oracle. All rights reserved.

Common Uses for Commit Triggers (continued)

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 pseudo delete or to update a join view
Post-Forms-Commit	Checks complex multi-row constraints
Post-Database-Commit	Determines if commit was successful; determines if there are posted uncommitted changes

Life of an Update



Life of an Update

To help you decide where certain trigger actions can be performed, consider an update operation as an example.

Life of an Update (continued)

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. By default, the row is locked on the base table.
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. Forms 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. Forms 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 ORDERS
        WHERE customer_id = :CUSTOMERS.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;
```

ORACLE

21-14

Copyright © 2004, Oracle. All rights reserved.

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, want 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, Forms Builder automatically creates the related triggers to implement constraints.

Delete Validation (continued)

Example

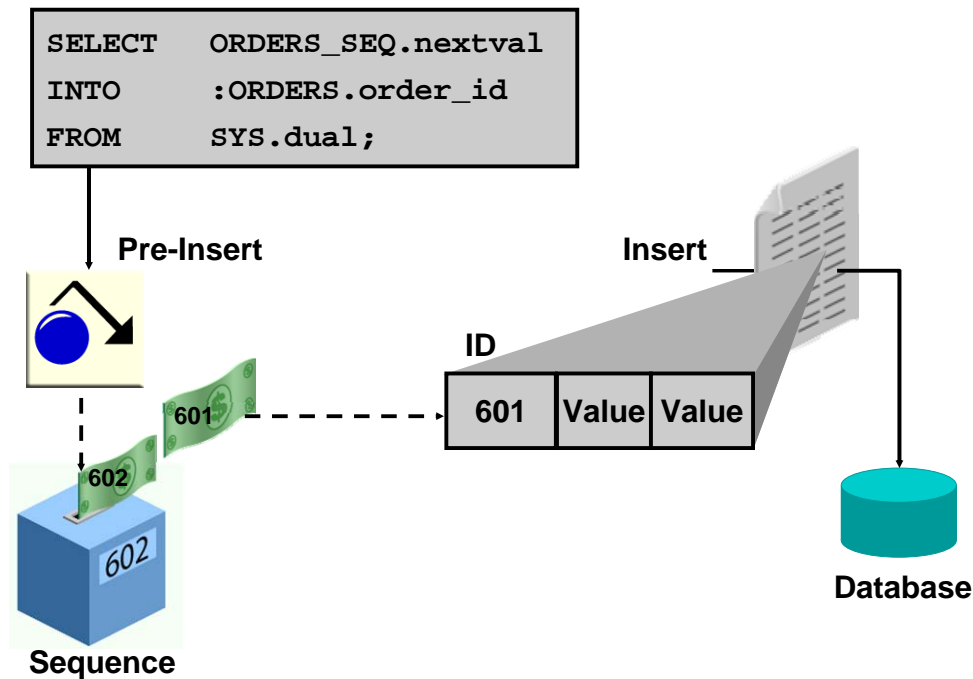
This Pre-Delete trigger on the CUSTOMER block of the CUSTOMERS form prevents deletion of rows if there are existing orders for the customer.

```
DECLARE
    CURSOR C1 IS
        SELECT 'anything' FROM ORDERS
        WHERE customer_id = :CUSTOMERS.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;
```

Instructor Note

In the preceding example, a local PL/SQL variable could be used rather than a global variable. When the block terminates, a PL/SQL variable ceases to exist. A global variable exists for the session.

Assigning Sequence Numbers



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

Assigning Sequence Numbers to Records (continued)

Example

This Pre-Insert trigger on the ORDERS block assigns an Order ID from the sequence ORDERS_SEQ, which will be written to the ORDER_ID column when the row is subsequently inserted.

```
SELECT ORDERS_SEQ.nextval
INTO :ORDERS.order_id
FROM SYS.dual;
```

Note: The Insert Allowed and Keyboard Navigable properties on :ORDERS.order_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.

Instructor Note

Demonstration

In the demonstration form that you are building:

- Create the Pre-Insert trigger, shown above, in the ORDERS block.
- Generate and run to show effects, noting that the assigned sequence number appears when the new record is saved.

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);
```

ORACLE

21-18

Copyright © 2004, Oracle. 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 ( :ORDERS.order_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.

```
:GLOBAL.insert_tot :=  
  TO_CHAR(TO_NUMBER(:GLOBAL.insert_tot)+1);
```

Testing the Results of Trigger DML

- SQL%FOUND
- SQL%NOTFOUND
- SQL%ROWCOUNT

```
UPDATE ORDERS
  SET order_date = SYSDATE
  WHERE order_id = :ORDERS.order_id;
IF SQL%NOTFOUND THEN
  MESSAGE('Record not found in database');
  RAISE form_trigger_failure;
END IF;
```

ORACLE

21-19

Copyright © 2004, Oracle. All rights reserved.

Testing the Results of Trigger DML

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).

Obtaining Cursor Information in PL/SQL

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

Testing the Results 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;
```

ORACLE

21-20

Copyright © 2004, Oracle. All rights reserved.

Testing the Results of Trigger DML (continued)

Obtaining Cursor Information in PL/SQL (continued)

Example

This When-Button-Pressed trigger records the date of posting as the date ordered for the current Order record. If a row is not found by the UPDATE statement, an error is reported.

```
UPDATE ORDERS
  SET order_date = SYSDATE
  WHERE order_id = :ORDERS.order_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
```

ORACLE

21-21

Copyright © 2004, Oracle. All rights reserved.

DML Statements Issued During Commit Processing

If you have not altered default commit processing, Forms 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
```

DML Statements Issued During Commit Processing

Rules:

- **DML statements may fire database triggers.**
- **Forms uses and retrieves ROWID.**
- **The Update Changed Columns Only and Enforce Column Security properties affect UPDATE statements.**
- **Locking statements are not issued.**

ORACLE

21-22

Copyright © 2004, Oracle. All rights reserved.

DML Statements Issued During Commit Processing (continued)

Rules

- These DML statements may fire associated database triggers.
- Forms uses the ROWID construct only when the Key mode block property is set to Unique (or Automatic, the default). Otherwise, the primary key is used to construct the WHERE clause.
- If Forms 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 Forms 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, Forms waits to lock the corresponding row until commit time.

Overriding Default Transaction Processing

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

Note: These triggers are meant to be used when connecting to data sources other than Oracle.

ORACLE

21-23

Copyright © 2004, Oracle. All rights reserved.

Overriding Default Transaction Processing

You have already seen that some commit triggers can be used to replace the default DML statements that Forms issues during commit processing. You can use several other triggers to override the default transaction processing of Forms.

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.

Overriding Default Transaction Processing

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	-

ORACLE

21-24

Copyright © 2004, Oracle. All rights reserved.

Overriding Default Transaction Processing (continued)

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 of 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.

Instructor Note

There are more transactional triggers, but those covered here are the most important.

Running Against Data Sources Other than Oracle

- **Two ways to run against data sources other than Oracle:**
 - Oracle Transparent Gateways
 - Write appropriate transactional triggers

ORACLE

21-25

Copyright © 2004, Oracle. All rights reserved.

Running Against Data Sources Other than Oracle

Two ways to run Against Data Sources Other than Oracle

- Use Oracle Transparent Gateway products.
- 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 Forms should interact with your data source. The specific settings depend on the capabilities of the data source.

Using Transactional Triggers

If no Open Gateway 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.

Running Against 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**

ORACLE

21-26

Copyright © 2004, Oracle. All rights reserved.

Running Against Data Sources Other than Oracle (continued)

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.

Instructor Note

Oracle no longer ships the Open Client Adaptor (OCA) for accessing databases through ODBC rather than SQL*Net. The stated replacement for OCA is to use the Oracle Transparent Gateways as a way to access data in databases such as Microsoft SQL Server or IBM DB2. In the initial releases of Forms 9i and 10g, however, you cannot access via the transparent gateways due to the lack of support in the gateways for “Select For Update” to enable row level locking. This deficiency will be addressed in a future release of Oracle Forms.

Getting and Setting the Commit Status

- **Commit status: Determines how record will be processed**
- **SYSTEM.RECORD_STATUS:**
 - **NEW**
 - **INSERT (also caused by control items)**
 - **QUERY**
 - **CHANGED**
- **SYSTEM.BLOCK_STATUS:**
 - **NEW (may contain records with status INSERT)**
 - **QUERY (also possible for control block)**
 - **CHANGED (block will be committed)**
- **SYSTEM.FORM_STATUS: NEW, QUERY, CHANGED**

ORACLE

21-27

Copyright © 2004, Oracle. 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.

Getting and Setting the Commit Status (continued)

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.)

The 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.)

The 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`

ORACLE

21-29

Copyright © 2004, Oracle. All rights reserved.

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
<code>GET_BLOCK_PROPERTY</code>	Use the <code>STATUS</code> property to obtain the block status of the specified block.
<code>GET_RECORD_PROPERTY</code>	Use the <code>STATUS</code> property to obtain the record status of the specified record in the specified block.
<code>SET_RECORD_PROPERTY</code>	Set the <code>STATUS</code> property of the specified record in the specified block to one of the following constants: <ul style="list-style-type: none">• <code>NEW_STATUS</code>• <code>INSERT_STATUS</code>• <code>QUERY_STATUS</code>• <code>CHANGED_STATUS</code>

Getting and Setting the Commit Status

- **Example: If the third record of block ORDERS 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.

ORACLE

21-30

Copyright © 2004, Oracle. All rights reserved.

Using Built-ins to Get the Commit Status (continued)

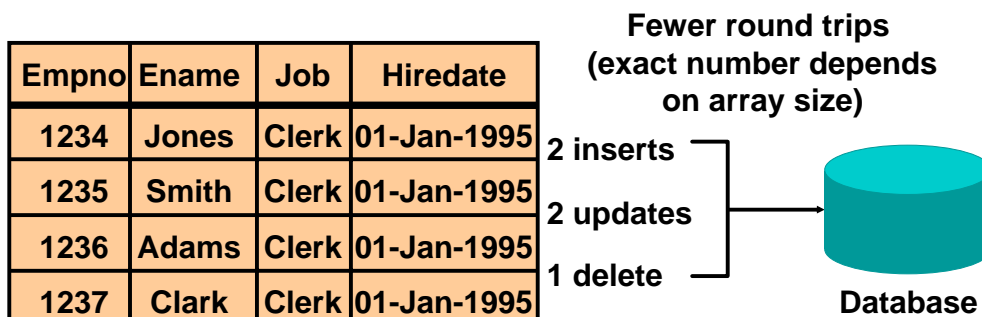
Example

If the third record of the ORDERS block is a changed database record, set the status back to QUERY.

```
BEGIN
  IF GET_RECORD_PROPERTY(3, 'ORDERS', status) = 'CHANGED'
  THEN
    SET_RECORD_PROPERTY(3, 'ORDERS', status,
                       query_status);
  END IF;
END;
```


Array DML

- Performs array inserts, updates, and deletes
- Vastly reduces network traffic



ORACLE

21-31

Copyright © 2004, Oracle. All rights reserved.

Array Processing

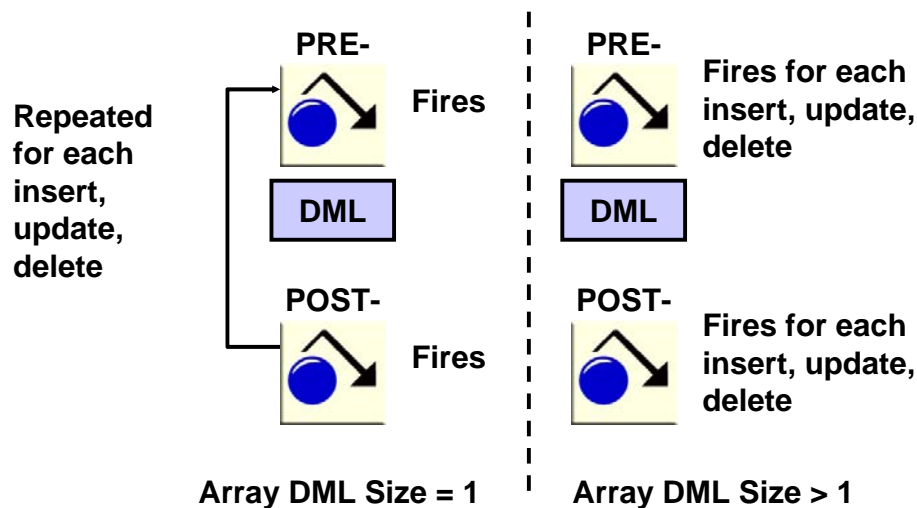
Overview

Array processing is an option in Forms Builder that alters the way records are processed. The default behavior of Forms 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. This is especially important in Web applications. With array processing, a structure (an array) containing multiple records is sent to or returned from the server for processing.

Forms 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



ORACLE

21-32

Copyright © 2004, Oracle. All rights reserved.

Array Processing (continued)

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.

Instructor Note

Some students may ask why DML Array Size is not always set to a very large number. There is a point of diminishing return. If each record is very large, a large array size might result in records being broken into multiple packets.

If you write On-DML triggers, you overwrite default transaction processing and cancel array processing.

Implementing Array DML

1. Enable the Array Processing option.
2. Specify a DML Array Size of greater than 1.
3. Specify block primary keys.

ORACLE

21-33

Copyright © 2004, Oracle. All rights reserved.

How to Implement Array DML

1. To set preferences:
 - Select Edit > 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

In this lesson, you should have learned that:

- **To apply changes to the database, Forms issues post and commit.**
- **The commit sequence of events:**
 1. **Validate the form.**
 2. **Process savepoint.**
 3. **Fire Pre-Commit.**
 4. **Validate the block (performed for all blocks in sequential order).**

ORACLE

21-34

Copyright © 2004, Oracle. 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

Summary

5. Perform the DML:

Delete records: Fire Pre-Delete, delete row or fire On-Delete, fire Post-Delete trigger

Insert records: Copy Value From Item, fire Pre-Insert, check record uniqueness, insert row or fire On-Insert, fire Post-Insert

Update records: Fire Pre-Update, check record uniqueness, update row or fire On-Update, fire Post-Update

6. Fire Post-Forms-Commit trigger.

If the current operation is COMMIT, then:

7. Issue an SQL-COMMIT statement.

8. Fire the Post-Database-Commit trigger.

ORACLE

Summary

- **You can supplement transaction processing with triggers:**
 - **Pre-Commit:** Fires once if form changes are made or uncommitted changes are posted
 - **[Pre | Post] – [Update | Insert | Delete]**
 - **On- [Update | Insert | Delete]:**
Fires per record, replacing default DML on row
Perform default functions with built-ins:
`[UPDATE | INSERT | DELETE] _RECORD`

ORACLE

21-36

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- **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.

Summary

- **Use the Pre-Insert trigger to allocate sequence numbers to records as they are applied to tables.**
- **Check or change commit status:**
 - `GET_BLOCK_PROPERTY, [GET | SET]_RECORD_STATUS`
 - `:SYSTEM.[FORM | BLOCK | RECORD]_STATUS`
- **Use transactional triggers to override or augment default commit processing.**
- **Reduce network roundtrips by setting DML Array Size block property to implement Array DML.**

ORACLE

21-37

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- **Common uses for commit triggers:** Check authorization, set up special locking requirements, generate sequence numbers, check complex constraints, replace default DML statements issued by Forms.
- **Overriding default transaction processing:**
 - Transactional On-*<Event>* triggers and “Do-the-Right-Thing” built-ins
 - Data sources other than Oracle use Transparent Gateway or transactional triggers
- **Getting and setting the commit status:**
 - System variables
 - Built-ins
- **Array DML**

Practice 21 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**

ORACLE

21-38

Copyright © 2004, Oracle. All rights reserved.

Practice 21 Overview

In this practice, you add transactional triggers to the ORDGXX form to automatically provide sequence numbers to records at save time. You also customize commit messages and the login screen in the CUSTGXX 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

Note: For solutions to this practice, see Practice 21 in Appendix A, “Practice Solutions.”

Practice 21

1. In the ORDGXX form, write a transactional trigger on the ORDERS block that populates `ORDERS.Order_Id` with the next value from the `ORDERS_SEQ` sequence. You can import the code from `pr21_1.txt`.
2. In the ORDERS block, set the Enabled property for the Order_ID item to No. Set the Required property for the Order_ID item to No. To ensure that the data remains visible, set the Background Property to gray.
3. Save, compile, and run the form to test.
4. Create a similar trigger on the ORDER_ITEMS block that assigns the Line_Item_Id when a new record is saved. Set the properties for the item as you did on `ORDERS.ORDER_ID`. You can import the code from `pr21_4.txt`.
5. Save and compile the form. Click Run Form to run the form and test the changes.
6. Open the CUSTGXX 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.
7. Create a procedure called `HANDLE_MESSAGE`. Import the `pr21_7a.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. Call the procedure when an error occurs. Pass the error code and an error message to be displayed. You can import the code from `pr21_7b.txt`. Call the procedure when a message occurs. Pass the message code and a message to be displayed. You can import the code from `pr21_7c.txt`.
8. 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 `pr21_8.txt` file.
9. Click Run Form to run the form and test the changes.

Instructor Note

Solution 21-4 is not the safest way. The better solution is to keep the total number of rows in another table that you can lock, but this solution is too advanced at this stage.

22

Writing Flexible Code

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	40 minutes	Lecture
	30 minutes	Practice
	70 minutes	Total

Objectives

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

- **Describe flexible code**
- **State the advantages of using system variables**
- **Identify built-in subprograms that assist flexible coding**
- **Write code to reference objects:**
 - **By internal ID**
 - **Indirectly**

ORACLE

22-2

Copyright © 2004, Oracle. All rights reserved.

Introduction

Overview

Forms Builder has a variety of features that enable you to write code in a flexible, reusable way.

What Is Flexible Code?

Flexible code:

- **Is reusable**
- **Is generic**
- **Avoids hard-coded object names**
- **Makes maintenance easier**
- **Increases productivity**

ORACLE

22-3

Copyright © 2004, Oracle. All rights reserved.

What Is Flexible Code?

Flexible code is code that you can use again. Flexible code is often generic code that you can use in any form module in an application. It typically includes the use of system variables instead of hard-coded object names.

Why Write Flexible Code?

Writing flexible code gives you the following advantages:

- It is easier for you and others to maintain.
- It increases productivity.

Using System Variables for Current Context

- **Input focus:**
 - SYSTEM.CURSOR_BLOCK
 - SYSTEM.CURSOR_RECORD
 - SYSTEM.CURSOR_ITEM
 - SYSTEM.CURSOR_VALUE

```
IF :SYSTEM.CURSOR_BLOCK = 'ORDERS' THEN
    GO_BLOCK('ORDER_ITEMS');
ELSIF :SYSTEM.CURSOR_BLOCK = 'ORDER_ITEMS' THEN
    GO_BLOCK('INVENTORIES');
ELSIF :SYSTEM.CURSOR_BLOCK = 'INVENTORIES' THEN
    GO_BLOCK('ORDERS');
END IF;
```

ORACLE

22-4

Copyright © 2004, Oracle. All rights reserved.

Using System Variables for Current Context

In this lesson, you use the system variables that provide the current status of the record, the block, and the form, as well as system variables that return the current input focus location.

System Variables for Locating Current Input Focus

System Variable	Function
CURSOR_BLOCK	The block that has the input focus
CURSOR_RECORD	The record that has the input focus
CURSOR_ITEM	The item and block that has the input focus
CURSOR_VALUE	The value of the item with the input focus

Example

The example above shows code that could be put in a When-Button-Pressed trigger to enable users to navigate to another block in the form. It tests the current block name, then navigates depending on the result.

Note: Be sure to set the button's Mouse Navigate property to No; otherwise the :SYSTEM.CURSOR_BLOCK will always be the block on which the button is located.

Using System Variables for Current Context

- **Trigger focus:**
 - `SYSTEM.TRIGGER_BLOCK`
 - `SYSTEM.TRIGGER_RECORD`
 - `SYSTEM.TRIGGER_ITEM`

ORACLE

22-5

Copyright © 2004, Oracle. All rights reserved.

Using System Variables for Current Context (continued)

System Variables for Locating Trigger Focus

System Variable	Function
TRIGGER_BLOCK	The block that the input focus was in when the trigger initially fired
TRIGGER_RECORD	The number of the record that Forms is processing
TRIGGER_ITEM	The block and item that the input focus was in when the trigger initially fired

Uses for Trigger Focus Variables

The variables for locating trigger focus are useful for navigating back to the initial block, record, and item after the trigger code completes. For example, the trigger code may navigate to other blocks, records, or items to perform actions upon them, but after the trigger fires, you may want the cursor to be in the same item instance that it was in originally. Because the navigation in the trigger occurs behind the scenes, the user will not even be aware of it.

Note: The best way to learn about system variables is to look at their values when a form is running. You can examine the system variables by using the Debugger.

System Status Variables

When-Button-Pressed

```
ENTER;  
IF :SYSTEM.BLOCK_STATUS = 'CHANGED' THEN  
    COMMIT_FORM;  
END IF;  
CLEAR_BLOCK;
```

ORACLE

22-6

Copyright © 2004, Oracle. All rights reserved.

System Variables for Determining the Current Status of the Form

You can use these system status variables presented in the previous lesson to write the code that performs one action for one particular status and a different action for another:

- SYSTEM.RECORD_STATUS
- SYSTEM.BLOCK_STATUS
- SYSTEM.FORM_STATUS

The example in the slide performs a commit before clearing a block if there are changes to commit within that block.

GET_<object>_PROPERTY Built-Ins

- GET_APPLICATION_PROPERTY
- GET_FORM_PROPERTY
- GET_BLOCK_PROPERTY
- GET_RELATION_PROPERTY
- GET_RECORD_PROPERTY
- GET_ITEM_PROPERTY
- GET_ITEM_INSTANCE_PROPERTY

Using Built-In Subprograms for Flexible Coding

Some of Forms Builder built-in subprograms provide the same type of run-time status information that built-in system variables provide.

GET_APPLICATION_PROPERTY

The GET_APPLICATION_PROPERTY built-in returns information about the current Forms application.

Example

The following example captures the username and the operating system information:

```
:GLOBAL.username := GET_APPLICATION_PROPERTY (USERNAME) ;  
:GLOBAL.o_sys :=  
GET_APPLICATION_PROPERTY (OPERATING_SYSTEM) ;
```

Note: The GET_APPLICATION_PROPERTY built-in returns information about the Forms application running on the middle tier. If you require information about the client machine, you can use a JavaBean.

GET_<object>_PROPERTY Built-Ins

- GET_LOV_PROPERTY
- GET_RADIO_BUTTON_PROPERTY
- GET_MENU_ITEM_PROPERTY
- GET_CANVAS_PROPERTY
- GET_TAB_PAGE_PROPERTY
- GET_VIEW_PROPERTY
- GET_WINDOW_PROPERTY

Using Built-In Subprograms for Flexible Coding (continued)

GET_BLOCK_PROPERTY

The GET_BLOCK_PROPERTY built-in returns information about a specified block.

Example

To determine the current record that is visible at the first (top) line of a block:

```
...GET_BLOCK_PROPERTY('blockname',top_record)...
```

GET_ITEM_PROPERTY

The GET_ITEM_PROPERTY built-in returns information about a specified item.

Example

To determine the canvas that the item with the input focus displays on, use:

```
DECLARE
  cv_name varchar2(30);
BEGIN
  cv_name :=
  GET_ITEM_PROPERTY(:SYSTEM.CURSOR_ITEM,item_canvas);
  ...
```

SET_<object>_PROPERTY Built-Ins

- SET_APPLICATION_PROPERTY
- SET_FORM_PROPERTY
- SET_BLOCK_PROPERTY
- SET_RELATION_PROPERTY
- SET_RECORD_PROPERTY
- SET_ITEM_PROPERTY
- SET_ITEM_INSTANCE_PROPERTY

ORACLE

22-9

Copyright © 2004, Oracle. All rights reserved.

SET_<object>_PROPERTY Built-Ins

SET_ITEM_INSTANCE_PROPERTY

The SET_ITEM_INSTANCE_PROPERTY built-in modifies the specified instance of an item in a block by changing the specified item property.

Example

The following example sets the visual attribute to VA_CURR for the current record of the current item:

```
SET_ITEM_INSTANCE_PROPERTY( :SYSTEM.CURSOR_ITEM,  
    VISUAL_ATTRIBUTE, CURRENT_RECORD, 'VA_CURR' );
```

SET_MENU_ITEM_PROPERTY

The SET_MENU_ITEM_PROPERTY built-in modifies the given properties of a menu item.

Example

To enable the save menu item in a file menu:

```
SET_MENU_ITEM_PROPERTY( 'FILE.SAVE', ENABLED, PROPERTY_TRUE );
```

SET_<object>_PROPERTY Built-Ins

- SET_LOV_PROPERTY
- SET_RADIO_BUTTON_PROPERTY
- SET_MENU_ITEM_PROPERTY
- SET_CANVAS_PROPERTY
- SET_TAB_PAGE_PROPERTY
- SET_VIEW_PROPERTY
- SET_WINDOW_PROPERTY

ORACLE

22-10

Copyright © 2004, Oracle. All rights reserved.

SET_<object>_PROPERTY Built-Ins (continued)

SET_TAB_PAGE_PROPERTY

The SET_TAB_PAGE_PROPERTY built-in sets the tab page properties of the specified tab canvas page.

Example

To enable tab_page_1, if it is already disabled, use:

```
DECLARE
  tbpg_id  TAB_PAGE;
BEGIN
  tbpg_id := FIND_TAB_PAGE('tab_page_1');
  IF GET_TAB_PAGE_PROPERTY(tbpg_id, enabled) = 'FALSE'
  THEN
    SET_TAB_PAGE_PROPERTY(tbpg_id, enabled,property_true);
  END IF;
END;
```

Referencing Objects by Internal ID

Finding the object ID:

```
lov_id := FIND_LOV('my_lov')
```



Referencing an object by ID:

```
...SHOW_LOV(lov_id)
```



Referencing an object by name:

```
...SHOW_LOV('my_lov')
```



ORACLE

22-11

Copyright © 2004, Oracle. All rights reserved.

Referencing Objects by Internal ID

Forms Builder assigns an ID to each object that you create. An object ID is an internal value that is never displayed. You can get the ID of an object by calling the built-in FIND_ subprogram appropriate for the object. The FIND_ subprograms require a fully qualified object name as a parameter. For instance, when referring to an item, use *BLOCKNAME.ITEMNAME*.

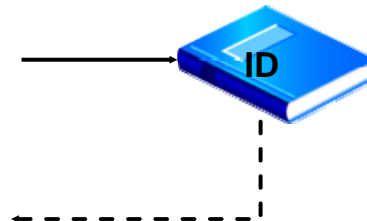
The return values of the FIND_ subprograms (the object IDs) are of a specific type. The types for object IDs are predefined in Forms Builder. There is a different type for each object.

Three Reasons for Using Object IDs

- Improving performance (Forms looks up the object only once when you initially call the FIND_ subprogram to get the ID. When you refer to an object by name in a trigger, Forms must look up the object ID each time.)
- Writing more generic code
- Testing whether an object exists (using the ID_NULL function and FIND_object)

FIND_ Built-Ins

- FIND_ALERT
- FIND_BLOCK
- FIND_CANVAS
- FIND_EDITOR
- FIND_FORM
- FIND_ITEM
- FIND_LOV
- FIND_RELATION
- FIND_VIEW
- FIND_WINDOW



ORACLE

22-12

Copyright © 2004, Oracle. All rights reserved.

Forms Builder FIND_ Built-Ins

The table below lists some of the FIND_ subprograms, along with the object classes that use them and the return types they produce:

Object Class	Subprogram	Return Type
Alert	FIND_ALERT	ALERT
Block	FIND_BLOCK	BLOCK
Canvas	FIND_CANVAS	CANVAS
Editor	FIND_EDITOR	EDITOR
Form	FIND_FORM	FORMMODULE
Item	FIND_ITEM	ITEM
LOV	FIND_LOV	LOV
Relation	FIND_RELATION	RELATION
View	FIND_VIEW	VIEWPORT
Window	FIND_WINDOW	WINDOW

Using Object IDs

- **Declare a PL/SQL variable of the same data type.**
- **Use the variable for any later reference to the object.**
- **Use the variable within the current PL/SQL block only.**

ORACLE

22-13

Copyright © 2004, Oracle. All rights reserved.

Declaring Variables for Object IDs

To use an object ID, you must first assign it to a variable. You must declare a variable of the same type as the object ID.

The following example uses the `FIND_ITEM` built-in to assign the ID of the item that currently has input focus to the variable `id_var`.

Once you assign an object ID to a variable in a trigger or PL/SQL program unit, you can use that variable to reference the object, rather than referring to the object by name.

```
DECLARE
    id_var item;
BEGIN
    id_var := FIND_ITEM(:SYSTEM.CURSOR_ITEM);
    . . .
END;
```

Using Object IDs

Example:

```
DECLARE
    item_var item;
BEGIN
    item_var := FIND_ITEM(:SYSTEM.CURSOR_ITEM);
    SET_ITEM_PROPERTY(item_var,position,30,55);
    SET_ITEM_PROPERTY(item_var,prompt_text,'Current');
END;
```

Declaring Variables for Object IDs (continued)

The following two examples show that you can pass either an item name or an item ID to the SET_ITEM_PROPERTY built-in subprogram. The following calls are logically equivalent:

```
SET_ITEM_PROPERTY('ORDERS.order_id',position,50,35);
SET_ITEM_PROPERTY(id_var,position,50,35);
```

You can use either object IDs or object names in the same argument list, provided that each individual argument refers to a distinct object.

You cannot, however, use an object ID and an object name to form a fully qualified object_name (blockname.itemname). The following call is illegal:

```
GO_ITEM(block_id.'item_name');
```

Note: Use the FIND_ built-in subprograms only when referring to an object more than once in the same trigger or PL/SQL program unit.

Increasing the Scope of Object IDs

- **A PL/SQL variable has limited scope.**
- **An `.id` extension:**
 - **Broadens the scope**
 - **Converts to a numeric format**
 - **Enables assignment to a global variable**
 - **Converts back to the object data type**

ORACLE

22-15

Copyright © 2004, Oracle. All rights reserved.

Using Object IDs Outside the Initial PL/SQL Block

You have seen how object IDs are referenced within the trigger or program unit by means of PL/SQL variables. You can reference these PL/SQL variables only in the current PL/SQL block; however, you can increase the scope of an object ID.

To reference an object ID outside the initial PL/SQL block, you need to convert the ID to a numeric format using an `.id` extension for your declared PL/SQL variable, then assign it to a global variable.

Example

The following example of trigger code assigns the object ID to a local PL/SQL variable (`item_var`) initially, then to a global variable (`global.item`):

```
DECLARE
    item_var item;
BEGIN
    item_var := FIND_ITEM(:SYSTEM.CURSOR_ITEM);
    :GLOBAL.item := item_var.id;
END;
```

Using Object IDs Outside the Initial PL/SQL Block (continued)

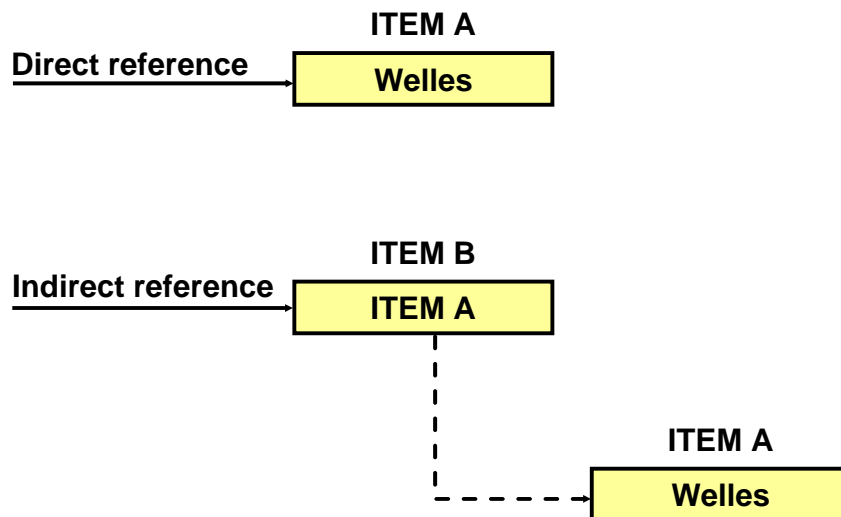
You can pass the global variable around within the application. To be able to reuse the object ID, you need to convert it back to its original data type.

Example

The following example shows the conversion of the global variable back to its original PL/SQL variable data type:

```
DECLARE
    item_var item;
BEGIN
    item_var.id := TO_NUMBER(:GLOBAL.item);
    GO_ITEM(item_var);
END;
```

Referencing Objects Indirectly



ORACLE

22-17

Copyright © 2004, Oracle. All rights reserved.

Referencing Items Indirectly

By referencing items indirectly, you can write more generic, reusable code. Using variables instead of actual item names, you can write a PL/SQL program unit to use any item whose name is assigned to the indicated variable.

You can reference items indirectly with the `NAME_IN` and `COPY` built-in subprograms.

Note: Use indirect referencing when you create procedures and functions in a library module, because direct references cannot be resolved.

Instructor Note

Check student understanding of this concept by asking them what the last sentence above means.

Referencing Objects Indirectly

The `NAME_IN` function:

- **Returns:**
 - The contents of variable
 - Character string
- **Use conversion functions for NUMBER and DATE**

Referencing Items Indirectly (continued)

Using the `NAME_IN` Built-in Function

The `NAME_IN` function returns the contents of an indicated variable. The following statements are equivalent. The first one uses a direct reference to `customer.name`, whereas the second uses an indirect reference:

```
IF :CUSTOMERS.cust_last_name = 'Welles'...
```

In a library, you could avoid this direct reference by using:

```
IF NAME_IN('CUSTOMERS.cust_last_name') = 'Welles'...
```

The return value of `NAME_IN` is always a character string. To use `NAME_IN` for a date or number item, convert the string to the desired data type with the appropriate conversion function. For instance:

```
date_var := TO_DATE(NAME_IN('ORDERS.order_date'));
```

Referencing Objects Indirectly

The COPY procedure allows:

- **Direct copy:**

```
COPY('Welles','CUSTOMERS.cust_last_name');
```

- **Indirect copy:**

```
COPY('Welles',NAME_IN('global.customer_name_item'));
```

ORACLE

22-19

Copyright © 2004, Oracle. All rights reserved.

Referencing Items Indirectly (continued)

Using the COPY Built-in Procedure

The COPY built-in assigns an indicated value to an indicated variable or item. Unlike the standard PL/SQL assignment statement, using the COPY built-in enables you to indirectly reference the item whose value is being set. The first example in the slide shows copying using a direct reference to the form item.

Using COPY with NAME_IN

Use the COPY built-in subprogram with the NAME_IN built-in to indirectly assign a value to an item whose name is stored in a global variable, as in the second example in the slide.

Instructor Note

Check student understanding of this concept by asking them what is contained in 'GLOBAL.customer_name_item' referenced in the last example above. Explain to students that the name of the form item, 'customers.cust_last_name' was previously assigned to the global variable, perhaps in a When-New-Form-Instance trigger.

Summary

In this lesson, you should have learned that:

- **Flexible code is reusable, generic code that you can use in any form module in an application.**
- **With system variables you can:**
 - **Perform actions conditionally based on current location (`SYSTEM.CURSOR_[RECORD | ITEM | BLOCK]`)**
 - **Use the value of an item without knowing its name (`SYSTEM.CURSOR_VALUE`)**
 - **Navigate to the initial location after a trigger completes: (`SYSTEM.TRIGGER_[RECORD | ITEM | BLOCK]`)**
 - **Perform actions conditionally based on commit status: `SYSTEM.[RECORD | BLOCK | FORM]_STATUS`**

ORACLE

22-20

Copyright © 2004, Oracle. All rights reserved.

Summary

Use the following to write flexible code:

- System variables:
 - To avoid hard-coding object names
 - To return information about the current state of the form

Summary

- **The [GET | SET]_<object>_PROPERTY built-ins are useful in flexible coding.**
- **Code that references objects is more efficient and generic:**
 - **By internal ID: Use FIND_<object> built-ins**
 - **Indirectly: Use COPY and NAME_IN built-ins**

ORACLE

22-21

Copyright © 2004, Oracle. All rights reserved.

Summary (continued)

- GET_<object>_PROPERTY built-ins, to return current property values for Forms Builder objects
- Object IDs, to improve performance
- Indirect referencing, to allow form module variables to be referenced in library and menu modules

Practice 22 Overview

This practice covers the following topics:

- Populating product images only when the image item is displayed.
- Modifying the `When-Button-Pressed` trigger of the `Image_Button` in order to use object IDs instead of object names.
- Write generic code to print out the names of the blocks in a form.

ORACLE

22-22

Copyright © 2004, Oracle. All rights reserved.

Practice 22 Overview

In this practice, you use properties and variables in the `ORDGXX` form to provide flexible use of its code. You also make use of object IDs.

- Populating product images only when the image item is displayed
- Modifying the `When-Button-Pressed` trigger of the `Image_Button` in order to use object IDs instead of object names
- Writing generic code to print out the names of blocks in a form and using the same code in two different forms

Note: For solutions to this practice, see Practice 22 in Appendix A, “Practice Solutions.”

Practice 22

1. In the ORDGXX form, alter the code called by the triggers that populate the Product_Image item when the image item is displayed.
Add a test in the code to check Product_Image. Perform the trigger actions only if the image is currently displayed. Use the GET_ITEM_PROPERTY built-in function. The code is contained in pr22_1.txt.
2. Alter the When-Button-Pressed trigger on the Image_Button so that object IDs are used.
Use a FIND_object function to obtain the IDs of each item referenced by the trigger. Declare variables for these IDs, and use them in each item reference in the trigger. The code is contained in pr22_2.txt.
3. Create a button called Blocks_Button in the CONTROL block and place it on the Toolbar canvas. Label the button Show Blocks. Set its navigation and color properties the same as the other toolbar buttons.
The code for the button should print a message showing what block the user is currently in. It should keep track of the block and item where the cursor was located when the trigger was invoked (:SYSTEM.CURSOR_BLOCK and :SYSTEM.CURSOR_ITEM). It should then loop through the remaining navigable blocks of the form and print a message giving the names (SYSTEM.current_block) of all the navigable blocks in the form. Finally, it should navigate back to the block and item where the cursor was located when the trigger began to fire. Be sure to set the Mouse Navigate property of the button to No. You may import the code for the trigger from pr22_3.txt.
4. Save, compile, and run the form to test these features.
5. The trigger code above is generic, so it will work with any form. Open the CUSTGXX form and define a similar Blocks_Button, labeled Show Blocks, in the CONTROL block, and place it just under the Color button on the CV_CUSTOMER canvas. Drag the When-Button-Pressed trigger you created for the Blocks_Button of the ORDGXX form to the Blocks_Button of the CUSTGXX form. Run the CUSTGXX form to test the button.

23

Sharing Objects and Code

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	50 minutes	Lecture
	30 minutes	Practice
	80 minutes	Total

Objectives

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

- **Describe the various methods for reusing objects and code**
- **Inherit properties from property classes**
- **Group related objects for reuse**
- **Explain the inheritance symbols in the Property Palette**
- **Reuse objects from an object library**
- **Reuse PL/SQL code**

ORACLE

23-2

Copyright © 2004, Oracle. All rights reserved.

Introduction

Overview

Forms Builder includes some features specifically for object and code reuse. In this lesson, you learn how to share objects between form modules using the Object Library. You also learn how to share code using the PL/SQL Library.

Benefits of Reusing Objects and Code

- **Increases productivity**
- **Decreases maintenance**
- **Increases modularity**
- **Maintains standards**
- **Improves application performance**

Benefits of Reusable Objects and Code

When you are developing applications, you should share and reuse objects and code wherever possible in order to:

- **Increase productivity:** You can develop applications much more effectively and efficiently if you are not trying to “start over” each time you write a piece of code. By sharing and reusing frequently used objects and code, you can cut down development time and increase productivity.
- **Decrease maintenance:** By creating applications that use or call the same object or piece of code several times, you can decrease maintenance time.
- **Increase modularity:** Sharing and reusing code increases the modularity of your applications.
- **Maintain standards:** You can maintain standards by reusing objects and code. If you create an object once and copy it again and again, you do not run the risk of introducing minor changes. In fact, you can create a set of standard objects and some pieces of standard code and use them as a starting point for all of your new form modules.

Benefits of Reusable Objects and Code (continued)

- **Improved application performance:** When Forms Services communicates the user interface to the Forms Client, it sends meta-data about the items on the form. This meta-data includes values of properties that differ from the default. Once an item is defined, the meta-data about the next item includes only those properties that differ from the previous item. This is referred to as message diffing. Promoting similarities among items by using the methods of object reuse presented in this lesson improves the efficiency of message diffing and thus decreases network traffic and increases performance.

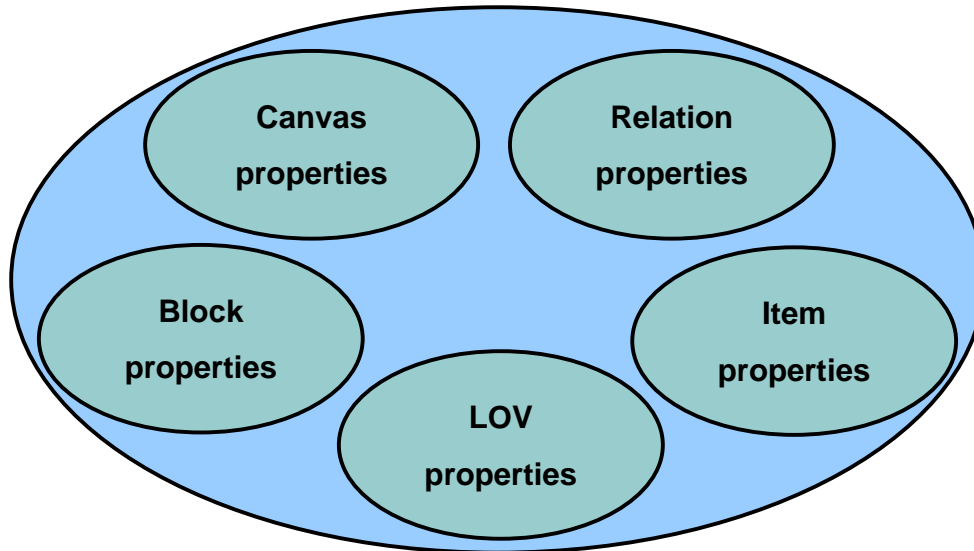
Promoting Similarities Among Objects

One of the easiest ways a developer can increase the efficiency of network performance through message diffing is by using consistent standards for all objects within an application. Items of different types should at least have common values for common or shared properties.

To maximize reuse, the developer should apply the following guidelines in the order shown:

- **Accept default properties as much as possible:** If the properties are not overwritten for each object, then the value for common properties will be the same regardless of the the object type, except for position and size.
- **Use SmartClasses to describe an object:** If, because of design standards, the use of default properties is not a viable option, then the subclassing of objects from a set of SmartClasses ensures that the development standards are being met. It also forces a high degree of property sharing across widgets. Items of the same type will then have (unless overridden) the same properties and hence will be able to share properties more effectively. You will learn about SmartClasses in this lesson.
- **Use sets of visual attributes:** If SmartClasses are not being used to enforce standards and common properties then use sets of partial visual attributes to enforce a common set of properties across objects of different types: for example, font, font size, foreground color, background color, and so on. These sets of visual attributes can be defined as Property Classes, as explained in the following slides.

What Are Property Classes?



ORACLE

23-5

Copyright © 2004, Oracle. All rights reserved.

Property Class

What is a Property Class?

A *property class* is a named object that contains a list of properties and their settings.

Why use Property Classes?

Use property classes to:

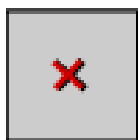
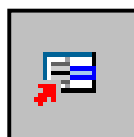
- Increase productivity by setting standard or frequently used values for common properties and associating them with several Forms Builder objects. You can use property classes to define standard properties not just for one particular object, but for several at a time. This results in increased productivity, because it eliminates the time spent on setting identical properties for several objects.
- Improve network performance by increasing the efficiency of message diffing.

Creating a Property Class

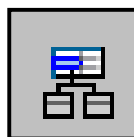
Add Property



Inherit Property



Delete Property



Property Class

Creating a Property Class

When you create a property class, you have all the properties from every Forms Builder object available. You choose the properties and their values to include in the property class. You can create a property class in two ways:

- Using the Create button in the Object Navigator
- Using the Create Property Class button

How to Create a Property Class from the Object Navigator

1. Click the Property Class node.
2. Click Create. A new property class entry displays.
3. Add the required properties and their values using the Add Property button in the Property Palette.

How to Create a Property Class from the Property Palette

1. In the Object Navigator, click the object whose properties you want to copy into a property class.
2. Move to the Property Palette, select the properties you want to copy into a property class, and click the Property Class icon. An information alert is displayed.
3. Use the Object Navigator to locate the property class and change its name.

Creating a Property Class (continued)

Adding a Property

Once you create a property class, you can add a property by clicking the Add Property button and selecting a property from the list. Set the value for that property using the Property Palette.

You can also use the Copy Properties button and the Paste Properties button to add several properties at a time to a property class.

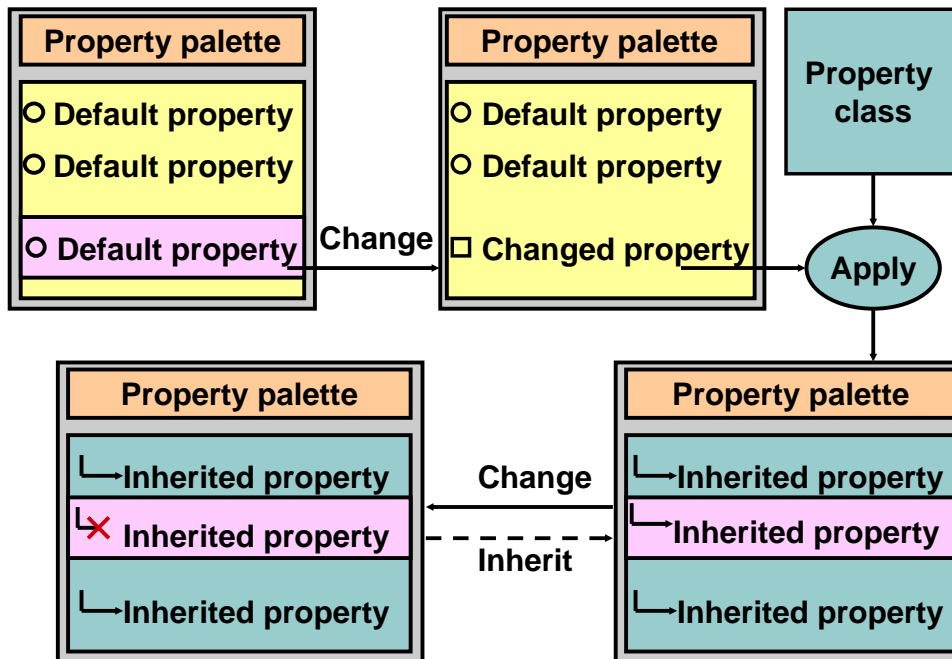
Deleting a Property

You can remove properties from a property class using the Delete Property button.

Instructor Note

- Create a property class in the Property Palette.
- Create a property class in the Object Navigator.
- Include properties for different objects.
- Associate a property class with an object.

Inheriting from a Property Class



Inheriting from a Property Class

Once you create a property class and add properties, you can use the property class. To apply the properties from a property class to an object, use the Subclass Information property in the Property Palette.

What is an Inherited Property?

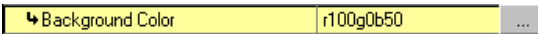

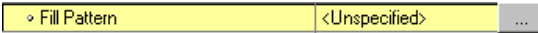
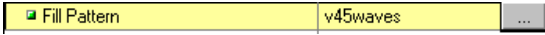
An *inherited property* is one that takes its value from the property class that you associated with the object. An inherited property is displayed with an arrow to the left of the property name.

What is a Variant Property?

A *variant property* is one that has a modified value even though it is inherited from the property class associated with the object. You can override the setting of any inherited property to make that property variant. Variant properties are displayed with a red cross over an arrow.

Inheriting from a Property Class

- **Set the Subclass Information property.**
- **Convert an inherited property to a variant property.**
- **Convert a variant property to an inherited property.**
- **Convert a changed property to a default property.**

Inherited Property	
Variant Property	
Default Property	
Changed Property	

ORACLE

23-9

Copyright © 2004, Oracle. All rights reserved.

Inheriting from a Property Class (continued)

How to Inherit Property Values from a Property Class

1. Click the object to which you want to apply the properties from the property class.
2. Click the Subclass Information property in the Property Palette.
3. Select the property class whose properties you want to use. The object takes on the values of that property class. Inherited properties are displayed with an arrow symbol.

Converting an Inherited Property to a Variant Property

To convert an inherited property to a variant property, simply enter a new value over the inherited one.

Converting a Variant Property to an Inherited Property

To convert a variant property to an inherited property, click the Inherit icon in the Property Palette.

Converting a Changed Property to a Default Property

You can also use the Inherit icon to revert a changed property back to its default.

What Are Object Groups?

Object groups:

- **Are logical containers**
- **Enable you to:**
 - **Group related objects**
 - **Copy multiple objects in one operation**

ORACLE

23-10

Copyright © 2004, Oracle. All rights reserved.

What Are Object Groups?

An *object group* is a logical container for a set of Forms Builder objects.

Why Use Object Groups?

You define an object group when you want to:

- Package related objects for copying or subclassing in another module
- Bundle numerous objects into higher-level building blocks that you can use again in another application.

Example

Your application can include an appointment scheduler that you want to make available to other applications. You can package the various objects in an object group and copy the entire bundle in one operation.

Creating and Using Object Groups

- **Blocks include:**
 - Items
 - Item-level triggers
 - Block-level triggers
 - Relations
- **Object groups cannot include other object groups**
- **Deleting an object group does not affect the objects**
- **Deleting an object affects the object group**

ORACLE

23-11

Copyright © 2004, Oracle. All rights reserved.

Creating and Using Object Groups

How to Create an Object Group

1. Click the Object Group node in the Object Navigator.
2. Click the Create icon. A new object group entry is displayed.
3. Rename the new object group.
4. Click the form module and expand all.
5. Control-click all the objects of one type that you want to include in the object group.
6. Drag the selected objects into the new object group entry. The objects are displayed as object group children.
7. Repeat steps 5 and 6 for different object types.

The objects in the object group are still displayed in their usual position in the Object Navigator, as well as within the object group. The objects in the object group are not duplicates, but pointers to the source objects.

Creating and Using Object Groups (continued)

Things to consider when using object groups

- Including a block in an object group also includes its items, the item-level triggers, the block-level triggers and the relations. You cannot use any of these objects in an object group without the block.
- It is not possible to include another object group.
- Deleting an object from a module automatically deletes the object from the object group.
- Deleting an object group from a module does not delete the objects it contains from the module.

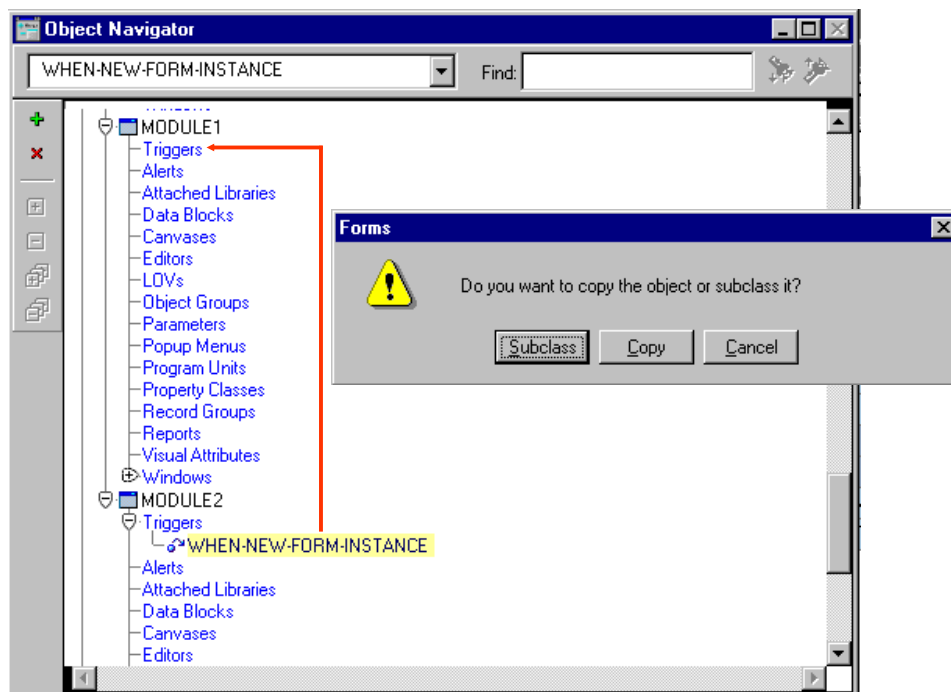
Subclass Information Dialog Box

The Subclass Information property of a form object shows a dialog box that provides information about the origins of the object. You can see whether an object is local to the form document or foreign to it. If the object is foreign to the current form, the Module field shows the module from which the object originates. The original object name is shown in the Object Name field.

Instructor Note

- Create an object group.
- Include a block, canvas-view, and form-level trigger.
- Point out that the objects in the object group are still visible in their respective positions in the Object Navigator.
- Drag the object group and drop it into a second form module.

Copying and Subclassing Objects and Code



23-13

Copyright © 2004, Oracle. All rights reserved.

ORACLE

Copying and Subclassing Objects and Code

You can copy or subclass objects:

- Between modules, by dragging objects between the modules in the Object Navigator
- Within a single module by selecting the object in the Object Navigator, pressing [Ctrl], and dragging it to create the new object

When you drag objects, a dialog box appears that asks whether you want to copy or subclass the object.

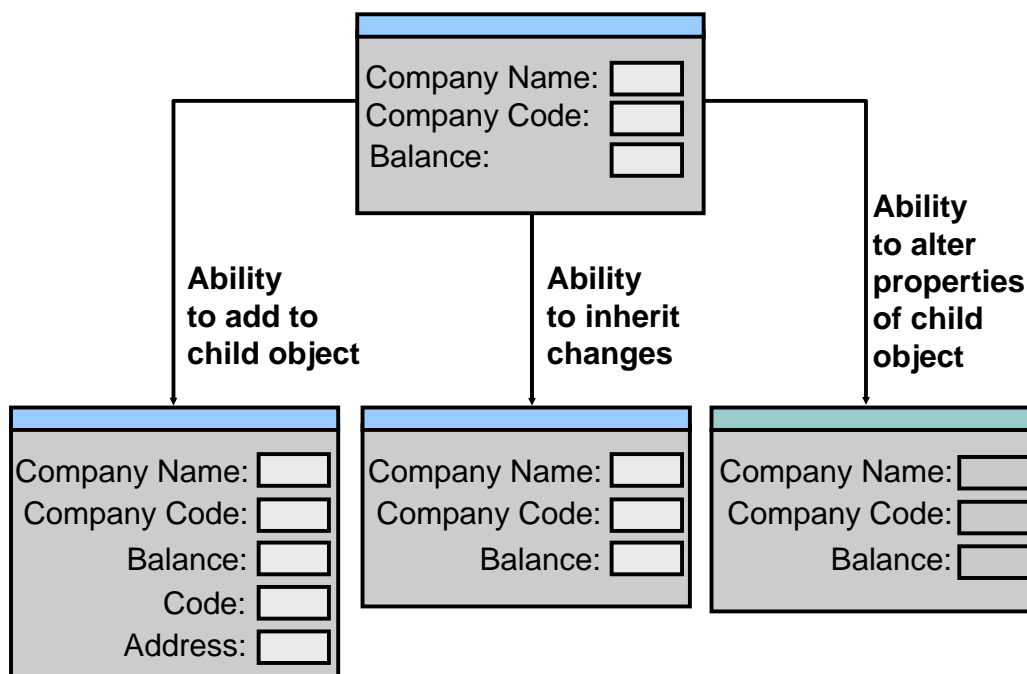
Copying an Object

Copying an object creates a separate, unique version of that object in the target module. Any objects owned by the copied object are also copied.

Points to Remember

- Use copying to export the definition of an object to another module.
- Changes made to a copied object in the source module do not affect the copied object in the target module.

Subclassing



ORACLE

23-14

Copyright © 2004, Oracle. All rights reserved.

Subclassing an Object

With subclassing you can make an exact copy, and then alter the properties of some objects if desired. If you change the parent class, the changes also apply to the properties of the subclassed object that you have not altered. However, any properties that you override remain overridden. This provides a powerful object inheritance model.

When you subclass a data block, you can:

- Change the structure of the parent, automatically propagating the changes to the child
- Add or change properties to the child to override the inheritance

When you subclass a data block you cannot:

- Delete items from the child
- Change the order of items in the child
- Add items to the child unless you add them to the end

Note: Subclassing is an object-oriented term that refers to the following capabilities:

- Inheriting the characteristics of a base class (Inheritance)
- Overriding properties of the base class (Specialization)

Subclassing an Object (continued)

Ability to add to an Object

You can create an exact copy of an object, and you can add to the subclassed object. For example, you can add additional items to the end of a subclassed block.

Ability to Alter Properties

With subclassing, you can make an exact copy and then alter the properties of some objects. If you change the parent class, the changes also apply to the properties of the subclassed object that you have not altered. However, any properties that you override remain overridden.

Ability to Inherit Changes

When you change the properties of a parent object, all child objects inherit those properties if they are not already overridden.





The child inherits changes:

- Immediately, if the parent and child objects are in the same form
- When you reload the form containing a child object

Ability to re-inherit

If you make changes to the child object to override properties of the parent object, you can click the Inherit icon to re-inherit the property from the parent object.

Property Palette icons: Enable you to identify inherited or overridden properties.

Property Palette Icon	Meaning
 Circle	The value for the property is the default.
 Square	The value for the property was changed from the default.
 Arrow	The value for the property was inherited.
 Arrow with red X	The value for the property was inherited but overridden (variant property).

Instructor Note

Demonstration: Open the following forms: `referenced.fmb`, `copied.fmb`, and `subclassed.fmb`. The `referenced.fmb` module contains a block with a single item. Open the Property Palette for that item and show that the X Position, Y Position, Width, and Height properties are all set to 20. As you make changes in properties in this demo, point out the different icons for inherited and overridden properties.

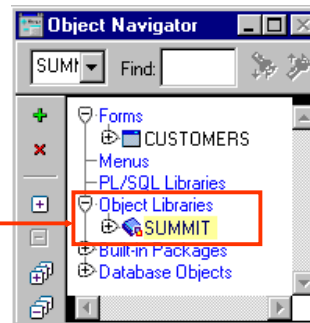
Drag and drop the block into `copied.fmb` and `subclassed.fmb`, copying it in the first case and subclassing it in the second. Change the item's Width in `referenced.fmb`. Demonstrate that the change is propagated to `subclassed.fmb`, but not to `copied.fmb`.

In `subclassed.fmb`, change the Height property of the item to 40. Now change the Height property of the item in `referenced.fmb` to 10. Return to `subclassed.fmb` and show that the overridden Height of 40 is still in effect. Select the Height property and click the Inherit icon. Point out to students that the new Height of 10 is inherited from `referenced.fmb`, not the original height of 20.

What Are Object Libraries?

An Object Library:

- Is a convenient container of objects for reuse
- Simplifies reuse in complex environments
- Supports corporate, project, and personal standards
- Simplifies the sharing of reusable components
- Is separate from the form module



What Are Object Libraries?

Object libraries are convenient containers of objects for reuse. They simplify reuse in complex environments, and they support corporate, project, and personal standards.

An object library can contain simple objects, property classes, object groups, and program units, but they are protected against change in the library. Objects can be used as standards (classes) for other objects.

Object libraries simplify the sharing of reusable components. Reusing components enables you to:

- Apply standards to simple objects, such as buttons and items, for consistent look and feel. This also improves network performance by promoting similarities among objects, thus increasing the efficiency of message diffing.
- Reuse complex objects such as a Navigator.

In combination with SmartClasses, which are discussed later, object libraries support both of these requirements.

What Are Object Libraries? (continued)

Why Object Libraries Instead of Object Groups?

- Object libraries are external to the form, so are easily shared among form modules.
- Object libraries can contain individual items; for example, iconic buttons. The smallest unit accepted in an object group is a block.
- Object libraries accept PL/SQL program units.
- If you change an object in an object library, all forms that contain the subclassed object reflect the change.

Benefits of the Object Library

- **Simplifies the sharing and reuse of objects**
- **Provides control and enforcement of standards**
- **Promotes increased network performance**
- **Eliminates the need to maintain multiple referenced forms**



ORACLE

23-18

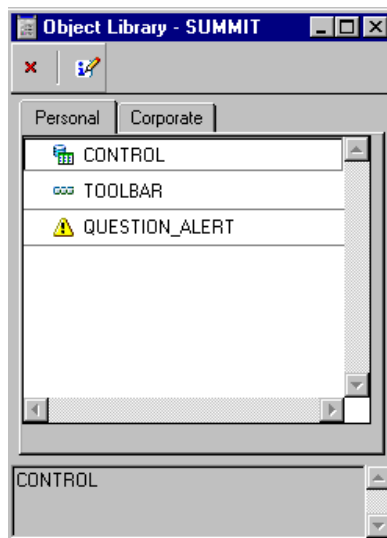
Copyright © 2004, Oracle. All rights reserved.

Benefits of the Object Library

There are several advantages to using object libraries to develop applications:

- Simplifies the sharing and reuse of objects
- Provides control and enforcement of standards
- Increases the efficiency of message diffing by promoting similarity of objects, thus increasing the performance of the application
- Eliminates the need to maintain multiple referenced forms

Working with Object Libraries



Object Libraries:

- Appear in the Navigator if they are open
- Are used with a simple tabbed interface
- Are populated by dragging Form objects to tab page
- Are saved to .olb file

ORACLE

23-19

Copyright © 2004, Oracle. All rights reserved.

Working with Object Libraries

Object libraries appear in the Navigator if they are open. You can create, open, and close object libraries like other modules. Forms Builder automatically opens all object libraries that were open when you last closed Forms Builder.

It is easy to use object libraries with a simple tabbed interface. Using the Edit menu, you can add or remove tab pages that help you to create your own groups. You can save object libraries to a file system as .olb files.

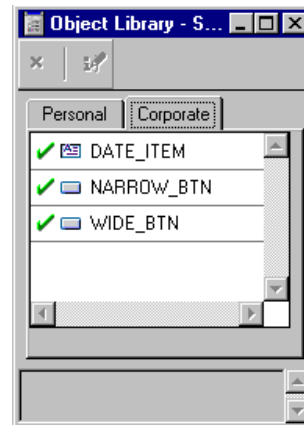
Note: You cannot modify objects inside the object library itself. To make changes, drag the object into a form, change it, and drag it back to the object library.

How to Populate an Object Library

1. Select Tools > Object Library to display the object library.
2. Drag objects from the Object Navigator into the object library.
3. You can edit the descriptive comment by clicking the Edit icon in the object library interface.

What Is a SmartClass?

- **A SmartClass:**
 - Is an object in an object library that is frequently used as a class
 - Can be applied easily and rapidly to existing objects
 - Can be defined in many object libraries
 - Is the preferred method to promote similarity among objects for performance
- You can have many SmartClasses of a given object type.



**Check indicates
a SmartClass**

What Are SmartClasses?

A SmartClass is a special member of an Object Library. It can be used to easily subclass existing objects in a form using the SmartClass option from the right mouse button popup menu. To use Object Library members which are not SmartClasses, you have to use the Subclass Information dialog available in the Property Palette of the form object that you are modifying.

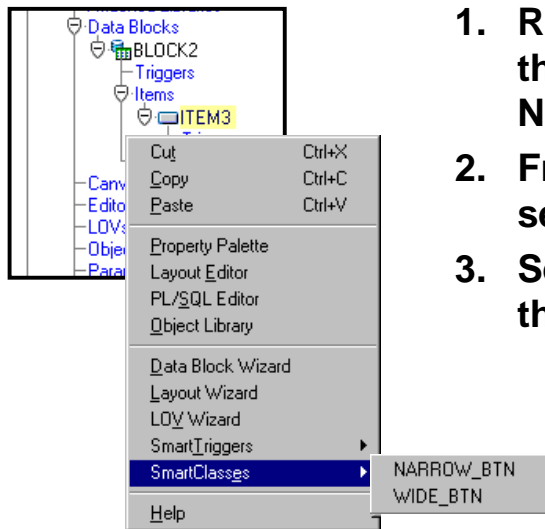
If you frequently use certain objects as standards, such as standard buttons, date items, and alerts, you can mark them as SmartClasses by selecting each object in the object library and choosing Edit > SmartClass.

You can mark many different objects, spread across multiple object libraries, as SmartClasses. Other than accepting default values for all object properties, using Smart Classes is the preferred method to promote similarities between objects for efficiency of message diffing, resulting in better performance of applications.

You can also have many SmartClasses of a given object type, for example:

- Wide_button
- Narrow_button
- Small_iconic_button

Working with SmartClasses



1. Right-click an object in the Layout Editor or Navigator.
2. From the pop-up menu, select SmartClasses.
3. Select a class from the list.

ORACLE

23-21

Copyright © 2004, Oracle. All rights reserved.

Working with SmartClasses

To apply a SmartClass to a Forms object, perform the following steps:

1. Right-click an object in the Layout Editor or Navigator.
2. From the pop-up menu, select SmartClasses. The SmartClasses pop-up menu lists all the SmartClasses from all open object libraries that have the same type as the object, and, for items, also have the same item type; for example, push button, text item..
3. Select a class for the object; it then becomes the parent class of the object. You can see its details in the Subclass Information dialog box in the object's Property Palette, just like any other subclassed object.

This mechanism makes it very easy to apply classes to existing objects.

Reusing PL/SQL

- **Triggers:**
 - Copy and paste text
 - Copy and paste within a module
 - Copy to or subclass from another module
 - Move to an object library
- **PL/SQL program units:**
 - Copy and paste text
 - Copy and paste within a module
 - Copy to or subclass in another module
 - Create a library module
 - Move to an object library

ORACLE

23-22

Copyright © 2004, Oracle. All rights reserved.

Reusing PL/SQL

PL/SQL in Triggers

You can reuse the PL/SQL in your triggers by:

- Copying and pasting, using the Edit menu
- Copying to another area of the current form module, using Copy and Paste on the menu of the right mouse button
- Copying to or subclassing from another form module, using drag and drop in the Object Navigator
- Moving the trigger to an object library

PL/SQL Program Units

Although triggers are the primary way to add programmatic control to a Forms Builder application, using PL/SQL program units supplement triggers, you can reuse code without having to retype it.

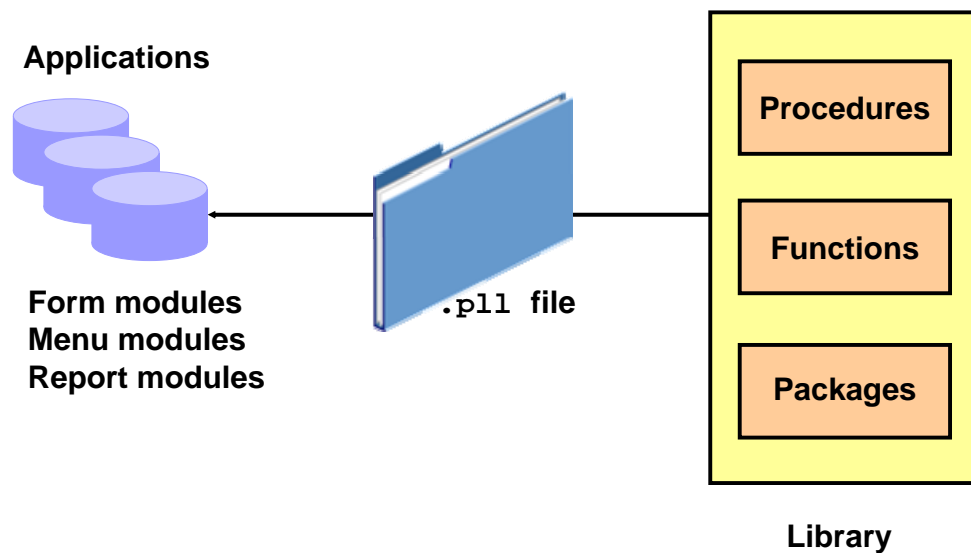
With Forms Builder, you can create PL/SQL program units to hold commonly used code. These PL/SQL program units can use parameters, which decrease the need to hard-code object names within the procedure body.

Reusing PL/SQL (continued)

You can reuse PL/SQL program units by:

- Copying and pasting, using the Edit menu
- Copying or subclassing to another form module, using drag and drop in the Object Navigator
- Creating a library module
- Moving the program unit to an object library

What Are PL/SQL Libraries?



What Are PL/SQL Libraries?

A *library* is a collection of PL/SQL program units, including procedures, functions, and packages. A single library can contain many program units that can be shared among the Oracle Forms Developer modules and applications that need to use them.

A library:

- Is produced as a separate module and stored in either a file or the database
- Provides a convenient means of storing client-side code and sharing it among applications
- Means that a single copy of program units can be used by many form, menu, or report modules
- Supports dynamic loading of program units

Scoping of Objects

Because libraries are compiled independently of the Forms modules that use them, bind variables in forms, menus, reports, and displays are outside the scope of the library. This means that you cannot directly refer to variables that are local to another module, because the compiler does not know about them when you compile the library program units.

Writing Code for Libraries

- **A library is a separate module, holding procedures, functions, and packages.**
- **Direct references to bind variables are not allowed.**
- **Use subprogram parameters for passing bind variables.**
- **Use functions, where appropriate, to return values.**

ORACLE

23-25

Copyright © 2004, Oracle. All rights reserved.

Writing Code for PL/SQL Libraries

There are two ways to avoid direct references to bind variables:

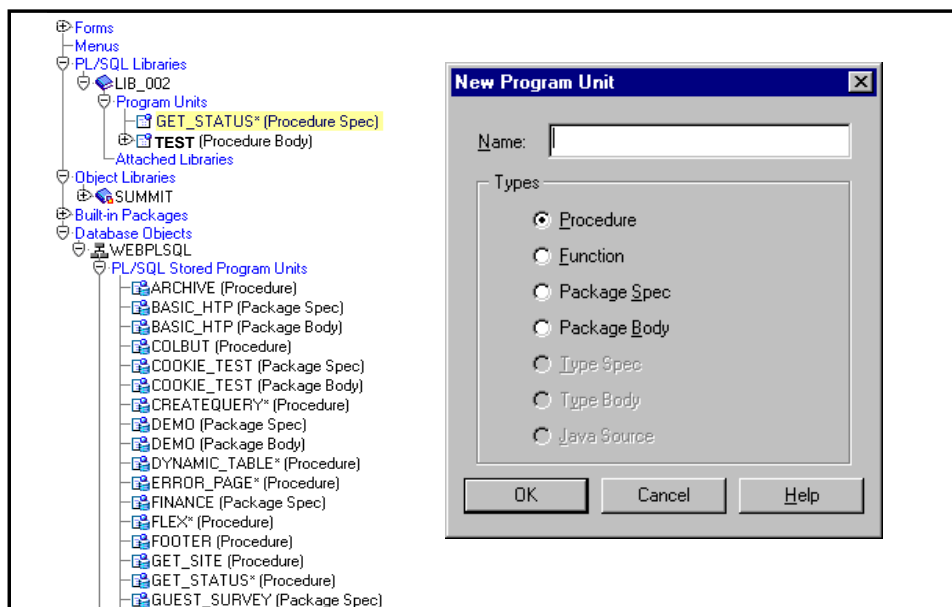
- You can refer to global variables and system variables in forms indirectly as quoted strings by using certain built-in subprograms.
- Write program units with IN and IN OUT parameters that are designed to accept references to bind variables. You can then pass the names of bind variables as parameters when calling the library program units from your Forms applications.

Example

Consider the second method listed above in the following library subprogram:

```
FUNCTION locate_emp(bind_value IN NUMBER) RETURN VARCHAR2 IS
    v_ename VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_ename FROM employees
    WHERE employee_id = bind_value;
    RETURN(v_ename);
END;
```

Creating Library Program Units



Working with PL/SQL Libraries

Creating a Library

You must first create libraries in the builder before you add program units. To do this, you can either:

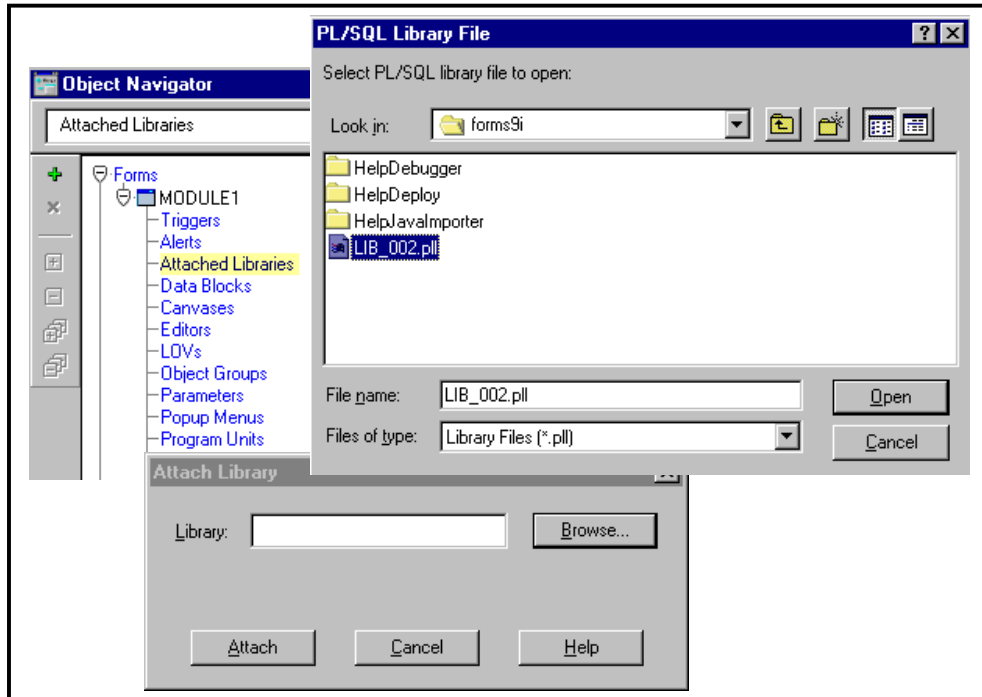
- Select File > New > PL/SQL Library from the menus (An entry for the new library then appears in the Navigator.)
- Select the Libraries node in the Object Navigator, and select the Create tool from the tool bar

There is a Program Units node within the library's hierarchy in the Navigator. From this node, you can create procedures, functions, package bodies, and specifications in the same way as in other modules.

How to Save the Library

1. With the context set on the library, select the Save option in Forms Builder.
2. Enter the name by which the library is to be saved.

Attach Library Dialog Box



ORACLE

23-27

Copyright © 2004, Oracle. All rights reserved.

Working with PL/SQL Libraries (continued)

How to Attach a Library

Before you can refer to a library program unit from a form, menu, report, or graphics, you must attach the library to the modules.

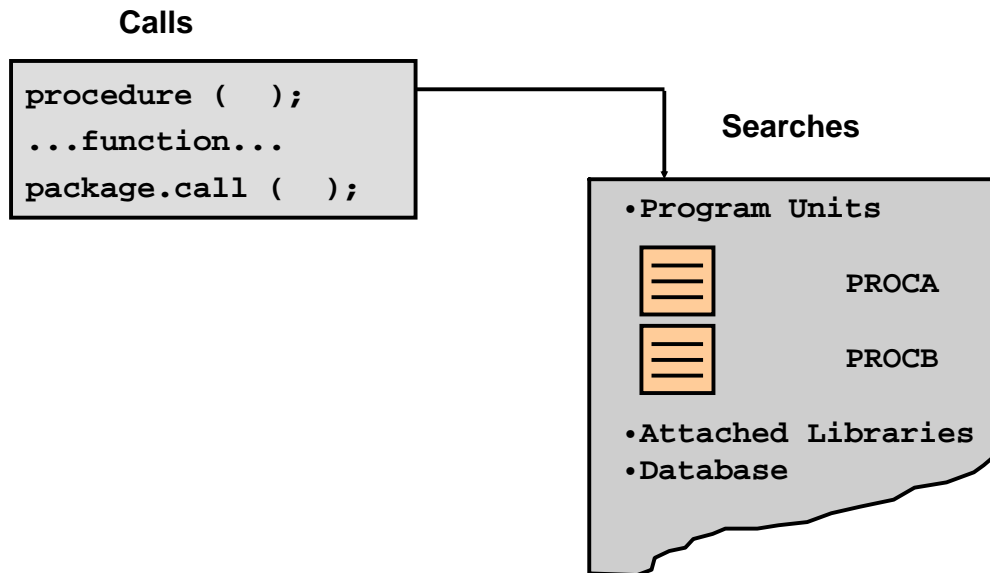
To attach a library to a module:

1. Open the module that needs to be attached to the library. This may be a form, menu, or another library module.
2. Expand the module and select the Attached Libraries node in the Navigator. When you select Create, the Attach Library dialog box appears.
3. In the Attach Library dialog box, specify the library's name.
4. Click Attach.
5. Save the module to which you have attached the library. This permanently records the library attachment in the definition of this module.

Detaching a Library

To later detach a library, simply delete the library entry from the list of Attached Libraries for that module, in the Navigator. That module will then no longer be able to reference the library program units, either in the Builder or at run time.

Calls and Searches



Referencing Attached Library Program Units

You refer to library program units in the same way as those that are defined locally, or stored in the database. Remember that objects declared in a package must be referenced with the package name as a prefix, whether or not they are part of a library.

Program units are searched for first in the calling module, then in the libraries that are attached to the calling module.

Example

Assume that the program units `report_totals`, `how_many_people`, and `pack5.del_emps` are defined in an attached library:

```
report_totals(:sub1);      --library procedure  
v_sum := how_many_people; --library function  
pack5.del_emps;          --library package procedure
```

Referencing Attached Library Program Units (continued)

When Several Libraries are Attached

You can attach several libraries to the same Oracle Forms Developer module. References are resolved by searching through libraries in the order in which they occur in the attachment list.

If two program units of the same name and type occur in different libraries in the attachment list, the one in the “higher” library will be executed, since it is located first.

Creating .PLX Files

The library .PLX file is a platform-specific executable that contains no source.

When you are ready to deploy your application, you will probably want to generate a version of your library that contains only the compiled p-code, without any source. You can generate a .PLX file from Forms Builder or from the command line.

Example

The following command creates a run-time library named `runlib1.plx` based on the open library `mylib.pll`:

```
GENERATE LIB mylib FILE runlib1;
```

Summary

In this lesson, you should have learned that:

- **You can reuse objects or code in the following ways:**
 - **Property Classes**
 - **Object Groups**
 - **Copying and subclassing**
 - **Object Libraries and SmartClasses**
- **To inherit properties from a property class, set an item's Subclass Information property.**
- **You can create an object group in one module to make it easy to reuse related objects in other modules.**

ORACLE

23-30

Copyright © 2004, Oracle. All rights reserved.

Summary

Forms provides a variety of methods for reusing objects and code. This lesson described how to use these methods.

Reasons to share objects and code:

- Increased productivity
- Increased modularity
- Decreased maintenance
- Maintaining standards
- Increased performance

Methods of sharing objects and code:

- Property classes
- Object groups
- Copying
- Subclassing
- Creating a library module
- Using object libraries and SmartClasses

Summary

- **Inheritance symbols in the Property Palette show whether the value is changed, inherited, overridden, or the default.**
- **You can drag objects from an object library or mark them as SmartClasses for even easier reuse.**
- **You can reuse PL/SQL code by:**
 - Copying and pasting in the PL/SQL Editor
 - Copying or subclassing
 - Defining program units to call the same code at multiple places within a module
 - Creating PL/SQL library to call the same code from multiple forms

ORACLE

Practice 23 Overview

This practice covers the following topics:

- **Creating an object group and using this object group in a new form module**
- **Using property classes**
- **Creating an object library and using this object library in a new form module**
- **Modifying an object in the object library and observing the effect on subclassed objects**
- **Setting and using SmartClasses**
- **Creating a PL/SQL program unit to be called from multiple triggers**

ORACLE

23-32

Copyright © 2004, Oracle. All rights reserved.

Practice 23 Overview

In this practice, you use an object group and an object library to copy Forms Builder objects from one form to another. You will also create a property class and use it to set multiple properties for several objects. You set SmartClasses in the object library and use these classes in the form module.

- Creating an object group and using this object group in a new form module
- Using property classes
- Creating an object library and using this object library in a new form module
- Modifying an object in the object library to see how the modification affects subclassed objects
- Setting and using SmartClasses
- Creating a PL/SQL program unit to be called from multiple triggers

Note: For solutions to this practice, see Practice 23 in Appendix A, “Practice Solutions.”

Practice 23

1. In the ORDGXX form, create an object group, called Stock_Objects, consisting of the INVENTORIES block, CV_INVENTORY canvas, and WIN_INVENTORY window.
2. Save the form.
3. Create a new form module and copy the Stock_Objects object group into it.
4. In the new form module, create a property class called ClassA. Include the following properties and settings:

Font Name:	Arial
Format Mask:	99,999
Font Size:	8
Justification:	Right
Delete Allowed:	No
Background Color:	DarkRed
Foreground Color:	Gray
5. Apply ClassA to CV_INVENTORY and the Quantity_on_Hand item.
6. Save the form module as STOCKXX.fmb, compile, and run the form and note the error.
7. Correct the error. Save, compile, and run the form again.
8. Create an object library and name it `summit_olb`.
Create two tabs in the object library called Personal and Corporate.
Add the CONTROL block, the Toolbar, and the Question_Alert from the Orders form to the Personal tab of the object library.
Save the object library as `summit.olb`.
9. Create a new form, and create a data block based on the DEPARTMENTS table, including all columns except DN. Use the Form layout style.
Drag the Toolbar canvas, CONTROL block, and Question_Alert from the object library into the new form, and select to **subclass** the objects.. For proper behavior, the DEPARTMENTS block must precede the CONTROL block in the Object Navigator
Some items are not applicable to this form.
Set the Canvas property for the following items to NULL: Image_Button, Stock_Button, Show_Help_Button, Product_Lov_Button, Hide_Help_Button, Product_Image, Total.
The code of some of the triggers does not apply to this form. Set the code for the When-Button-Pressed triggers for the above buttons to: `NULL;`
For the Total item, set the Calculation Mode and Summary Function properties to None, and set the Summarized Block property to Null.
Use Toolbar as the Horizontal Toolbar canvas for this form.
Set the Window property to WINDOW1 for the Toolbar canvas.
Set the Horizontal Toolbar Canvas property to TOOLBAR for the window.

Practice 23 (continued)

10. Save this form as DEPTGXX, compile, and run the form to test it.
11. Try to delete items on the Null canvas. What happens and why?
12. Change the Exit button of the Object Library's CONTROL block to have a gray background. Run the Departments form again to see that the Exit button is now gray.
13. Create two sample buttons, one for wide buttons and one for medium buttons, by means of width.
Create a sample date field. Set the width and the format mask to your preferred standard.
Drag these items to the Corporate tab of your object library.
Mark these items as SmartClasses.
Create a new form and a new data block in the form. Apply these SmartClasses in your form. Place the Toolbar canvas in the new form.
14. In the Orders form, note the similarity of the code in the Post-Query trigger of the Orders block and in the When-Validate-Item triggers of the Orders.Customer_Id and Orders.Sales_Rep_Id items. Move the similar code to PL/SQL program units and call the program units from the triggers; then run the form to test the changes.

Using WebUtil to Interact with the Client

24

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	30 minutes	Lecture
	30 minutes	Practice
	60 minutes	Total

Objectives

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

- **Describe the benefits of the WebUtil utility**
- **Integrate WebUtil into a form**
- **Use WebUtil to interact with a client machine**

ORACLE

24-2

Copyright © 2004, Oracle. All rights reserved.

Introduction

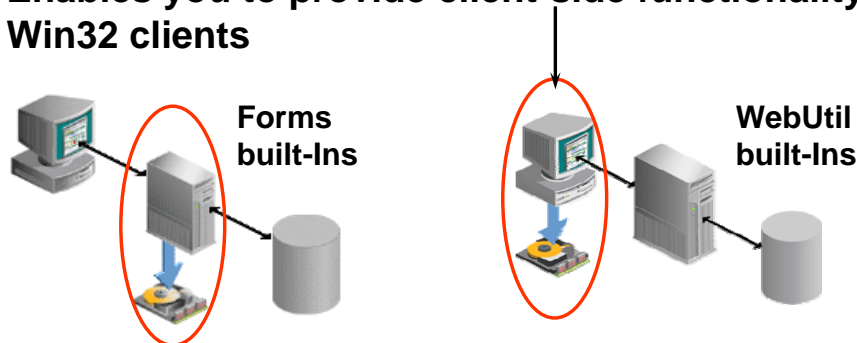
Overview

Forms built-in subprograms are called on the middle-tier application server. However, there are times when you want to be able to interact with the client machine. You can do so with JavaBeans and PJC's, but there is a utility called WebUtil that includes much prewritten functionality for client interaction. This lesson shows you how to use WebUtil to interface with the client machine.

WebUtil Overview

WebUtil is a utility that:

- **Enables you to provide client-side functionality on Win32 clients**



- **Consists of:**
 - Java classes
 - Forms objects
 - PL/SQL library

ORACLE

24-3

Copyright © 2004, Oracle. All rights reserved.

WebUtil Overview

Forms built-ins typically are executed on the application server machine. Some Forms built-ins interact with the machine to create or read a file, read an image file, or execute operating system commands. Although in some cases it is desirable to execute such built-ins on the application server machine, there is often a need to perform such functionality on the client. To do this you can use a JavaBean or PJC, but that requires that you either write or locate prewritten components and integrate each into Forms applications.

WebUtil is a utility that enables you to easily provide client-side functionality. It consists of a set of Java classes, Forms objects, and a PL/SQL API that enables you to execute the many Java functions of WebUtil without knowing Java.

WebUtil must be installed and configured separately from Oracle Developer Suite 10g. For installation and configuration instructions and much other information about WebUtil, see the WebUtil page on OTN:

<http://otn.oracle.com/products/forms/htdocs/webutil/webutil.htm>

Note: The client machine must be a Windows 32-bit platform, although the middle-tier server on which WebUtil is installed can be any platform on which Forms Services is supported.

Benefits of the WebUtil Utility

Why use WebUtil?

- **Developer has only to code in PL/SQL (no Java knowledge required)**
- **Free download (part of Forms 10g in a patch set)**
- **Easy to integrate into a Forms application**
- **Extensible**
- **WebUtil provides:**
 - **Client-server parity APIs**
 - **Client-server added value functions**
 - **Public functions**
 - **Utility functions**
 - **Internal functions**

ORACLE

24-4

Copyright © 2004, Oracle. All rights reserved.

Benefits of the WebUtil Utility

Why Use WebUtil?

Any Forms developer can use WebUtil to carry out complex tasks on client browser machines by simply coding PL/SQL.

WebUtil is available as a free download from OTN, and is planned to be included in a patch set for Forms 10g. It is very easy to integrate WebUtil into your Forms applications using its object group and PL/SQL library, and you can easily extend it by adding your own custom functionality while leveraging its basic structure.

You can use WebUtil to perform a multitude of tasks, enabling you to:

- Read and write text files on the client machine
- File transfer between the client, application server, and database
- Read client-side variables
- Manipulate client-side files
- Integrate with C code on the client
- Obtain information about the client
- Use a file selection dialog box on the client
- Run operating system commands on the client machine and call back into Forms
- Integrate with the browser, such as displaying messages to the browser window
- Perform OLE automation, such as using Word and Excel, on the client
- Read and write client side images

Benefits of the WebUtil Utility (continued)

What Functionality Is Available with WebUtil?

There is a wealth of functionality available in the utility, including the following:

- Client-server parity APIs that enable you to retrieve a file name from the client, read or write an image to or from the client, get information about the client machine, or perform HOST and TEXT_IO commands and OLE automation on the client (without the WebUtil functions, these built-ins execute on the application server machine.)
- Client-server added value functions (ported from d2kwutil, a client-server package):
 - CREATE_REGISTRY_KEY and DELETE_REGISTRY_KEY
 - GET_COMPUTER_NAME
 - GET_NET_CONNECTION
 - GET_TEMP_DIRECTORY, GET_WINDOWS_DIRECTORY, and GET_WORKING_DIRECTORY
 - GET_WINDOWS_USERNAME
 - READ_INI_FILE and WRITE_INI_FILE
 - READ_REGISTRY, WRITE_REGISTRY, and WRITE_REGISTRYEX

Note: Some of these may duplicate other WebUtil functions, but are provided as an easy way to migrate code that uses d2kwutil.

- Public functions: The core of WebUtil is a set of packages, each of which provides an API to implement certain functionality.
 - WebUtil_ClientInfo Package: Contains the following functions to obtain information about the client machine:

Function	Returns:
GET_DATE_TIME	Date and time on client machine
GET_FILE_SEPARATOR	Character used on client as file separator (“\” on Windows)
GET_HOST_NAME	Name of client machine
GET_IP_ADDRESS	IP address of client (string)
GET_JAVA_VERSION	JVM version that is running the Forms applet
GET_LANGUAGE	Language code of the client machine, such as de for German
GET_OPERATING_SYSTEM	Name of OS running the browser

Instructor Note

The many functions of WebUtil are listed here just to give students an idea of the capability of the utility. Do not cover them in detail. The client parity APIs are covered in more detail later in this lesson.

Benefits of the WebUtil Utility (continued)

Function	Returns:
GET_PATH_SEPARATOR	Character used on client to separate directory locations on paths (“;” on Windows)
GET_SYSTEM_PROPERTY	Any Java system property
GET_TIME_ZONE	Time zone of client machine
GET_USER_NAME	Name of user logged in to the client

- WebUtil_C_API Package: Contains the following functions to call into C libraries on the client:

Function	Purpose:
IsSupported	Returns True if the client is a valid platform
RegisterFunction	Returns a handle to a specified C library function
DeregisterFunction	Deregisters function and frees client resources
Create_Parameter_List	Creates a parameter list to pass to C function
Destroy_Parameter_List	Deletes a parameter list and frees its resources
Add_Parameter	Adds a parameter to the parameter list
Get_Parameter_Number Get_Parameter_Ptr Get_Parameter_String	Typed functions to return parameter values
Rebind_Parameter	Change parameter values of the existing parameter list to reuse it
Invoke_*	Typed functions to execute a registered C function
Parameter_List_Count	Returns the number of parameter lists that have been created
Function_Count	Returns the number of functions that have been registered
Id_Null	Checks to see whether the various types used in the package are null

Benefits of the WebUtil Utility (continued)

- WebUtil_File Package: Contains a new type called FILE_LIST, which is a PL/SQL table used to pass back multiple file names; also contains the following APIs to manipulate files and directories on the client and to display file selection dialog boxes:

Function	Purpose:
COPY_FILE RENAME_FILE DELETE_FILE	Copy, rename, or delete a file and return a Boolean value to indicate success
CREATE_DIRECTORY	Creates the named directory if it does not exist; returns a Boolean value to indicate success
DIRECTORY_ROOT_LIST	Returns a FILE_LIST containing the directory roots on the client system (the drives on Windows)
DIRECTORY_FILTERED_LIST	Returns a list of files in a directory that you filter using wildcard characters (* and ?)
FILE_EXISTS	Returns a Boolean value indicating the existence of the specified file on the client
FILE_IS_DIRECTORY	Returns a Boolean value indicating whether the specified file is a directory
FILE_IS_HIDDEN	Returns a Boolean value indicating whether the specified file has its hidden attribute set
FILE_IS_READABLE FILE_IS_WRITABLE	Returns a Boolean value indicating whether the file can be read or written to
DIRECTORY_SELECTION_DIALOG	Displays a directory selection dialog and returns the selected directory
FILE_SELECTION_DIALOG	Enables definition of File Save or File Open dialog with configurable file filter and returns the selected file
FILE_MULTI_SELECTION_DIALOG	Enables definition of File Save or File Open dialog with configurable file filter and returns the selected files in a FILE_LIST
GET_FILE_SEPARATOR	Returns character used on the client machine as a file separator (“\” on Windows)
GET_PATH_SEPARATOR	Returns character used on client machine to separate directory locations on paths (“;” on Windows)

Benefits of the WebUtil Utility (continued)

- **WebUtil_File_Transfer Package:** The WebUtil_File_Transfer package contains APIs to transfer files to and from the client browser machine and to display a progress bar as the transfer occurs. The following APIs are included in the WebUtil_File_Transfer package:

Function	Purpose:
URL_TO_CLIENT URL_TO_CLIENT_WITH _PROGRESS	Transfers a file from a URL to the client machine (and displays a progress bar)
CLIENT_TO_DB CLIENT_TO_DB_WITH _PROGRESS	Uploads a file from the client to a database BLOB column (and displays a progress bar)
DB_TO_CLIENT DB_TO_CLIENT_WITH _PROGRESS	Downloads file from a BLOB column in the database to the client machine (and displays a progress bar)
CLIENT_TO_AS CLIENT_TO_AS_WITH _PROGRESS	Uploads a file from the client to the application server (with progress bar)
AS_TO_CLIENT AS_TO_CLIENT_WITH _PROGRESS	Transfers a file from the application server to the client (with a progress bar)
IN_PROGRESS	Returns True if an asynchronous update is in progress
ASYNCHRONOUS_UPLOAD _SUCCESS	Returns a Boolean value indicating whether an asynchronous upload succeeded
GET_WORK_AREA	Returns a work area directory on the application server that is private to the user
IS_AS_READABLE IS_AS_WRITABLE	Returns True if the rules defined in the WebUtil configuration allow the specified file to be read or written to

- **WebUtil_Session Package:** Provides a way to react to an interruption of the Forms session by defining a URL to which the user is redirected if the session ends crashes; contains the following:

Function	Purpose:
ENABLE_REDIRECT_ON _TIMEOUT	Enables the time-out monitor and the specification of a redirection URL
DISABLE_REDIRECT_ON _TIMEOUT	Switches off the monitor; if you do not call this function before EXIT_FORM, the redirection occurs even if the exit is normal

Benefits of the WebUtil Utility (continued)

- **WebUtil_Host Package:** Contains routines to execute commands on the client machine. Includes two types: `PROCESS_ID` to hold a reference to a client-side process, and `OUTPUT_ARRAY`, a PL/SQL table which holds the `VARCHAR2` output from a client-side command. The `WebUtil_Host` package also contains the following functions:

Function	Purpose:
HOST	Runs the specified command in a BLOCKING mode on the client and optionally returns the return code
BLOCKING	Runs the specified command in a BLOCKING mode on the client and optionally returns the process ID
NONBLOCKING	Runs the specified command in a NON-BLOCKING mode on the client and optionally returns the process ID
NONBLOCKING_WITH_CALLBACK	Runs the specified command in a NON-BLOCKING mode on the client and executes a supplied trigger when the process is complete
TERMINATE_PROCESS	Kills the specified process on the client
GET_RETURN_CODE	Returns the return code of a specified process as an integer
GET_STANDARD_OUTPUT	Returns an <code>OUTPUT_ARRAY</code> containing output that was sent to standard output by the specified client process
GET_STANDARD_ERROR	Returns an <code>OUTPUT_ARRAY</code> containing output that was sent to standard error by the specified client process
RELEASE_PROCESS	Frees resources of specified client process
GET_CALLBACK_PROCESS	Returns the process ID of the finished process when a callback trigger executes following a command called from <code>NonBlocking_With_Callback</code>
ID_NULL	Tests if a process ID is null
EQUALS	Tests whether two process IDs represent the same process

Benefits of the WebUtil Utility (continued)

- `WebUtil_Core` Package: Contains mostly private functions, but you can call the following functions:

Function	Purpose:
<code>IsError</code>	Checks whether the last WebUtil call succeeded
<code>ErrorCode</code>	Returns the last WebUtil error code
<code>ErrorText</code>	Returns the text of the last WebUtil error

- Utility functions: The following functions are not related to client integration, but they can be useful:

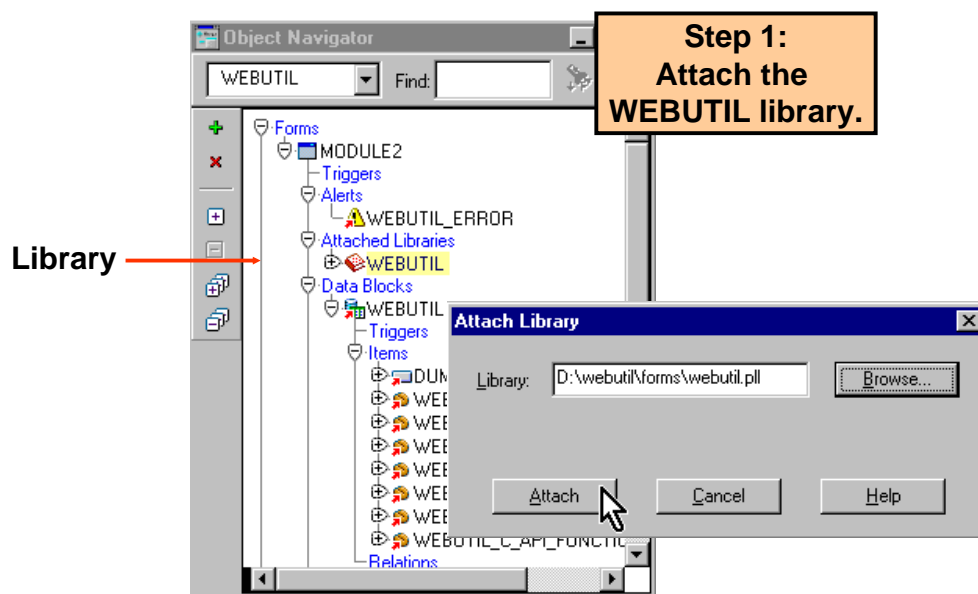
Function	Purpose:
<code>DelimStr</code>	Provides interaction with delimited strings
<code>Show_WebUtil_Information</code>	Calls the hidden WebUtil window to show the version of all WebUtil components
<code>WebUtil_Util</code>	Provides <code>BoolToStr()</code> function for converting Boolean to text

- Internal APIs that should not be called directly

For more information about WebUtil, including a sample form that showcases its functionality, see OTN at:

<http://otn.oracle.com/products/forms/htdocs/webutil/webutil.htm>

Integrating WebUtil into a Form



24-11

Copyright © 2004, Oracle. All rights reserved.

ORACLE

Integrating WebUtil into a Form

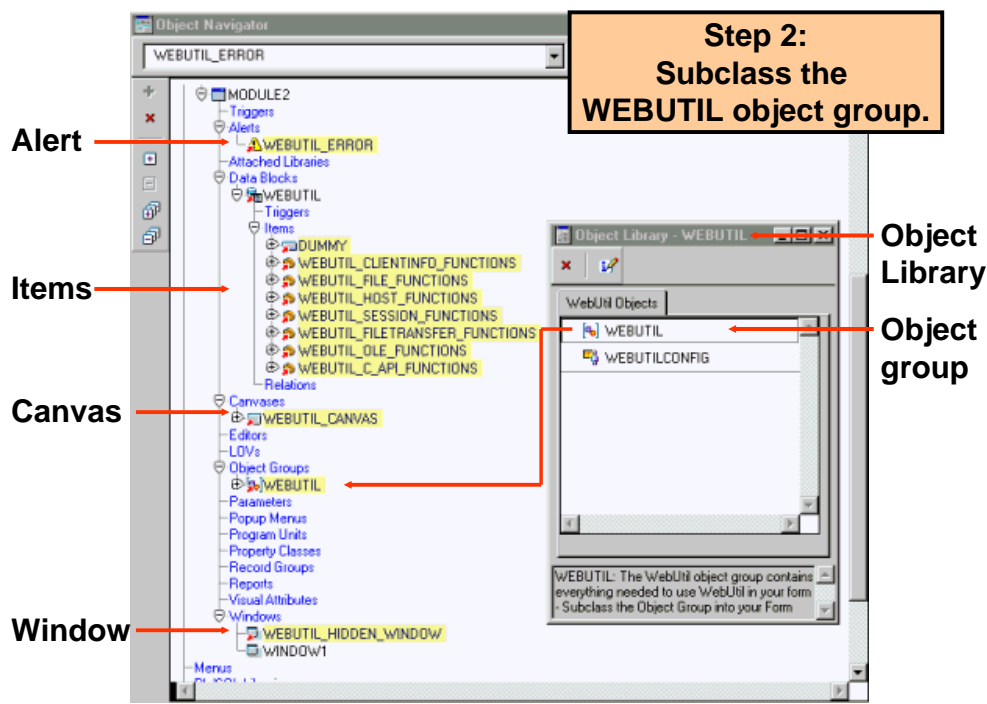
Step 1: Attaching the WebUtil Library

To use the functions of WebUtil in a Forms application, you first must attach the `webutil.pll` library to any module that will use the WebUtil PL/SQL API.

Instructor Note

Demonstration: In Forms Builder, open the form `WU_TEST.fmb` (the sample form from OTN) from the demo directory. Show students that the WEBUTIL library is attached to the form. Leave the form open for additional demonstrations.

Integrating WebUtil into a Form



24-12

Copyright © 2004, Oracle. All rights reserved.

ORACLE

Integrating WebUtil into a Form (continued)

Step 2: Subclassing WebUtil Forms Objects

Part of the WebUtil utility is a set of Forms objects contained in the `webutil.olb`. This object library contains an object group called WebUtil that you can subclass into your form. **You must ensure that WebUtil is the last block in the Object Navigator.**

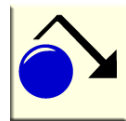
If you subclass the WebUtil object group into an empty form, you can see that it contains the following objects:

- A generic alert to display WebUtil error messages
- A data block with items, including a button and several bean area items to implement the JavaBeans (the bean area items are hidden because they do not provide a visual component on the form)
- A canvas to contain the items
- A window to display the canvas

Instructor Note

Demonstration: In the WU_TEST form, point out to students the above objects in the Object Navigator. Leave the form open for additional demonstrations.

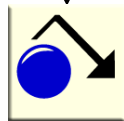
When to Use WebUtil Functionality



Pre-Form
When-New-Form-Instance
When-New-Block-Instance
(first block)



Form starts
JavaBeans are instantiated



Any trigger after form starts
and while form is running



ORACLE

24-13

Copyright © 2004, Oracle. All rights reserved.

When to Use WebUtil Functionality

After the WebUtil library has been attached to your form, you can start to add calls to the various PL/SQL APIs defined by the utility. However, there is an important restriction in the use of WebUtil functions—WebUtil can communicate with the client only after the Form has instantiated the WebUtil JavaBeans.

This means that you cannot call WebUtil functions before the user interface is rendered, so you should not use WebUtil functionality in triggers such as `Pre-Form`, `When-New-Form-Instance`, and `When-New-Block-Instance` for the first block in the form. In the case of the `When-New-Form-Instance` trigger it is possible, however, to call WebUtil functions after a call to the `SYNCHRONIZE` built-in has been issued, because this ensures that the user interface is rendered.

Further, you cannot call WebUtil functions after the user interface has been destroyed. For example, you should not use a WebUtil call in a `Post-Form` trigger.

Interacting with the Client

Forms Built-Ins / Packages	WebUtil Equivalents
HOST	CLIENT_HOST
GET_FILE_NAME	CLIENT_GET_FILE_NAME
READ_IMAGE_FILE	CLIENT_IMAGE.READ
WRITE_IMAGE_FILE	(WRITE)_IMAGE_FILE
OLE2	CLIENT_OLE2
TEXT_IO	CLIENT_TEXT_IO
TOOL_ENV	CLIENT_TOOL_ENV

ORACLE

24-14

Copyright © 2004, Oracle. All rights reserved.

Interacting with the Client

As previously mentioned, Forms built-ins work on the application server. For the most common Forms built-ins that you would want to use on the client, rather than on the application server, you can add a prefix to use the WebUtil equivalent.

These client-server parity APIs make it easy to provide similar functionality in applications that were written for client-server deployment by preceding those built-ins with “CLIENT_” or “CLIENT_IMAGE.”. Although this makes it easy to upgrade such applications, other WebUtil commands may provide similar, but better, functionality. The client-server parity APIs include the following:

- CLIENT_HOST
- CLIENT_GET_FILE_NAME

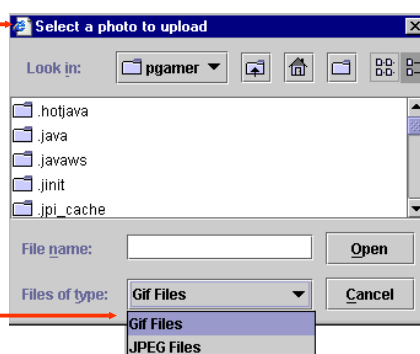
You can use READ_IMAGE_FILE on the client by calling the WebUtil equivalent contained in a package: CLIENT_IMAGE.READ_IMAGE_FILE.

In addition, there are certain Forms packages that you can use on the client with WebUtil:

- CLIENT_OLE2
- CLIENT_TEXT_IO
- CLIENT_TOOL_ENV

Example: Opening a File Dialog on the Client

```
DECLARE
  v_file VARCHAR2(250) := CLIENT_GET_FILE_NAME('', '',
    'Gif Files|*.gif|JPEG Files|*.jpg|',
    'Select a photo to upload', open_file, TRUE);
```



ORACLE

24-15

Copyright © 2004, Oracle. All rights reserved.

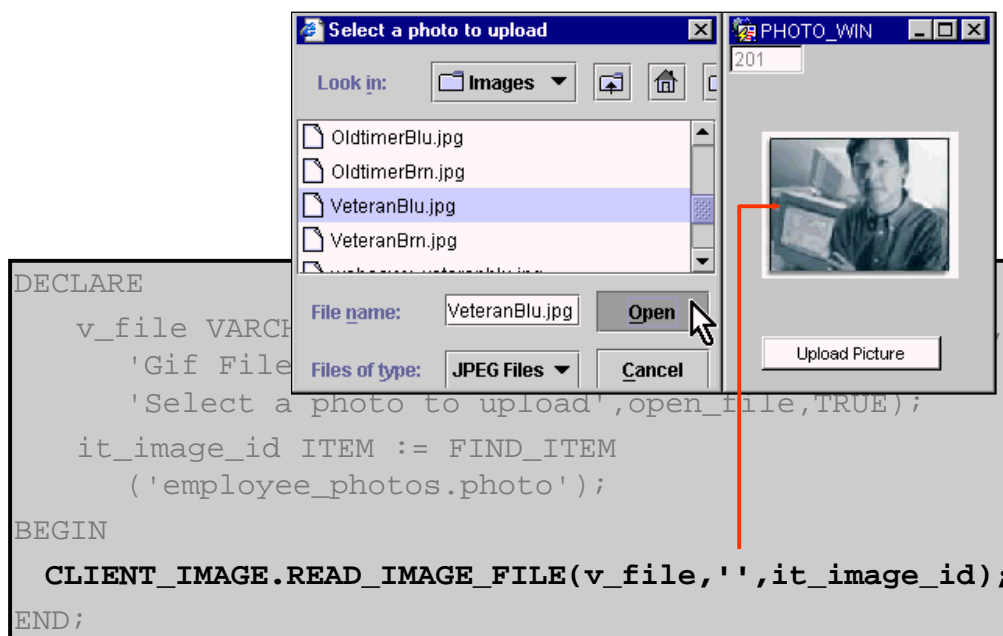
Example: Opening a File Dialog on the Client

To open a file dialog box on the client for selecting a file, you can use:
CLIENT_GET_FILE_NAME (**DIRECTORY_NAME** IN VARCHAR2,
FILE_NAME IN VARCHAR2, **FILE_FILTER** IN VARCHAR2,
MESSAGE IN VARCHAR2, **DIALOG_TYPE** IN NUMBER,
SELECT_FILE IN BOOLEAN) RETURN VARCHAR2;

The arguments for this WebUtil function are:

- **DIRECTORY_NAME:** Specifies the name of the directory containing the file you want to open. If **DIRECTORY_NAME** is NULL, subsequent invocations of the dialog box may open the last directory visited.
- **FILE_NAME:** Specifies the name of the file you want to open.
- **FILE_FILTER:** Specifies that only particular files be shown. On Windows, the default is "All Files (*.*)|*.*)" if NULL.
- **MESSAGE:** Specifies the title of the file upload dialog box
- **DIALOG_TYPE:** Specifies the intended dialog box to **OPEN_FILE** or **SAVE_FILE**.
- **SELECT_FILE:** Specifies whether the user is selecting files or directories. The default value is TRUE; if set to FALSE, the user must select a directory. If **DIALOG_TYPE** is set to **SAVE_FILE**, **SELECT_FILE** is internally set to TRUE.

Example: Reading an Image File into Forms from the Client



ORACLE

24-16

Copyright © 2004, Oracle. All rights reserved.

Example: Reading an Image File into Forms from the Client

You can use the `CLIENT_IMAGE` package to read or write image files. For example, the `CLIENT_IMAGE.READ_IMAGE_FILE` procedure reads an image from the client file system and displays it in the Forms image item:

```
CLIENT_IMAGE.READ_IMAGE_FILE (FILE_NAME VARCHAR2,  
FILE_TYPE VARCHAR2, ITEM_ID ITEM or ITEM_NAME VARCHAR2);
```

The arguments for this WebUtil procedure are:

- **FILE_NAME Valid file name:** The file name designation can include a full path statement appropriate to your operating system.
- **FILE_TYPE The valid image file type:** BMP, CALS, GIF, JFIF, JPG, PICT, RAS, TIFF, or TPIC. (Note: File type is optional, because Oracle Forms will attempt to deduce it from the source image file. To optimize performance, however, you should specify the file type.)
- **ITEM_ID:** The unique ID Oracle Forms assigns to the image item when it creates it.
- **ITEM_NAME:** The name you gave the image item when you created it.

Example: Writing Text Files on the Client

```
DECLARE
  v_dir VARCHAR2(250) := 'c:\temp';
  ft_tempfile CLIENT_TEXT_IO.FILE_TYPE;
begin
  ft_tempfile := CLIENT_TEXT_IO.FOPEN(v_dir ||
    '\tempdir.bat', 'w');
  CLIENT_TEXT_IO.PUT_LINE(ft_tempfile, 'dir ' ||
    v_dir || '> ' || v_dir || '\mydir.txt');
  CLIENT_TEXT_IO.PUT_LINE(ft_tempfile,
    'notepad ' || v_dir || '\mydir.txt');
  CLIENT_TEXT_IO.PUT_LINE(ft_tempfile, 'del ' ||
    v_dir || '\mydir.*');
  CLIENT_TEXT_IO.FCLOSE(ft_tempfile);
  CLIENT_HOST('cmd /c ' || v_dir || '\tempdir');
END;
```

1

2

3

4

ORACLE

24-17

Copyright © 2004, Oracle. All rights reserved.

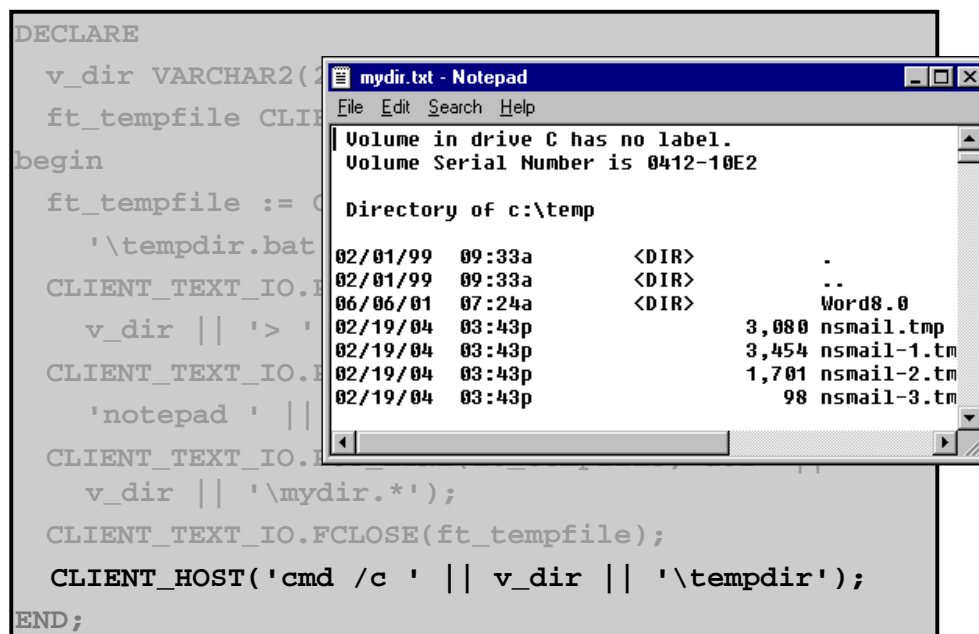
Example: Writing Text Files on the Client

With `CLIENT_TEXT_IO` commands, you can create or read text files on the client containing any ASCII text. This example creates a batch file on the client to display a directory listing of the `c:\temp` directory. The code does the following:

1. Declares a variable to hold a handle to the file
2. Opens a file named `tempdir.bat` on the client for writing
3. Writes the following lines of text to the file:

```
dir c:\temp> c:\temp\mydir.txt
notepad c:\temp\mydir.txt
del c:\temp\mydir.*
```
4. Closes the file

Example: Executing Operating System Commands on the Client



ORACLE

24-18

Copyright © 2004, Oracle. All rights reserved.

Example: Executing OS Commands on the Client

You can execute simple HOST commands on the client using the CLIENT_HOST command of WebUtil. The example shows running the batch file that was created in the previous example with CLIENT_TEXT_IO; cmd /c opens a command window and closes it after running the command. You must use cmd /c rather than running the command directly or it will not work. You can run any command that you would be able to execute from the Windows Start > Run menu.

Rather than creating the batch file with CLIENT_TEXT_IO, alternatively you can execute the commands it contains as follows:

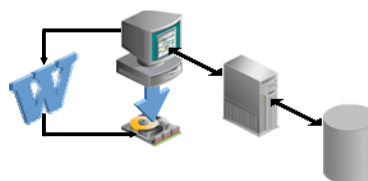
```
CLIENT_HOST('cmd /c dir c:\temp > c:\temp\mydir.txt');
CLIENT_HOST('cmd /c notepad c:\temp\mydir.txt');
CLIENT_HOST('cmd /c del c:\temp\mydir.*');
```

Note: You can obtain greater versatility in executing operating system commands by using the WebUtil_Host package. This enables you to execute commands synchronously or asynchronously and to call back into Forms from asynchronous commands when execution is complete.

Example: Performing OLE Automation on the Client

You can use the following for OLE automation:

<code>CLIENT_OLE2.OBJ_TYPE</code>	<code>CLIENT_OLE2.CREATE</code>
<code>CLIENT_OLE2.LIST_TYPE</code>	<code>CLIENT_OLE2._ARGLIST</code>
<code>CLIENT_OLE2.CREATE_OBJ</code>	<code>CLIENT_OLE2.ADD_ARG</code>
<code>CLIENT_OLE2.SET</code>	<code>CLIENT_OLE2.INVOKE</code>
<code>CLIENT_OLE2.GET_OBJ</code>	<code>CLIENT_OLE2.DESTROY</code>
<code>CLIENT_OLE2.INVOKE_OBJ</code>	<code>CLIENT_OLE2._ARGLIST</code>
	<code>CLIENT_OLE2.RELEASE_OBJ</code>



ORACLE

24-19

Copyright © 2004, Oracle. All rights reserved.

Example: Performing OLE Automation on the Client

You can use any OLE2 package on the client by preceding it with `CLIENT_`. You can see the list of the OLE2 package procedures and functions in the Forms Builder Object Navigator under the Built-in Packages node.

You can see examples of client OLE on OTN at <http://otn.oracle.com/products/forms/htdocs/webutil/webutil.htm>. The following example takes data from a form to construct a Word document and save it to the client machine:

```
DECLARE
    app CLIENT_OLE2.OBJ_TYPE;
    docs CLIENT_OLE2.OBJ_TYPE;
    doc CLIENT_OLE2.OBJ_TYPE;
    selection CLIENT_OLE2.OBJ_TYPE;
    args CLIENT_OLE2.LIST_TYPE;
BEGIN
    -- create a new document
    app := CLIENT_OLE2.CREATE_OBJ('Word.Application');
    CLIENT_OLE2.SET_PROPERTY(app, 'Visible', 1);
```

Example: Performing OLE Automation on the Client (continued)

```
docs := CLIENT_OLE2.GET_OBJ_PROPERTY(app, 'Documents');
doc := CLIENT_OLE2.INVOKE_OBJ(docs, 'add');

selection := CLIENT_OLE2.GET_OBJ_PROPERTY(app,
'Selection');
-- Skip 10 lines
args := CLIENT_OLE2.CREATE_ARGLIST;
CLIENT_OLE2.ADD_ARG(args,6);
FOR i IN 1..10 LOOP
    CLIENT_OLE2.INVOKE(selection, 'InsertBreak', args);
END LOOP;

-- insert data into new document

CLIENT_OLE2.SET_PROPERTY(selection, 'Text',
'RE: Order# ' || :orders.order_id);
FOR i in 1..2 LOOP
    CLIENT_OLE2.INVOKE(selection, 'EndKey');
    CLIENT_OLE2.INVOKE(selection, 'InsertBreak', args);
END LOOP;

CLIENT_OLE2.SET_PROPERTY(selection, 'Text',
'Dear ' || :customer_name || ':');
FOR i in 1..2 LOOP
    CLIENT_OLE2.INVOKE(selection, 'EndKey');
    CLIENT_OLE2.INVOKE(selection, 'InsertBreak', args);
END LOOP;

CLIENT_OLE2.SET_PROPERTY(selection, 'Text', 'Thank you for
your ' || 'order dated' ||
to_char(:orders.order_date, 'fmMonth DD, YYYY') ||
', in the amount of ' ||
to_char(:control.total, '$99,999.99') ||
'. We will process your order immediately and want you to '
|| 'know that we appreciate your business');

FOR i in 1..2 LOOP
    CLIENT_OLE2.INVOKE(selection, 'EndKey');
    CLIENT_OLE2.INVOKE(selection, 'InsertBreak', args);
END LOOP;
```


Example: Performing OLE Automation on the Client (continued)

```
CLIENT_OLE2.SET_PROPERTY(selection, 'Text', 'Sincerely,');
FOR i in 1..5 LOOP
    CLIENT_OLE2.INVOKE(selection, 'EndKey');
    CLIENT_OLE2.INVOKE(selection, 'InsertBreak', args);
END LOOP;

IF :orders.sales_rep_id IS NOT NULL THEN
    CLIENT_OLE2.SET_PROPERTY(selection, 'Text',
        :orders.sales_rep_name || ', Sales Representative');
    CLIENT_OLE2.INVOKE(selection, 'EndKey');
    CLIENT_OLE2.INVOKE(selection, 'InsertBreak', args);
END IF;

CLIENT_OLE2.SET_PROPERTY(selection, 'Text', 'Summit Office
Supply');

-- save document in temporary directory
CLIENT_OLE2.DESTROY_ARGLIST(args);
args := CLIENT_OLE2.CREATE_ARGLIST;
CLIENT_OLE2.ADD_ARG(args, 'letter_' ||
    :orders.order_id || '.doc');
CLIENT_OLE2.INVOKE(doc, 'SaveAs', args);
CLIENT_OLE2.DESTROY_ARGLIST(args);

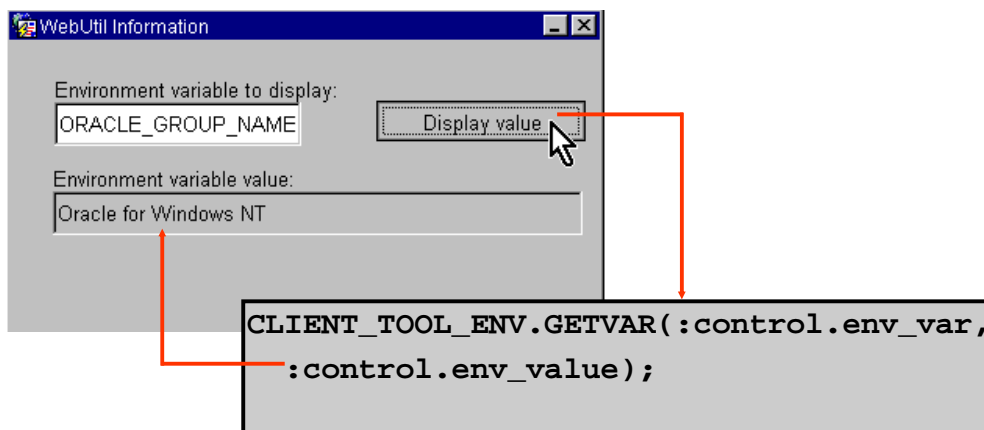
-- close example.tmp
args := CLIENT_OLE2.CREATE_ARGLIST;
CLIENT_OLE2.ADD_ARG(args, 0);
CLIENT_OLE2.INVOKE(doc, 'Close', args);
CLIENT_OLE2.DESTROY_ARGLIST(args);
CLIENT_OLE2.RELEASE_OBJ(selection);
CLIENT_OLE2.RELEASE_OBJ(doc);
CLIENT_OLE2.RELEASE_OBJ(docs);

-- exit MSWord
CLIENT_OLE2.INVOKE(app, 'Quit');
message('Letter created: letter_' ||
    :orders.order_id || '.doc');
END;
```

Instructor Note

This code is provided for student reference; there is no need to go over it in detail.

Example: Obtaining Environment Information about the Client



Example: Obtaining Environment Information about the Client

You can use the `CLIENT_TOOL_ENV.GETVAR` procedure from WebUtil to obtain information about registry variables on the client machine. You can obtain the values of any registry variables in the key, `HKEY_LOCAL_MACHINE > SOFTWARE > ORACLE`.

Note: You can obtain a greater variety of information about the client with the `WebUtil_ClientInfo` package.

Instructor Note

Demonstration: Run the `WU_TEST` form from Forms Builder. To show students additional functionality of WebUtil, demonstrate as many features as you have time for. You can find detailed instructions for running this demo in `WU_TEST.html` in the demo directory. Emphasize to students that they can download this demo from OTN to see examples of coding the various functions of WebUtil.

Summary

In this lesson, you should have learned that:

- **WebUtil is a free extensible utility that enables you to interact with the client machine**
- **Although WebUtil uses Java classes, you code in PL/SQL**
- **You integrate WebUtil into a form by attaching its PL/SQL library and using an object group from its object library; then you can use its functions after the form has started and while it is running**
- **With WebUtil, you can do the following on the client machine: open a file dialog box, read and write image or text files, execute operating system commands, perform OLE automation, and obtain information about the client machine**

ORACLE

24-23

Copyright © 2004, Oracle. All rights reserved.

Summary

WebUtil, included as part of Developer Suite 10g Patchset 1 and a free download from OTN prior to that, consists of a set of Java classes and a PL/SQL API. You can extend WebUtil by adding Java classes. The PL/SQL API enables you to do all coding within the form in PL/SQL.

After the middle tier has been configured for WebUtil, in order to use it in a form you need only add an object group from WebUtil's object library and attach WebUtil's PL/SQL library. You should not use WebUtil functions in triggers that fire as the form is starting up or after its user interface has been destroyed.

WebUtil includes much functionality. Some of the most common commands enable you to:

- Open a file dialog box on the client (`CLIENT_GET_FILE_NAME`)
- Read or write an image file on the client (`CLIENT_IMAGE` package)
- Read or write a text file on the client (`CLIENT_TEXT_IO`)
- Execute operating system commands (`CLIENT_HOST` or `WebUtil.HOST` package)
- Perform OLE automation on the client (`CLIENT_OLE2`)
- Obtain information about the client machine (`CLIENT_TOOL_ENV`)

Practice 24 Overview

This practice covers the following topics:

- **Integrating WebUtil with a form**
- **Using WebUtil functions to:**
 - **Open a file dialog box on the client**
 - **Read an image file from the client into the form**
 - **Obtain the value of a client environment variable**
 - **Create a file on the client**
 - **Open the file on the client with Notepad**
 - **Use OLE automation to create a form letter on the client**

ORACLE

24-24

Copyright © 2004, Oracle. All rights reserved.

Practice 24 Overview

In this practice, you integrate WebUtil with a form, and then use WebUtil to perform various functions on the client machine.

Note: For solutions to this practice, see Practice 24 in Appendix A, “Practice Solutions.”

Instructor Note

The version of WebUtil (1.0.2) used in this course does not enable use of the debugger. See bug 3497366, which is fixed in the production release of WebUtil.

Practice 24

1. In the ORDGXX form, integrate the WebUtil objects and library.
2. Save the form.
3. Change the Exit button in the Control block. Rename it: New_Image_Btn. Relabel it: New Image. Delete the current code for the button and write code to enable the user to choose a new JPEG image to display in the Product_Image item.
Hint: You will need to use `CLIENT_GET_FILENAME` and `CLIENT_IMAGE.READ_IMAGE_FILE`. You can import the code from `pr24_3.txt`.
4. Set the Forms Builder run-time preferences to use a WebUtil configuration that has been set up for you, `?config=webutil`, and then run the form to test it. Try to load one of the `.jpg` images in the lab directory.
Note: Because the image item is not a base table item, the new image is not saved when you exit the form.
5. If you have time, experiment with some of the other client/server parity APIs by adding additional code to the New_Image_Btn button. For example, you could:
 - a. Display a message on the status line that contains the value of the `ORACLE_HOME` environment variable. You can import the code from `pr24_4a.txt`.
 - b. Create a file called `hello.txt` in the `c:\temp` directory that contains the text "Hello World" and invoke Notepad to display this file. You can import the code from `pr24_4b.txt`.

Introducing Multiple Form Applications

25

ORACLE

Copyright © 2004, Oracle. All rights reserved.

Schedule:	Timing	Topic
	50 minutes	Lecture
	30 minutes	Practice
	80 minutes	Total

Objectives

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

- **Call one form from another form module**
- **Define multiple form functionality**
- **Share data among open forms**

ORACLE

25-2

Copyright © 2004, Oracle. All rights reserved.

Introduction

Overview

Oracle Forms Developer applications rarely consist of a single form document. This lesson introduces you to the ways in which you can link two or more forms.

Multiple Form Applications Overview

- **Behavior:**
 - Flexible navigation between windows
 - Single or multiple database connections
 - Transactions may span forms, if required
 - Commits in order of opening forms, starting with current form
- **Links:**
 - Data is exchanged by global variables, parameter lists, global record groups, or PL/SQL variables in shared libraries
 - Code is shared as required, through libraries and the database

ORACLE

25-3

Copyright © 2004, Oracle. All rights reserved.

Multiple Form Applications Overview

At the beginning of the course, we discussed the ability to design Forms applications where blocks are distributed over more than one form, producing a modular structure. A modular structure indicates the following:

- Component forms are only loaded in memory if they are needed.
- One form can be called from another, providing flexible combinations, as required.

How does the Application Behave?

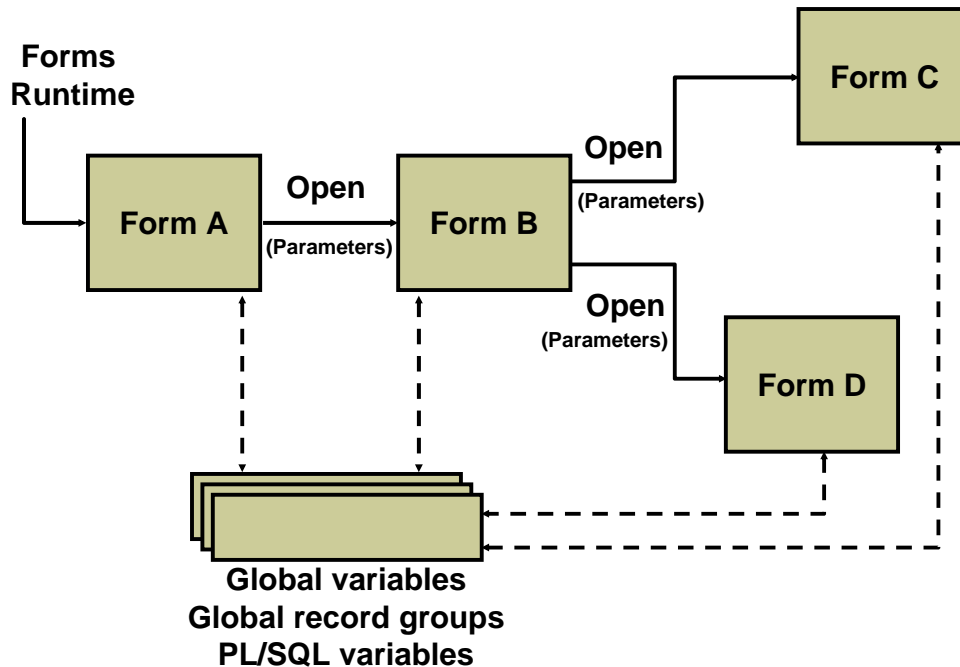
The first form module to run is specified before the Forms session begins, using Forms Runtime. Other form modules can be opened in the session by calling built-ins from triggers.

You can design forms to appear in separate windows, so the user can work with several forms concurrently in a session (when forms are invoked by the `OPEN_FORM` built-in). Users can then navigate between visible blocks of different forms, much as they can in a single form.

You can design forms for a Forms Runtime session according to the following conditions:

- Forms share the same database session, or open their own separate sessions.

Multiple Form Session



ORACLE

25-4

Copyright © 2004, Oracle. All rights reserved.

Multiple Form Applications Overview (continued)

- Database transactions are continued across forms, or ended before control is passed to another form. The commit sequence starts from the current form and follows the opening order of forms.
- Forms Builder provides the same menus across the application, or each form provides its own separate menus when it becomes the active form.

What Links the Forms Together?

Each form runs within the same Forms Runtime session, and Forms remembers the form that invoked each additional form. This chain of control is used when you exit a form or commit transactions.

Data can be exchanged between forms as follows:

- Through global variables, which span sessions
- Through parameter lists, for passing values between specific forms
- Through global record groups
- Through PL/SQL variables in shared libraries

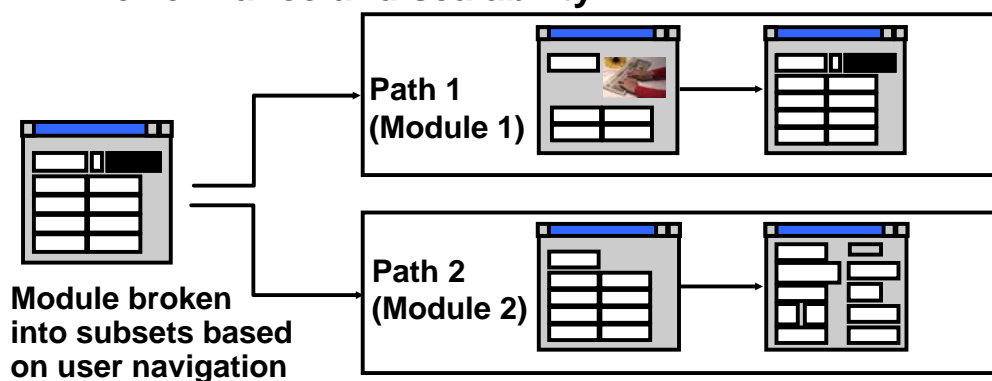
Code can be shared through the following:

- Library modules, by attaching them to each form as required
- Stored program units in the database

Benefits of Multiple Form Applications

Breaking your application into multiple forms offers the following advantages:

- Easier debugging
- Modularity
- Performance and scalability



ORACLE

25-5

Copyright © 2004, Oracle. All rights reserved.

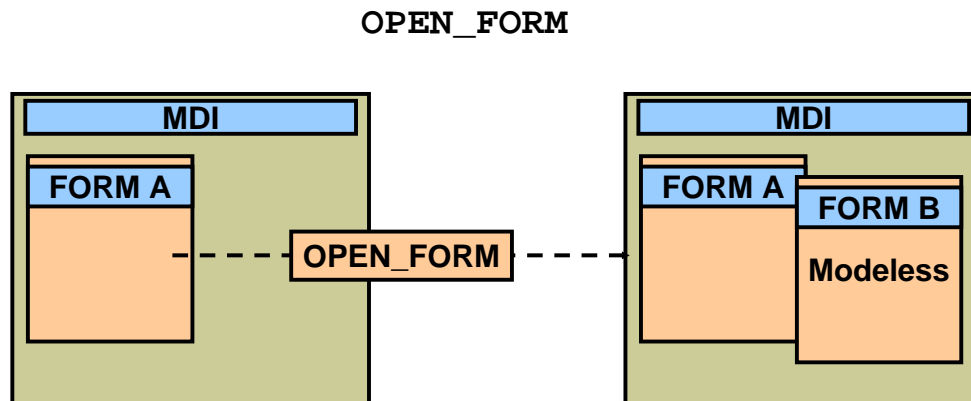
Multiple Form Applications Overview (continued)

Benefits of Multiple Form Applications

Since you can use different windows and canvases in a form module, wouldn't it be easier to put the entire application into one large form? Separating your application into multiple forms offers the following benefits:

- **Debugging:** It is easier to debug a small form; once a single form is working perfectly, you have only to integrate it into your application.
- **Logic modularity:** You break the application into pieces based on the logical functions the module is designed to perform. For example, you would not want the functions of human resources management and accounts receivable combined in a single form, and within each of these major functions are other logical divisions.
- **Network performance and scalability:** Sufficient information must be downloaded to describe the entire form before the form appears on the user's screen. Large forms take longer to download the relevant information, and fewer users can be supported on the given hardware. Break large applications into smaller components based on the likelihood of user navigation, enabling logical areas to be loaded on demand rather than all at once. This approach enables the module to start faster and uses less memory on the application server.

Starting Another Form Module



How to Start Another Form Module

When the first form in a Forms Runtime session has started, it can provide the user with facilities for starting additional forms. This can be performed by one of two methods:

- Calling a built-in procedure from a trigger in the form
- Calling a built-in procedure from a menu item in an attached menu

Built-In Procedures for Starting Another Form

You can use the OPEN_FORM built-in to start another form module from one that is already active. This is a restricted procedure, and cannot be called in the Enter Query mode. OPEN_FORM enables you to start another form in a modeless window, so the user can work in other running forms at the same time.

You can start another form using OPEN_FORM without passing control to it immediately, if required. This built-in also gives you the option to begin a separate database session for the new form.

Instructor Note

The broken line indicates that control need not pass immediately to the opened form (depending on the Activate_Mode argument).

How to Start Another Form Module (continued)

Syntax:

```
OPEN_FORM('form_name', activate_mode, session_mode,  
          data_mode, paramlist);
```

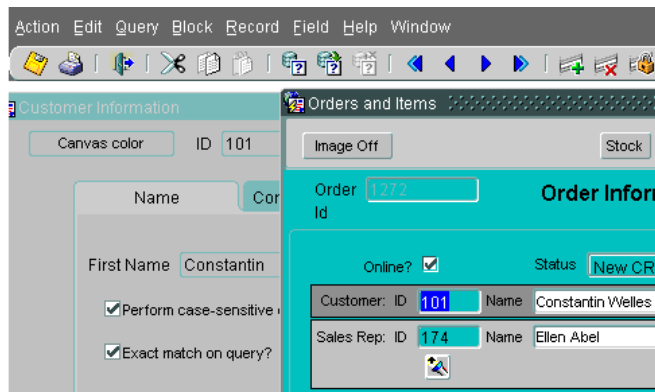
Parameter	Description
Form_Name	Filename of the executable module (without the .FMX suffix)
Activate_Mode	Either ACTIVATE (the default), or NO_ACTIVATE
Session_Mode	Either NO_SESSION (the default), or SESSION
Data_Mode	Either NO_SHARE_LIBRARY_DATA (the default) or SHARE_LIBRARY_DATA (Use this parameter to enable Forms to share data among forms that have identical libraries attached.)
Paramlist	Either the name (in quotes) or internal ID of a parameter list

There are two other built-ins that can call another form. This course concentrates on OPEN_FORM, which is considered the primary method to invoke multiple forms, rather than the CALL_FORM or NEW_FORM built-ins. For a discussion of these additional built-ins, see the OU Online Library course *Oracle9i Forms Developer: Enhance Usability*. You can access the online library from the Oracle Education Web page at: <http://www.oracle.com/education>

Defining Multiple Form Functionality

Summit application scenario:

- Run the CUSTOMERS and ORDERS forms in the same session, navigating freely between them.
- You can make changes in the same transaction across forms.
- All forms are visible together.



Defining Multiple Form Functionality

Using OPEN_FORM to Provide Forms in Multiple Windows

You can use OPEN_FORM to link form modules in an application and enable the user to work in them concurrently. Consider these requirements for the Summit application:

- The CUSTOMERS form must provide an option to start the ORDERS form in the same transaction, and orders for the current customer can be viewed, inserted, updated, and deleted.
- The user can see all open forms at the same time, and freely navigate between them to apply changes.
- Changes in all forms can be saved together.

Using OPEN_FORM to open both forms in the same session satisfies the requirements. However, having both forms open in the same session may have an undesired effect: If changes are made in the opened form, but not in the calling form, when saving the changes users may receive an error message indicating that no changes have been made. This error is produced by the calling form; changes in the opened form are saved successfully, but the error message may be confusing to users. To avoid this, you may decide to open the second form in a separate session, but then changes to each form would need to be saved separately.

Defining Multiple Form Functionality

Actions:

1. Define windows and positions for each form.
2. Plan shared data, such as global variables and their names.
3. Implement triggers to:
 - Open other forms
 - Initialize shared data from calling forms
 - Use shared data in opened forms

ORACLE

25-9

Copyright © 2004, Oracle. All rights reserved.

Defining Multiple Form Functionality (continued)

To provide this kind of functionality, perform the following steps:

1. Create each of the form modules. Plan where the windows of each module will appear in relation to those of other modules.
2. Plan how to share data among forms, such as identifying names for global variables. You need one for each item of data that is to be accessible across all the forms in the application. Note that each form must reference a global variable by the same name.
Note: You can also share data among forms using parameter lists, global record groups, or PL/SQL variables in shared libraries.
3. Plan and implement triggers to:
 - Open another form (You can do this from item interaction triggers, such as When-Button-Pressed, or from When-New-“*object*”-Instance triggers, or from a Key- trigger that fires on a keystroke or equivalent menu selection.)
 - Initialize shared data in calling forms so that values such as unique keys are accessible to other forms when they open. This might need to be done in more than one trigger, if the reference value changes in the calling form.
 - Make use of shared data in opened forms. For example, a Pre-Query trigger can use the contents of the global variable as query criteria.

Conditional Opening

Example

```
IF ID_NULL(FIND_FORM('ORDERS')) THEN
    OPEN_FORM('ORDERS');
ELSE
    GO_FORM('ORDERS');
END IF;
```

ORACLE

25-10

Copyright © 2004, Oracle. All rights reserved.

Conditional Opening

You can start up several instances of the same form, using `OPEN_FORM`, unless the application does appropriate tests before calling this built-in. For example, test a flag (global variable) set by an opened form at startup, which the opened form could reset on exit. This method may be unreliable, however, because if the form exits with an error the flag may not be properly reset. A better practice is to use the `FIND_FORM` built-in.

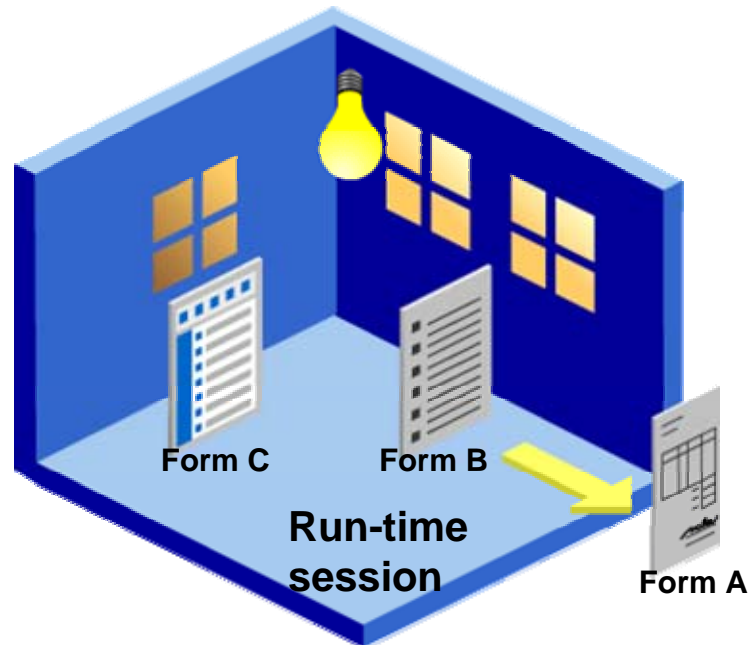
Here is a variation of the `When-Button-Pressed` trigger on `Orders_Button` in the `CUSTOMERS` form. If the `ORDERS` form is already running, it simply passes control to it, using `GO_FORM`.

```
.....
IF ID_NULL(FIND_FORM('ORDERS')) THEN
    OPEN_FORM('ORDERS');
ELSE
    GO_FORM('ORDERS');
END IF;
```

Note: If the name of the form module and its file name are different:

- Use the file name for `OPEN_FORM`: `OPEN_FORM('orderswk23');`
- Use the form module name for `GO_FORM`: `GO_FORM('orders');`

Closing the Session



“Will the last one out please turn off the lights?”

ORACLE

25-11

Copyright © 2004, Oracle. All rights reserved.

Closing Forms and Forms Run-Time Sessions

A form may close down and pass control back to its calling form under the following conditions:

- The user presses Exit or selects Exit from the Action menu.
- The `EXIT_FORM` built-in is executed from a trigger.

If the closing form is the only form still running in the Forms run-time session, the session will end as a result. When a multiple form session involves the `OPEN_FORM` built-in, it is possible that the last form to close is not the one that began the session.

Closing a Form with EXIT_FORM

- The default functionality is the same as for the Exit key.
- The Commit_Mode argument defines action on uncommitted changes.

```
ENTER ;
IF   :SYSTEM.FORM_STATUS = 'CHANGED' THEN
    EXIT_FORM( DO_COMMIT );
ELSE
    EXIT_FORM( NO_COMMIT );
END IF ;
```

ORACLE

25-12

Copyright © 2004, Oracle. All rights reserved.

Closing a Form with EXIT_FORM

When a form is closed, Forms checks to see whether there are any uncommitted changes. If there are, the user is prompted with the standard alert:

Do you want to save the changes you have made?

If you are closing a form with EXIT_FORM, the default functionality is the same as described above. You can, however, make the decision to commit (save) or roll back through the EXIT_FORM built-in, so the user is not asked. Typically, you might use this built-in from a Key-Exit or When-Button-Pressed trigger.

```
EXIT_FORM(commit_mode);
```

Parameter	Description
Commit_Mode	Defines what to do with uncommitted changes to the current form: <ul style="list-style-type: none">• ASK_COMMIT (the default) gives the decision to the user.• DO_COMMIT posts and commits changes across all forms for the current transaction.• NO_COMMIT validates and rolls back uncommitted changes in the current form.• NO_VALIDATE is the same as NO_COMMIT, but without validation.

Other Useful Triggers

Maintain referential links and synchronize data between forms:

- **In the parent form:**
 - `When-Validate-Item`
 - `When-New-Record-Instance`
- **In opened forms: `When>Create-Record`**
- **In any form: `When-Form-Navigate`**

ORACLE

25-13

Copyright © 2004, Oracle. All rights reserved.

Other Useful Triggers When Using `OPEN_FORM`

One drawback of designing applications with multiple forms is that you do not have the functionality available within a single form to automatically synchronize data such as master and detail records. You must provide your own coding to ensure that related forms remain synchronized.

Because `OPEN_FORM` enables the user to navigate among open forms, potentially changing and inserting records, you can use the triggers shown in the slide to help keep referential key values in step across forms.

Example

In the parent form (`CUSTOMERS`), this assignment to `GLOBAL.CUSTOMERID` can be performed in a `When-Validate-Item` trigger on `:CUSTOMERS.Customer_Id`, so that the global variable is kept up-to-date with an applied change by the user. The statement can also be issued from a `When-New-Record-Instance` trigger on the `CUSTOMERS` block, in case the user navigates to a line item record for a different customer.

```
:GLOBAL.customerid := :CUSTOMERS.customer_id;
```

Other Useful Triggers When Using OPEN_FORM (continued)

Example

In the opened form (ORDERS), a `When-Create-Record` trigger on the ORDERS block ensures that new records use the value of `GLOBAL.CUSTOMERID` as their default.

When items are assigned from this trigger, the record status remains `NEW`, so that the user can leave the record without completing it.

```
:ORDERS.customer_id := :GLOBAL.customerid;
```

Example

You may have a query-only form that displays employee IDs and names, with a button to open another form that has all the columns from the `EMPLOYEE` table so that users can insert new records.

You can use a `When-Form-Navigate` trigger in the query-only form to reexecute the query, so that when the user navigates back to that form the newly created records are included in the display.

Sharing Data Among Modules

You can pass data between modules using:

- **Global variables**
- **Parameter lists**
- **Global record groups**
- **PL/SQL package variables in shared libraries**

ORACLE

25-15

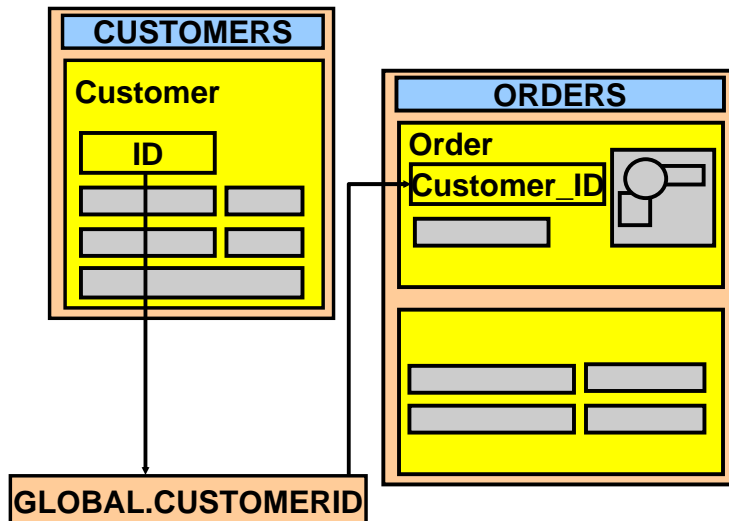
Copyright © 2004, Oracle. All rights reserved.

Sharing Data among Modules

Data can be exchanged between forms as follows:

- Through global variables, which span sessions
- Through parameter lists, for passing values between specific forms
- Through record groups created in one form with global scope
- Through PL/SQL variables in shared libraries

Linking by Global Variables



Defining Multiple Form Functionality

Planning Global Variables and Their Names

You need a global variable for each item of data that is used across the application.

Reminders:

- Global variables contain character data values, with a maximum of 255 characters.
- Each global variable is known by the same name to each form in the session.
- Global variables can be created by a PL/SQL assignment, or by the DEFAULT_VALUE built-in, which has no effect if the variable already exists.
- Attempting to read from a nonexistent global variable causes an error.

The scenario in the slide shows one global variable: GLOBAL.CUSTOMERID ensures that orders queried at the startup of the ORDERS form apply to the current customer.

Global Variables: Opening Another Form

Example

```
:GLOBAL.customerid := :CUSTOMERS.customer_id;  
OPEN_FORM('ORDERS');
```

Notes

- **Control passes immediately to the ORDERS form—no statements after OPEN_FORM are processed.**
- **If the Activate_Mode argument is set to NO_ACTIVATE, you retain control in the current form.**
- **The transaction continues unless it was explicitly committed before.**

ORACLE

25-17

Copyright © 2004, Oracle. All rights reserved.

Opening Another Form

- When you default the `Activate_Mode` argument in `OPEN_FORM`, control is passed immediately to the specified form, and any remaining statements after `OPEN_FORM` are not executed.
- If you set `Activate_Mode` to `NO_ACTIVATE`, control remains in the calling form, although the specified form starts up and the rest of the trigger is processed. Users can then navigate to the other form when they choose.
- If you want to end the current transaction before opening the next form, call the `COMMIT_FORM` built-in before `OPEN_FORM`. You can check to see if the value of `:SYSTEM.FORM_STATUS = 'CHANGED'` to decide whether a commit is needed. Alternatively, you can just post changes to the database with `POST`, then open the next form in the same transaction.

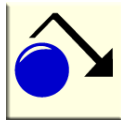
Opening the ORDERS Form from the CUSTOMERS Form

This `When-Button-Pressed` trigger on `:CONTROL.Orders_Button` opens the `ORDERS` form, and passes control immediately to it. `ORDERS` will use the same database session and transaction.

```
:GLOBAL.customerid := :CUSTOMERS.customer_id;  
OPEN_FORM('ORDERS');
```

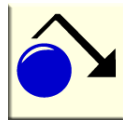
Global Variables: Restricted Query at Startup

When-New-Form-Instance



```
Execute_Query;
```

Pre-Query



```
:ORDERS.customer_id := :GLOBAL.customerid;
```

ORACLE

25-18

Copyright © 2004, Oracle. All rights reserved.

Performing a Restricted Query on Startup

To display a query automatically in the opened form, with data in context to the calling form, you produce two triggers:

- **When-New-Form-Instance:** This form-level trigger fires when the form is opened (regardless of whether control is passed to this form immediately or not). You can use this trigger to initiate a query by using the EXECUTE_QUERY built-in procedure. Executing a query fires a Pre-Query trigger if one is defined. The ORDERS form contains the following When-New-Form-Instance trigger:

```
EXECUTE_QUERY;
```
- **Pre-Query:** This is usually on the master block of the opened form. Because this trigger fires in Enter Query mode, it can populate items with values from global variables, which are then used as query criteria. This restriction applies for every other query performed on the block thereafter.

This Pre-Query trigger is on the ORDERS block of the ORDERS form:

```
:ORDERS.customer_id := :GLOBAL.customerid;
```


Assigning Global Variables in the Opened Form

- **DEFAULT_VALUE** ensures the existence of globals.
- You can use globals to communicate that the form is running.

Pre-Form example:

```
DEFAULT_VALUE(' ', 'GLOBAL.customerid');
```

ORACLE

25-19

Copyright © 2004, Oracle. All rights reserved.

Assigning Global Variables in the Opened Form

If, for some reason, a global variable has not been initialized before it is referenced in a called form, an error is reported:

```
FRM-40815: Variable GLOBAL.customerid does not exist.
```

You can provide independence, and ensure that global variables exist by using the `DEFAULT_VALUE` built-in when the form is opening.

Example

This Pre-Form trigger in the `ORDERS` form assigns a `NULL` value to `GLOBAL.CUSTOMERID` if it does not exist when the form starts. Because the Pre-Form trigger fires before record creation, and before all of the When-New-*“object”*-Instance triggers, it ensures existence of global variables at the earliest point.

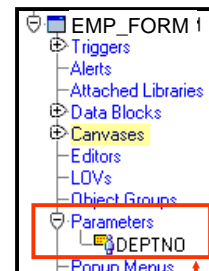
```
DEFAULT_VALUE(' ', 'GLOBAL.customerid');
```

The `ORDERS` form can now potentially be called without the `CUSTOMERS` form.

Linking by Parameter Lists

Parameters:

- Are form module objects
- Properties:
 - Name
 - Parameter Data Type
 - Maximum Length
 - Parameter Initial Value
- Can optionally receive a new value:



```
http://myhost:8889/forms90/f90servlet  
?form=emp.fmx&otherparams=deptno=140
```

Creating and Passing Parameter Lists

You can create any number of parameters in a form to hold data. Unlike global variables, parameters can be of any data type. However, their use in multi-form applications is limited by the fact that they are visible only to the form in which they are defined.

When you run a form, you can pass a value to the parameter as a name-value pair, such as `&otherparams=deptno=140`. Once the form receives the parameter value, a trigger can use that value to perform such functionality as restricting a query to records containing that value.

In the preceding example, you could construct a Pre-Query trigger to assign the value of `:parameter.deptno` to the `:employees.department_id` form item. When a query is executed on the Employees block, only the employees from department 140 would be retrieved.

You can also pass parameters to called forms programmatically by means of a parameter list.

Linking by Parameter Lists

Example:

```
DECLARE
    pl_id    ParamList;
    pl_name  VARCHAR2(10) := 'tempdata';
BEGIN
    pl_id := GET_PARAMETER_LIST(pl_name);
    1 IF ID_NULL(pl_id) THEN
        pl_id := CREATE_PARAMETER_LIST(pl_name);
    ELSE
        DELETE_PARAMETER(pl_id,'deptno');
    END IF;
    2 ADD_PARAMETER(pl_id,'deptno',TEXT_PARAMETER,
        to_char(:departments.department_id));
    3 OPEN_FORM('called_param',ACTIVATE,NO_SESSION,pl_id);
END;
```

ORACLE

25-21

Copyright © 2004, Oracle. All rights reserved.

Creating and Passing Parameter Lists (continued)

A parameter list is a named programmatic construct that is simply a list of parameter names and character values. The built-in `OPEN_FORM` optionally takes as an argument the name or ID of a parameter list. To receive this information, the called form must contain a parameter with the same name as each of those in the parameter list. In the called form, you can use the parameter by preceding its name with `:parameter`, for example `:parameter.customer_name`.

To use a parameter list, in the calling form:

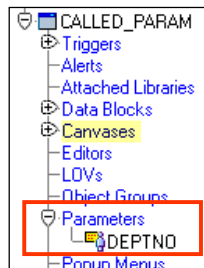
1. Create the parameter list (after checking that it does not already exist).
2. Add a parameter as a name/value pair text parameter. (There is another type of parameter, but it is not used to pass data between forms.)
3. Open the called form and pass the parameter list.

There are several built-ins that enable you to work with parameter lists, including:

```
GET_PARAMETER_LIST
CREATE_PARAMETER_LIST
DESTROY_PARAMETER_LIST
ADD_PARAMETER
DELETE_PARAMETER
```

Linking by Parameter Lists

Example: Called form



Create parameter
in the form

When-New-Form-Instance Trigger

```
IF :parameter.deptno IS NOT NULL THEN
  SET_BLOCK_PROPERTY('employees',
    DEFAULT_WHERE, 'department_id =
    ' | :parameter.deptno);
  SET_WINDOW_PROPERTY('window1',
    TITLE, 'Employees in Department
    | :parameter.deptno);
END IF;
GO_BLOCK('employees');
EXECUTE_QUERY;
```

Use parameter name
preceded by :parameter

ORACLE

25-22

Copyright © 2004, Oracle. All rights reserved.

Creating and Passing Parameter Lists (continued)

To use a parameter in the called form, you must first create the parameter in the form. Select the Parameters node in the Object Navigator and click Create, then change the name of the parameter to be the name that you are passing in the parameter list from the calling form. Once you have defined the parameter, you can use it in any of the called form's code by preceding the parameter name with `:parameter`. You can use the form independently of the calling form if you check to see if the parameter is null before using it or if you set the Parameter Initial Value property of the parameter.

Instructor Note

Demonstration: The example above is contained in the forms `CALLING_PARAM.fmb` and `CALLED_PARAM.fmb`. Open these forms to show students the code contained in the When-Button-Pressed trigger of the calling form and the When-New-Form-Instance trigger of the called form. Compile both forms. Run `CALLED_PARAM` in a browser and pass the parameter to it in a URL (`?form=called_param.fmx&otherparams=deptno=140`) to show that the code restricts the query based on the parameter that is passed. Then run the calling form and press the button to show that you can pass the parameter in a parameter list.

Linking by Global Record Groups

1. Create record group with global scope:

```
DECLARE
    rg_name      VARCHAR2(40) := 'LIST';
    rg_id        RecordGroup;
    Error_Flag   NUMBER;
BEGIN
    rg_id := FIND_GROUP(rg_name);
    IF ID_NULL(rg_id) THEN
        rg_id := CREATE_GROUP_FROM_QUERY('LIST',
            'Select last_name, to_char(employee_id)
            from employees', GLOBAL_SCOPE);
    END IF;
```

2. Populate record group:

```
Error_Flag := POPULATE_GROUP(rg_id);
```

3. Use record group in any form.

ORACLE

25-23

Copyright © 2004, Oracle. All rights reserved.

Sharing Global Record Groups among Forms

The `CREATE_GROUP_FROM_QUERY` built-in has as scope argument that defaults to `FORM_SCOPE`. However, if you use `GLOBAL_SCOPE`, the record group is global, and can be used within all forms in the application. Once created, a global record group persists for the remainder of the run time session. See Lesson 8 for a description of using a record group as a basis for a list of values (LOV). There are many other ways to use record groups that are not covered in this course.

To use a global record group:

1. Use `CREATE_GROUP_FROM_QUERY` to create the record group with `GLOBAL_SCOPE`.
2. Populate the record group with the `POPULATE_GROUP` built-in.
3. The record group is now available to any form in the same session.

Instructor Note

Demonstration: Open the `CALLING_RG.fmb` and `CALLED_RG.fmb` files. Show students the code in their When-New-Form-Instance triggers. Compile `CALLED_RG.fmb` (ignore the FRM-30351 error – this is just a warning because there are no list elements because the list is to be populated at run time). Run `CALLING_RG.fmb`.

Linking by Shared PL/SQL Variables

Advantages:

- Use less memory than global variables
- Can be of any data type

To use:

1. Create a PL/SQL library.
2. Create a package specification with variables.
3. Attach the library to multiple forms.
4. Set variable values in calling form.
5. OPEN_FORM with SHARE_LIBRARY_DATA option.
6. Use variables in opened form.

ORACLE

25-24

Copyright © 2004, Oracle. All rights reserved.

Linking by Package Variables in Shared PL/SQL Library

Perhaps the simplest and most efficient way to share data among forms is by using packaged variables in PL/SQL libraries. This enables you to use any data type, even user-defined types, to pass information between forms.

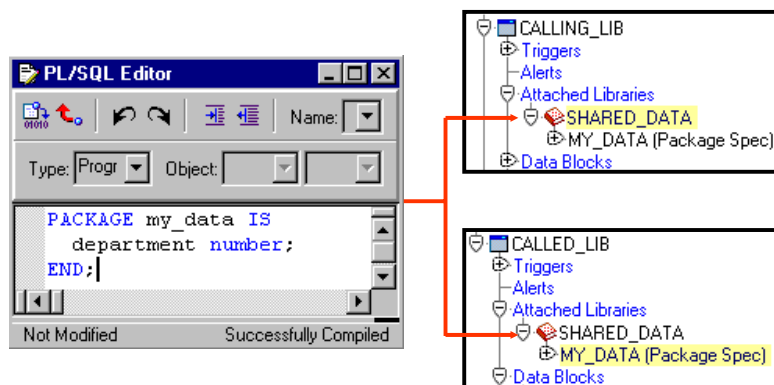
You create a library with at least a Package Specification that contains one or more variable declarations. You then attach that library to the calling and called forms.

When you open the called form with the SHARE_LIBRARY_DATA option, the variable value can be set and used by both open forms, making it very easy to share information among multiple forms.

Instructor Note

Demonstration: The example on the next page is contained in the forms `CALLING_LIB.fmb` and `CALLED_LIB.fmb`. Open these forms to show students the code contained in the When-Button-Pressed trigger of the calling form and the When-New-Form-Instance trigger of the called form. Compile the called form, then run the calling form and press the button to demonstrate that the code restricts the query of the called form based on the parameter that is passed.

Linking by Shared PL/SQL Variables



```
OPEN_FORM('called_lib',ACTIVATE,
          NO_SESSION,SHARE_LIBRARY_DATA);
```

ORACLE

25-25

Copyright © 2004, Oracle. All rights reserved.

Linking by Package Variables in Shared PL/SQL Library (continued)

For example, with the package shown above in a library that is attached to two forms, in a When-Button-Pressed trigger of the calling form:

```
my_data.department := :departments.department_id;
OPEN_FORM('called_lib',ACTIVATE,NO_SESSION,SHARE_LIBRARY_DATA);
```

And in the When-New-Form-Instance trigger of the called form:

```
IF my_data.department IS NOT NULL THEN
  SET_BLOCK_PROPERTY('employees',DEFAULT_WHERE,
    'department_id=' || TO_CHAR(my_data.department));
END IF;
GO_BLOCK('employees');
EXECUTE_QUERY;
```

Summary

In this lesson, you should have learned that:

- **OPEN_FORM is the primary method to call one form from another form module**
- **You define multiple form functionality such as:**
 - **Whether all forms run in the same session**
 - **Where the windows appear**
 - **Whether multiple forms should be open at once**
 - **Whether users should be able to navigate among open forms**
 - **How data will be shared among forms**

ORACLE

25-26

Copyright © 2004, Oracle. All rights reserved.

Summary

This lesson is an introduction lesson to multiple form applications. You should have learned how to:

- Open more than one form module in a Forms Runtime session
- Define how multiple forms in an application will function
- Pass information among forms

Summary

- **You can share data among open forms with:**
 - **Global variables, which span sessions**
 - **Parameter lists, for passing values between specific forms**
 - **Record groups created in one form with global scope**
 - **PL/SQL variables in shared libraries**

ORACLE

Practice 25 Overview

This practice covers the following topics:

- **Using a global variable to link ORDERS and CUSTOMERS forms**
- **Using built-ins to check whether the ORDERS form is running**
- **Using global variables to restrict a query in the ORDERS form**

ORACLE

25-28

Copyright © 2004, Oracle. All rights reserved.

Practice 25 Overview

In this practice, you produce a multiple form application by linking the CUSTGXX and the ORDGXX form modules.

- Linking ORDERS and CUSTOMERS forms by using a global variable
- Using built-ins to check whether the ORDERS form is running
- Using global variables to restrict a query in the ORDERS form

Note: For solutions to this practice, see Practice 25 in Appendix A, “Practice Solutions.”

Practice 25

1. In the ORDGXX form, create a Pre-Form trigger to ensure that a global variable called Customer_Id exists.
2. Add a trigger to ensure that queries on the ORDERS block are restricted by the value of GLOBAL.Customer_Id.
3. Save, compile, and run the form to test that it works as a stand-alone.
4. In the CUSTGXX form, create a CONTROL block button called Orders_Button and set appropriate properties. Place in on the CV_CUSTOMER canvas below the Customer_Id item.
5. Define a trigger for CONTROL.Orders_Button that initializes GLOBAL.Customer_Id with the current customer's ID, and then opens the ORDGXX form, passing control to it.
6. Save and compile each form. Run the CUSTGXX form and test the button to open the Orders form.
7. Change the window location of the ORDGXX form, if required.
8. Alter the Orders_Button trigger in CUSTGXX so that it does not open more than one instance of the Orders form, but uses GO_FORM to pass control to ORDGXX if the form is already running. Use the FIND_FORM built-in for this purpose.
9. If you navigate to a second customer record and click the Orders button, the Orders form still displays the records for the previous customer. Write a trigger to reexecute the query in the ORDERS form in this situation.
10. Write a When-Create-Record trigger on the ORDERS block that uses the value of GLOBAL.Customer_Id as the default value for ORDERS.Customer_Id.
11. Add code to the CUSTGXX form so that GLOBAL.Customer_Id is updated when the current Customer_Id changes.
12. Save and compile the ORDGXX form. Save, compile, and run the CUSTGXX form to test the functionality.
13. If you have time, you can modify the appearance of the ORDXX form to make it easier to read, similar to what you see in ORDERS . fmb.



The screenshot shows a form with two rows of data. The first row is for a Customer with ID 104 and Name Harrison Sutherland. The second row is for a Sales Rep with ID 155 and Name Oliver Tuvault. Below the Sales Rep name is a small icon of a person with a magnifying glass. The form has a light gray background and a dark border.

Customer: ID	104	Name	Harrison Sutherland
Sales Rep: ID	155	Name	Oliver Tuvault

