

Oracle Fusion Middleware 11g: Build Applications with ADF I

Volume I • Student Guide

D53979GC20

Edition 2.0

July 2010

D68160

ORACLE®

Authors

Kate Heap
Patrice Daux

**Technical Contributor
and Reviewer**

Joe Greenwald
Glenn Maslen

Editors

Aju Kumar
Daniel Milne

Graphic Designer

Satish Bettegowda

Publishers

Pavithran S. Adka
Jobi Varghese

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

I Introduction

- Course Objectives I-2
- Course Agenda: Day 1 I-3
- Course Agenda: Day 2 I-4
- Course Agenda: Day 3 I-5
- Course Agenda: Day 4 I-6
- Course Agenda: Day 5 I-7

1 Introduction to Oracle Fusion and Oracle ADF

- Objectives 1-2
- Examining Oracle Fusion Architecture 1-3
- Oracle Application Development Framework (ADF) 1-4
- Model-View-Controller Design Pattern 1-5
- How the Oracle ADF Framework Implements MVC 1-6
- Technology Choices for ADF BC Applications 1-7
- Introducing JDeveloper: Oracle's Java and Web Development Tool 1-8
- Obtaining Additional Information 1-9
- Summary 1-10

2 Getting Started with JDeveloper

- Objectives 2-2
- Describing the Benefits of Using JDeveloper 2-3
- Launching JDeveloper 2-5
- Defining Roles 2-6
- Using JDeveloper Features 2-7
- Using JDeveloper's Application Navigator 2-8
- Using JDeveloper's Database Navigator 2-10
- Using JDeveloper's Editors 2-11
- Using JDeveloper's Component Palette 2-13
- Using JDeveloper's Resource Palette 2-14
- Using JDeveloper's Structure Window 2-15
- Using JDeveloper's Property Inspector 2-16
- Using JDeveloper's Log Window 2-17
- Working with JDeveloper Windows 2-18
- Setting IDE Preferences 2-19
- Getting Started in JDeveloper 2-20

Creating an Application in JDeveloper	2-21
Using the Application Overview Page	2-23
The Application Overview Checklist	2-24
Creating a Project in JDeveloper	2-25
Creating Database Connections	2-27
Creating a Database Connection in JDeveloper	2-28
Describing the Course Application	2-30
Presenting the Storefront User Interface	2-31
Summary	2-32
Practice 2 Overview: Using JDeveloper	2-33

3 Building a Business Model with ADF Business Components

Objectives	3-2
Describing ADF Business Components (ADF BC)	3-3
ADF BC Implementation Architecture	3-4
Types of ADF Business Components	3-5
Creating ADF Business Components	3-6
Create Business Components from Tables Wizard: Entity Objects	3-7
Create Business Components from Tables Wizard: Updatable View Objects	3-8
Create Business Components from Tables Wizard: Read-Only View Objects	3-9
Create Business Components from Tables Wizard: Application Module	3-10
Create Business Components from Tables Wizard: Diagram	3-11
Examining Created Objects	3-12
Testing the Data Model	3-13
Exposing the Application Module to the User Interface	3-14
Summary	3-15
Practice 3 Overview: Building a Business Model	3-16

4 Querying and Persisting Data

Objectives	4-2
Using View Objects	4-3
Characteristics of a View Object (VO)	4-4
Creating View Objects for Queries	4-5
Testing View Objects with the Business Components Browser	4-11
Characteristics of an Entity Object (EO)	4-12
Using Entity Objects to Persist Data	4-13
Creating Entity Objects	4-14
Creating Entity Objects from Tables, Views, or Synonyms	4-16
Synchronizing an Entity Object with Changes to Its Database Table	4-17
Generating Database Tables from Entity Objects	4-18
Characteristics of Associations	4-19

Creating Associations	4-21
Association Types	4-22
Characteristics of Updatable View Objects	4-23
Creating Updatable View Objects	4-24
Creating Updatable View Objects: Attributes and Settings	4-25
Creating Updatable View Objects: Query	4-27
Creating Updatable View Objects: Additional Settings	4-29
Interaction Between Views and Entities: Retrieving Data	4-30
Interaction Between Views and Entities: Updating Data	4-31
Creating a Join View Object	4-32
Including Reference Entities in Join View Objects	4-33
Creating Master-Detail Relationships with View Objects	4-34
Linking View Objects	4-35
Comparing Join View Queries with View Links	4-36
Refactoring Objects	4-37
Summary	4-38
Practice 4 Overview: Creating Entity Objects and View Objects	4-39

5 Exposing Data

Objectives	5-2
Oracle ADF Application Module (AM)	5-3
Characteristics of an Application Module	5-4
Creating an Application Module	5-5
Defining the Data Model for the Application Module	5-6
Using Master-Detail View Objects in Application Modules	5-7
Determining the Size of an Application Module	5-8
Business Components Transactions	5-9
Using Nested Application Modules	5-11
Shared Application Modules	5-12
Application Module Pooling	5-14
Managing Application State	5-15
Role of ADF Model	5-16
Describing the Course Application: Database Objects	5-17
Describing the Course Application: View Objects	5-18
Describing the Course Application: Data Controls	5-20
Summary	5-21
Practice 5 Overview: Defining Application Modules	5-22

6 Declaratively Customizing Data Services

Objectives	6-2
Using Groovy	6-3

Using Groovy Syntax in ADF	6-4
Using Groovy Expressions for Validation	6-6
Internationalizing the Data Model	6-8
Editing Business Components	6-9
Modifying the Default Behavior of Entity Objects	6-10
Defining Attribute Control Hints	6-12
Modifying the Default Behavior of Entity Objects	6-14
Synchronizing with Trigger-Assigned Values	6-15
Modifying the Default Behavior of Entity Objects	6-17
Using Alternate Key Entity Constraints	6-18
Creating Alternate Key Entity Constraints	6-19
Editing View Objects	6-21
Modifying the Default Behavior of View Objects	6-22
Defining View Object Control Hints	6-24
Modifying the Default Behavior of View Objects	6-26
Performing Calculations	6-27
Modifying the Default Behavior of View Objects	6-28
Restricting and Reordering the Columns Retrieved by a Query	6-29
Modifying the Default Behavior of View Objects	6-30
Changing the Order of Queried Rows	6-31
Modifying the Default Behavior of View Objects	6-32
Restricting the Rows Retrieved by a Query	6-33
Using View Criteria (Structured WHERE Clauses)	6-34
Role of View Criteria in Search Forms	6-36
Using Parameterized WHERE Clauses	6-37
Using Named Bind Variables	6-38
Modifying the Default Behavior of View Objects	6-39
Retaining and Reusing a View Link Accessor Row Set	6-40
Modifying the Default Behavior of View Objects	6-41
Creating View Accessors	6-42
Using a List of Values (LOV)	6-43
Defining the View Accessor for the List of Values	6-44
Defining the List of Values	6-45
Modifying Application Modules	6-46
Changing the Locking Behavior of an Application Module	6-47
Summary	6-48
Practice 6 Overview: Declaratively Modifying Business Components	6-49

7 Programmatically Customizing Data Services

Objectives	7-2
Generating Java Classes for Adding Code	7-3

Programmatically Modifying the Default Behavior of Entity Objects	7-4
Supporting Entity Java Classes	7-5
Traversing Associations	7-7
Overriding Base Class Methods	7-8
Overriding Base Class Methods Example: Updating a Deleted Flag Instead of Deleting Rows	7-9
Overriding Base Class Methods Example: Eagerly Assigning Values from a Database Sequence	7-10
Programmatically Modifying the Default Behavior of View Objects	7-11
Supporting View Object Java Classes	7-12
Examining View Object Methods	7-13
Changing View Object WHERE or ORDER BY Clause at Run Time	7-16
Using Named Bind Variables at Run Time	7-18
Programmatically Retaining and Reusing a View Link Accessor Row Set	7-19
Traversing Links	7-20
Application Module Files	7-22
Centralizing Implementation Details	7-23
Adding Service Methods to an Application Module	7-25
Coding the Service Method	7-27
Publishing the Service Method	7-28
Testing Service Methods in the Business Components Browser	7-30
Accessing a Transaction	7-31
Committing Transactions	7-33
Customizing the Post Phase	7-34
Customizing the Commit Phase	7-35
Using Entity Objects and Associations Programmatically	7-36
Finding an Entity Object by Primary Key	7-37
Updating or Removing an Existing Entity Row	7-38
Creating a New Entity Row	7-39
Using Client APIs	7-41
Creating a Test Client	7-42
Using View Objects in Client Code	7-43
Using Query Results Programmatically	7-44
Using View Criteria Programmatically	7-45
Iterating Master-Detail Hierarchy	7-47
Finding a Row and Updating a Foreign Key Value	7-48
Creating a New Row	7-49
Summary	7-48
Practice 7 Overview: Programmatically Modifying Business Components	7-51

8 Validating User Input

Objectives 8-2

Validation Options for ADF BC Applications 8-3

Triggering Validation Execution 8-5

Handling Validation Errors 8-6

Specifying the Severity of an Error Message 8-7

Using Groovy Variables in Error Messages 8-8

Storing Error Messages as Translatable Strings 8-9

Defining Validation in the Business Services 8-10

Using Declarative Validation: Built-in Rules 8-11

Defining Declarative Validation 8-12

Using Declarative Validation: Built-in Rules 8-13

Using Declarative Built-in Rules: Collection Validator 8-14

Using Declarative Built-in Rules: Unique Key Validator 8-15

Using Declarative Validation: Built-in Rules 8-16

Using Declarative Built-in Rules: Compare Validator 8-17

Using Declarative Built-in Rules: Key Exists Validator 8-19

Using Declarative Built-in Rules: Length Validator 8-20

Using Declarative Built-in Rules: List Validator 8-21

Using Declarative Built-in Rules: Range Validator 8-22

Using Declarative Built-in Rules: Regular Expression Validator 8-23

Using Declarative Validation: Built-in Rules 8-24

Using Declarative Built-in Rules: Script Expression Validator 8-25

Using Declarative Custom Rules: Entity-Specific Rules (Method Validators) 8-26

Using Declarative Custom Rules: Creating an Entity-Specific Method Validator 8-27

Using Declarative Global Validation Rules 8-28

Creating the Java Class for a Global Rule 8-29

Examining the Generated Skeleton Code 8-30

Modifying the Code for the Global Rule 8-31

Assigning the Global Rule to an Object 8-33

Testing the Global Validation Rule 8-34

Using Programmatic Validation 8-35

Debugging Custom Validation Code with the JDeveloper Debugger 8-36

Using a Domain to Create Custom-Validated Data Types 8-37

Creating and Using a Domain 8-38

Coding Validation in a Domain 8-39

Specifying Validation Order 8-40

Summary 8-41

Practice 8 Overview: Implementing Validation 8-42

9 Troubleshooting ADF BC Applications	
Objectives	9-2
Troubleshooting the Business Service	9-3
Troubleshooting the UI	9-4
Using Logging and Diagnostics	9-5
Displaying Debug Messages to the Console	9-6
Java Logging	9-7
Core Java Logging	9-9
Using ADF Logging	9-10
Configuring ADF Logging	9-11
ODL Configuration Overview Editor	9-13
Creating Logging Configurations in JDeveloper	9-14
Viewing ODL Logs	9-15
Using Design-Time Code Validation	9-16
Auditing Java Code	9-17
Design-Time XML Validation	9-18
Design-Time JSPX Validation	9-19
Using Tools and Utilities	9-20
Testing Java Code with JUnit	9-21
Unit Testing with JUnit	9-22
Using JDeveloper's Profiler	9-23
Running the Profiler	9-24
Using Audit Profiles	9-25
Identifying Search Paths on Windows with FileMon	9-26
Using the JDeveloper Debugger	9-27
Understanding Breakpoint Types	9-28
Using Breakpoints	9-30
ADF Framework Debugging	9-31
Using the EL Evaluator	9-32
ADF Structure Window and ADF Data Window	9-33
Object Preferences	9-34
Using Oracle ADF Source Code for Debugging	9-35
Setting Up Oracle ADF Source Code for Debugging	9-36
Using Quick Javadoc	9-37
Setting Breakpoints in Source Code	9-38
Using Common Oracle ADF Breakpoints	9-39
Debugging Interactions with the Model Layer	9-40
Correcting Failures to Display Data	9-41
Correcting Failures to Invoke Actions and Methods	9-44
Debugging Life Cycle Events: Task Flows	9-46
Debugging Life Cycle Events: Parameters and Methods	9-47

Debugging Life Cycle Events: Switching Between the Main Page and Regions	9-48
Obtaining Help	9-49
Requesting Help	9-50
Summary	9-51
Practice 9 Overview: Troubleshooting	9-52

10 Understanding UI Technologies

Objectives	10-2
Enabling the World Wide Web with HTML and HTTP	10-3
Describing the Java Programming Language	10-4
Using Java as a Language for Web Development	10-6
What Are Servlets?	10-8
JavaServer Pages (JSP)	10-9
What Are JavaBeans?	10-10
What Is JavaServer Faces (JSF)?	10-11
JSF Key Concepts	10-13
JSF Component Model	10-14
JSF Multiple Renderers	10-15
Traditional Navigation	10-16
Defining Navigation by Using the JSF Controller	10-17
JSF Navigation: Example	10-18
Using JSF Components	10-19
Using JSF Managed Beans	10-20
Overview of JSF Page Life Cycle	10-21
Formal Phases of the JSF Life Cycle	10-22
Key Characteristics of Rich User Interfaces	10-24
Adding to JSF with ADF Faces	10-25
Using the ADF Controller	10-26
ADF Life Cycle Phases	10-27
Summary	10-28

11 Binding UI Components to Data

Objectives	11-2
Creating a JSF Page	11-3
Adding UI Components to the Page	11-5
Using the Component Palette	11-6
Using the Context Menu	11-7
Using the Data Controls Panel	11-8
Oracle ADF Data Controls	11-9
Describing the ADF Model Layer	11-10
Types of Data Bindings	11-11

Using Expression Language (EL)	11-13
Expression Language and Bindings	11-14
Creating and Editing Data Bindings	11-16
Rebinding: Example	11-17
Opening a Page Definition File	11-18
Editing Bindings in a Page Definition File	11-19
Editing Bindings from a Page	11-20
Tracing Data Binding: From Database to Databound Components	11-21
Tracing Data Binding: From AM to Data Control	11-22
Tracing Data Binding: Creating Databound Components	11-23
Tracing Data Binding: From Data Control to Databound Components	11-24
Examining Data Binding Objects and Metadata Files	11-27
Binding Existing Components to Data	11-29
Accessing Data Controls and Bindings Programmatically	11-30
Running and Testing the Page	11-31
Summary	11-32
Practice 11 Overview: Creating Databound Pages	11-33

12 Planning the User Interface

Objectives	12-2
Describing the Model-View-Controller (MVC) Design Pattern	12-3
Implementing MVC with the ADF Framework	12-4
Characteristics of ADF Task Flows	12-5
Characteristics of Unbounded ADF Task Flows	12-6
Working with Unbounded Task Flows	12-7
Characteristics of Bounded Task Flows	12-8
Comparing Unbounded and Bounded Task Flows	12-10
Bounded and Unbounded ADF Task Flows: Example	12-11
Creating an Unbounded Task Flow	12-12
Creating a Bounded Task Flow	12-13
Converting Task Flows	12-14
Using a Bounded Task Flow	12-15
Using ADF Task Flow Components	12-16
Defining ADF Control Flow Rules	12-19
Example of ADF Control Flow Rules	12-20
Using the Navigation Modeler to Define Control Flow	12-21
Using the Configuration Editor to Define Control Flow	12-22
Editing the.xml File to Define Control Flow	12-23
Using the Structure Window to Modify a Task Flow	12-24
Using Wildcards to Define Global Navigation	12-25
Using Routers for Conditional Navigation	12-27

Defining Router Activities	12-28
Calling Methods and Other Task Flows	12-29
Defining a Task Flow Return Activity	12-30
Making View Activities Bookmarkable (or Redirecting)	12-31
Adding UI Code	12-32
Incorporating Validation into the User Interface	12-33
Describing the Course Application: UI Functionality	12-34
Summary	12-35
Practice 12 Overview: Defining Task Flows	12-36

13 Adding Functionality to Pages

Objectives	13-2
Internationalization	13-3
Resource Bundles	13-4
Steps to Internationalize an Application	13-5
Automatically Creating a Resource Bundle	13-7
Automatically Creating a Text Resource	13-8
Using Component Facets	13-9
Using ADF Faces Rich Client Components	13-10
Using ADF Faces Input Components	13-11
Defining a List	13-13
Defining Lists at the Model Layer	13-14
Selecting a Value from a List	13-15
Selecting a Date	13-16
Using ADF Faces Table and Tree Components	13-18
Using Tables	13-19
Rendering (Stamping) the Table Data	13-20
Setting Table Attributes	13-21
Using Trees	13-23
Using Tree Tables	13-24
Providing Data for Trees	13-26
ADF Faces panelCollection Component	13-28
Using ADF Faces Output Components	13-29
Using ADF Faces Query Components	13-30
Using the af:query Component	13-31
Using the af:quickQuery Component	13-33
Creating a Query Search Form	13-35
Modifying Query Behavior	13-36
Using ADF Data Visualization Components	13-37
Visualizing Data	13-38

Summary 13-40
Practice 13 Overview: Using ADF Faces Components 13-41

14 Implementing Navigation on Pages

Objectives 14-2
Using ADF Faces Navigation Components 14-3
Performing Navigation 14-4
Using Buttons and Links 14-5
Defining Access Keys 14-6
Defining Tool Tips 14-8
Using Toolbars, Toolbar Buttons, and Toolboxes 14-9
Using Menus for Navigation 14-10
Creating Menus 14-11
Creating Pop-Up Menus 14-13
Creating Context Menus 14-14
Using a Navigation Pane 14-16
Using Breadcrumbs 14-17
Using Explicitly Defined Breadcrumbs 14-18
Using XML Menu Model for Dynamic Navigation Items 14-19
Creating an ADF Menu Model 14-20
Examining the ADF Menu Model 14-22
Binding the Navigation Pane to an XML Menu Model 14-23
Binding Breadcrumbs to an XML Menu Model 14-25
Defining a Sequence of Steps 14-26
Creating a Train 14-28
Skipping a Train Stop 14-29
Summary 14-30
Practice 14 Overview: Using ADF Faces Navigation Components 14-31

15 Achieving the Required Layout

Objectives 15-2
Using ADF Faces Layout Components 15-3
Adding Spaces and Lines: Spacer and Separator 15-5
Stretching Components 15-6
Enabling Automatic Component Stretching: Panel Splitter or Panel Stretch Layout 15-7
Stretching a Table Column 15-8
Creating Resizable Panes: Panel Splitter 15-9
Printing Layout Panel Content: Show Printable Page Behavior Operation 15-10
Creating Collapsible Panes: Panel Splitter 15-11
Creating Collapsible Panes: Panel Accordion 15-12

Panel Accordion Overflow	15-13
Setting Panel Accordion Properties	15-14
Arranging Items in Columns or Grids: Panel Form Layout	15-16
Creating Stacked Tabs: Panel Tabbed with Show Detail Item	15-18
Hiding and Displaying Groups of Content: Show Detail	15-19
Arranging Items Horizontally or Vertically, with Scrollbars: Panel Group Layout	15-21
Displaying Table Menus, Toolbars, and Status Bars: Panel Collection	15-23
Creating Titled Sections and Subsections: Panel Header	15-25
Grouping Related Components: Group	15-26
Displaying a Bulleted List: Panel List	15-27
Displaying Items in a Content Container Offset by Color: Panel Box	15-28
Arranging Content Around a Central Area: Panel Border Layout	15-29
Arranging Content Around a Central Area: Panel Stretch Layout	15-31
Using ADF Faces Skins	15-32
ADF Faces Skins	15-33
Using Dynamic Page Layout	15-34
Using Expression Language to Conditionally Display Components	15-35
Characteristics of Partial Page Rendering (PPR)	15-36
Enabling PPR Declaratively	15-37
Native PPR: Example	15-39
Declarative PPR: Example	15-40
Enabling PPR Programmatically	15-42
Enabling Automatic PPR	15-43
Conforming to PPR Guidelines	15-44
Summary	15-45
Practice 15 Overview: Using ADF Faces Layout Components	15-46

16 Ensuring Reusability

Objectives	16-2
Benefits of Reusability	16-3
Designing for Reuse	16-4
Using a Resource Catalog	16-5
Creating a Resource Catalog	16-6
Reusing Components	16-7
Creating an ADF Library	16-8
Adding an ADF Library to a Project by Using the Resource Palette	16-9
Removing an ADF Library from a Project	16-10
Restricting BC Visibility in Libraries	16-11
Types of Reusable Components	16-12
Using Task Flow Templates	16-13

Characteristics of Page Templates	16-14
Creating a Page Template	16-15
Editing a Page Template	16-17
Applying a Page Template to a Page	16-18
Characteristics of Declarative Components	16-19
Creating a Declarative Component	16-21
Using a Declarative Component on a Page	16-22
Characteristics of Page Fragments	16-23
Creating a Page Fragment	16-24
Using a Page Fragment on a Page	16-25
Characteristics of Regions	16-26
Wrapping a Task Flow as a Region	16-27
Converting a Bounded Task Flow to Use Page Fragments	16-28
Deciding Which to Use	16-29
Summary	16-30
Practice 16 Overview: Implementing Reusability	16-31

17 Passing Values Between UI Elements

Objectives	17-2
Holding Values in the Data Model (Business Components)	17-3
Holding Values in Managed Beans	17-4
Using Managed Properties	17-5
Managed Properties: Examples	17-6
Using Memory-Scoped Attributes	17-7
Memory Scope Duration with a Called Task Flow	17-9
Memory Scope Duration with a Region	17-10
Accessing Memory-Scoped Attribute Values	17-11
Using Memory-Scoped Attributes Without Writing Java Code	17-12
Overview of Parameters	17-13
Using Page Parameters	17-14
Job of the Page Parameter	17-15
Using Task Flow Parameters	17-16
Job of the Task Flow Parameter	17-18
Using Region Parameters	17-19
Job of the Region Parameter	17-20
Developing a Page Independently of a Task Flow	17-21
Using View Activity Parameters	17-22
Job of the View Activity Parameter	17-23
Summary: Passing a Value from a Containing Page to a Reusable Page Fragment in a Region	17-24
Passing Values to a Task Flow from a Task Flow Call Activity	17-25

Returning Values to a Calling Task Flow 17-27
Deciding Which Type of Parameter to Use 17-29
Summary 17-30
Practice 17 Overview: Passing Values Between Pages 17-31

18 Responding to Application Events

Objectives 18-2
Adding UI Code 18-3
Creating Managed Beans 18-4
Registering Existing Java Classes as Managed Beans 18-5
Configuring Managed Beans 18-6
Referencing Managed Beans 18-7
Describing JSF and ADF Life-Cycle Roles 18-8
Coordinating JSF and ADF Life Cycles 18-9
Specifying When to Refresh Binding Executables 18-11
Specifying Whether to Refresh Binding Executables 18-13
Describing Types of Events 18-14
Using Phase Listeners 18-15
Using Event Listeners 18-16
Responding to Action Events 18-17
Creating Action Methods 18-18
Using Action Listeners 18-19
Value Change Events 18-20
Listening for Value Change Events 18-21
Event and Listener Execution Order 18-22
ADF Faces Enhanced Event Handling 18-23
Using JavaScript in ADF Faces Applications 18-24
Other ADF Faces Server Events 18-25
Using Table Model Methods in a Selection Listener 18-27
Using Tree Model Methods in a Selection Listener 18-28
Additional AJAX Events 18-29
Characteristics of the Contextual Event Framework 18-30
Contextual Events: Overview 18-31
Using the Contextual Event Framework to Coordinate Page Regions 18-32
Using the Contextual Event Framework to Coordinate Page Regions: Step 1 18-33
Using the Contextual Event Framework to Coordinate Page Regions: Step 2 18-34
Using the Contextual Event Framework to Coordinate Page Regions: Step 3 18-35
Using the Contextual Event Framework to Coordinate Page Regions: Step 4 18-36
Using the Contextual Event Framework to Coordinate Page Regions: Step 5 18-37
Using the Contextual Events Tab to Define an Event 18-38
Using the Contextual Event Framework to Coordinate Page Regions: Step 6 18-39

Using the Contextual Event Framework to Coordinate Page Regions: Step 7	18-40
Using the Contextual Event Framework to Coordinate Page Regions: Step 8	18-41
Using the Contextual Events Tab to Map the Event	18-42
Using the Contextual Event Framework to Coordinate Page Regions: Step 9	18-44
Summary	18-45
Practice 18 Overview: Responding to Events	18-46
19 Implementing Transactional Capabilities	
Objectives	19-2
Handling Transactions with ADF BC	19-3
Default ADF Model Transactions	19-4
Transactions in Task Flows	19-5
Controlling Transactions in Task Flows	19-6
Transaction Support Features of Bounded Task Flows	19-7
Specifying Task Flow Transaction Start Options	19-8
Specifying Task Flow Return Options	19-9
Enabling Transactions on a Task Flow	19-11
Sharing Data Controls	19-13
Handling Transaction Exceptions	19-15
Designating an Exception Handler Activity	19-16
Defining Response to the Back Button	19-17
Saving for Later	19-19
Enabling Explicit Save for Later	19-21
Enabling Implicit Save for Later	19-23
Restoring Savepoints	19-24
Setting Global Save for Later Properties	19-26
Summary	19-27
Practice 19 Overview: Controlling Transactions	19-28
20 Implementing Security in ADF Applications	
Objectives	20-2
Benefits of Securing Web Applications	20-3
Examining Security Aspects	20-4
ADF Security Framework: Overview	20-5
Configure ADF Security Wizard: Configuring ADF Security Authentication	20-6
Configure ADF Security Wizard: Choosing the Authentication Type	20-7
Using Form-Based Authentication	20-8
Configure ADF Security Wizard: Choosing the Welcome Page	20-9
Configure ADF Security Wizard: Enabling ADF Authorization	20-10
Files Modified by Configure ADF Security Wizard: web.xml	20-11
Other Files Modified or Created by Configure ADF Security Wizard	20-12

Enabling Users to Access Resources	20-13
Defining Users and Roles in the Identity Store	20-14
Defining Security Policies	20-15
Defining Application Roles in the Policy Store	20-16
Assigning Identity Store Roles to Application Roles	20-17
Granting Permissions to Roles	20-18
Securing Groups of Pages (Bounded Task Flows)	20-19
Securing Individual Pages (Page Definitions)	20-20
ADF BC Model Authorization	20-21
Securing Row Data (Entity Objects or Attributes)	20-22
Granting Privileges on Entity Objects or Attributes	20-23
Application Authentication at Run Time	20-24
ADF Security: Implicit Authentication	20-25
ADF Security: Explicit Authentication	20-27
ADF Security: Authorization at Run Time	20-28
Programmatically Accessing ADF Security Context	20-29
Using Expression Language to Extend Security Capabilities	20-30
Using Global Security Expressions	20-31
Using a Security Proxy Bean	20-32
Summary	20-33
Practice 20 Overview: Implementing ADF Seurity	20-34

Appendix A: Modeling the Database Schema

Objectives	A-2
Modeling Database Schemas	A-3
Goals of Database Modeling	A-4
Characteristics of the JDeveloper Database Modeler	A-5
Database Modeling Tools in JDeveloper	A-6
Modeling Database Objects Offline	A-7
Creating a New Offline Database	A-8
Creating New Schema Objects in an Offline Database	A-9
Creating a Database Diagram	A-10
Importing Tables to the Diagram from an Offline Database	A-11
Editing Objects on the Diagram	A-12
Generating Changes from the Diagram to the Database	A-13
Reconciling Changes to the Database	A-14
Generating Changes from the Offline Database Object to the Database	A-15
Importing Database Objects Without a Diagram	A-16
Presenting the Storefront Schema	A-17
Summary	A-18
Practice Overview: Modeling the Schema for the Course Application	A-19

Appendix B: Deploying ADF Applications

Objectives	B-2
Steps in the Deployment Process	B-3
Configuring Deployment Options	B-5
Creating Deployment Profiles	B-6
Specifying Deployment Profile Options	B-7
Creating a Business Components Deployment Profile	B-8
Web Module Deployment	B-9
Typical Web Application Deployment Example	B-10
Example: Creating a WAR Deployment Profile for a UI Project	B-11
Example: Creating an EAR Deployment Profile for the Application	B-12
Using Deployment Descriptors	B-13
Steps in the Deployment Process	B-14
Preparing the Oracle WebLogic Server	B-15
Installing the ADF Runtime to the WebLogic Installation	B-16
Creating and Configuring the WebLogic Domain	B-17
Creating a JDBC Data Source	B-19
Configuring the Data Control to Use the Data Source	B-21
Steps in the Deployment Process	B-23
Creating a Connection to an Application Server	B-24
Example: Deploying the Application	B-26
Steps in the Deployment Process	B-27
Deploying the Application from the WebLogic Administration Server Console	B-28
Steps in the Deployment Process	B-29
Using Ant to Automate the Deployment Process	B-30
Creating an Ant Buildfile in JDeveloper	B-31
Defining Ant Deployment Tasks	B-32
Adding Elements to the Buildfile	B-33
Running Ant on Buildfile Targets	B-34
Creating an External Ant Tool	B-36
Implementing Security in Deployed Applications	B-37
Deployment Testing During Development	B-38
Deployment Testing for Production	B-39
Summary	B-40
Practice Overview: Deploying the Web Application	B-41

I

Introduction

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Course Objectives

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

- Build and customize a data model by using ADF Business Components
- Expose the data model in a Web application with a rich ADF Faces user interface
- Secure Web applications



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Course Aim

This course teaches you how to build Web applications by using ADF Business Components (ADF BC), ADF Faces components on JSF pages, and ADF data binding. You use Oracle JDeveloper 11g (11.1.1.2.0) to build, test, and secure Web applications. The course assumes that you are already familiar with the basics of the Java language.

The course also contains appendixes pertaining to modeling the database schema and deploying ADF applications, along with optional practices for these topics. These appendixes and practices are provided for student information, but are not presented in the classroom.

Course Agenda: Day 1

Basic ADF Business Components:

- Lesson 1: Introduction to Oracle Fusion and Oracle ADF
- Lesson 2: Getting Started with JDeveloper
- Lesson 3: Building a Business Model with ADF Business Components
- Lesson 4: Querying and Persisting Data
- Lesson 5: Exposing Data



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Course Agenda

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

Course Agenda: Day 2

Customization, Validation, and Troubleshooting:

- Lesson 6: Declaratively Customizing Data Services
- Lesson 7: Programmatically Customizing Data Services
- Lesson 8: Validating User Input
- Lesson 9: Troubleshooting ADF BC Applications



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Course Agenda: Day 3

Basic UI Concepts:

- Lesson 10: Understanding UI Technologies
- Lesson 11: Binding UI Components to Data
- Lesson 12: Planning the User Interface
- Lesson 13: Adding Functionality to Pages
- Lesson 14: Implementing Navigation on Pages



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Course Agenda: Day 4

Basic ADF Faces UI:

- Lesson 15: Achieving the Required Layout
- Lesson 16: Ensuring Reusability
- Lesson 17: Passing Values Between UI Elements



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Course Agenda: Day 5

Advanced ADF Faces:

- Lesson 18: Responding to Application Events
- Lesson 19: Implementing Transactional Capabilities
- Lesson 20: Implementing Security in ADF Applications



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Osi S.R.L. use only

1

Introduction to **Oracle Fusion and Oracle ADF**

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Describe the Fusion architecture
- Explain how Application Development Framework (ADF) fits into the Fusion architecture
- Describe the ADF technology stack (MVC)
- Find more information about ADF

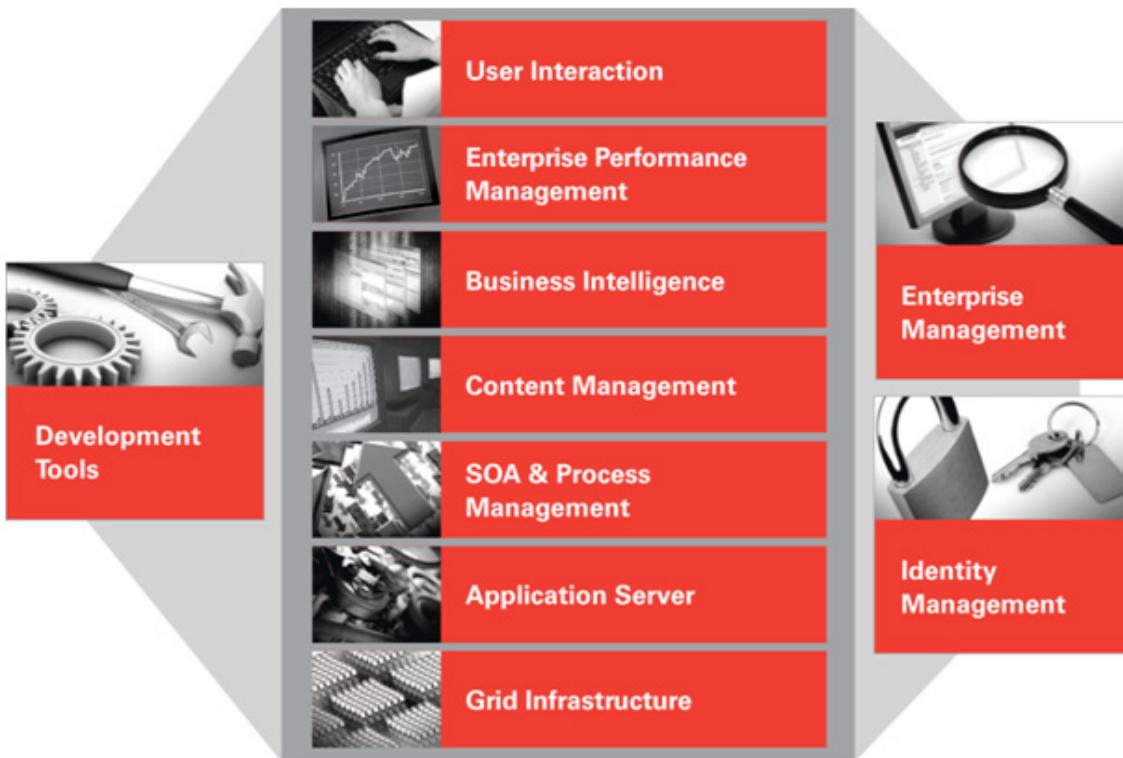


Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson gives you an overview of Fusion architecture and components. It introduces you to the essential concepts and structure of ADF and explain how it fits into the Fusion architecture.

Examining Oracle Fusion Architecture



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Components of Oracle Fusion Architecture

Oracle Fusion Middleware is a standards-based family of products that are often deployed and used in conjunction with one another to develop Java Platform, Enterprise Edition (Java EE) applications, providing the benefits of common security, management, deployment architecture, and development tools.

Oracle Fusion Middleware's architecture enables you to leverage your investments in any existing application, system, or technology.

Oracle Fusion Middleware includes the following best-of-breed technologies:

- **Oracle BPEL Process Manager:** A native Business Process Execution Language (BPEL) engine for Web service orchestration, enabling you to design, define, and execute business processes
- **Oracle Web Services Manager:** A console to secure and manage your Web services
- **Oracle Business Rules Engine:** A product to enable agile management of business rules
- **Oracle Enterprise Service Bus:** A standards-based product that connects existing IT systems and business partners as a set of services, and supports event-driven architectures
- **Oracle Business Activity Monitoring:** A product to report insight into business operations
- Oracle Services Registry: A Universal Description, Discovery and Integration (UDDI) v3-compliant registry
- **Oracle JDeveloper:** An integrated development environment for creating and composing applications in a unified toolset for all Oracle Fusion Middleware tools. Not only can you use Oracle JDeveloper to build applications, but Oracle developers use JDeveloper to build the other tools.

Oracle Application Development Framework (ADF)

- Is an end-to-end Java EE framework that is extensible
- Utilizes and adds value to the Java EE platform
- Abstracts Java EE complexity
- Provides declarative and visual development
- Enables developers to focus on the application, not the low-level infrastructure
- Creates reusable and maintainable code
- Uses metadata, simplifying the basic task of wiring user interfaces to services
- Implements Java EE best practices and design patterns, including MVC



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Oracle Application Development Framework (ADF)

Oracle ADF is an end-to-end application framework that builds on Java EE standards and open-source technologies to simplify and accelerate implementing Java EE applications. It is fully extensible and customizable by adding or modifying libraries.

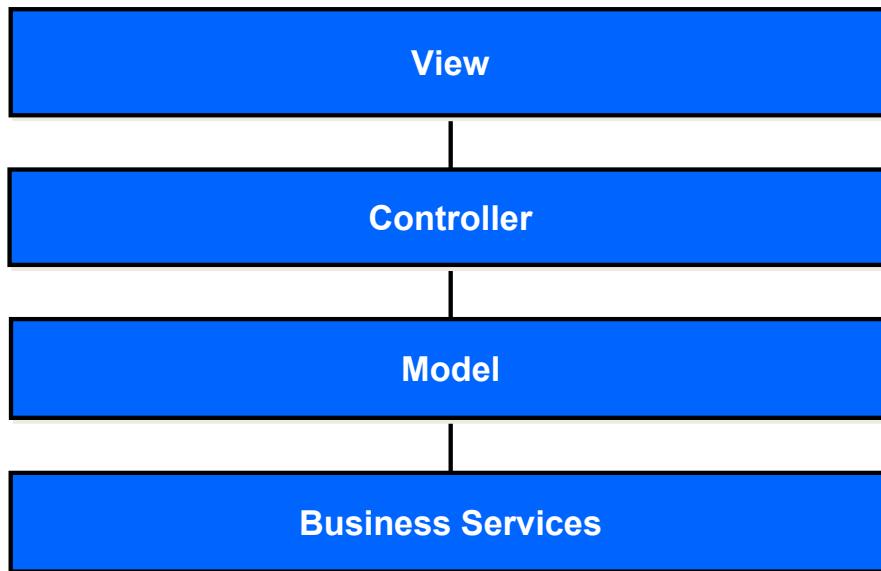
ADF simplifies the tasks of:

- Creating business services
- Designing user interfaces to access those services

Oracle ADF simplifies Java EE development by minimizing the need to write code that implements design patterns and the application infrastructure. These implementations are provided as part of the framework. Oracle ADF also provides a visual and declarative development experience that minimizes the need to write code and reduces the learning curve for 4GL developers.

Business services are implemented as metadata, enabling them to be bound to user interfaces in the same manner regardless of the technology employed in the underlying data model. The use of metadata also enables business rules for databound fields to be specified at the model layer, along with labels, validation, and tool tip properties.

Model-View-Controller Design Pattern



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

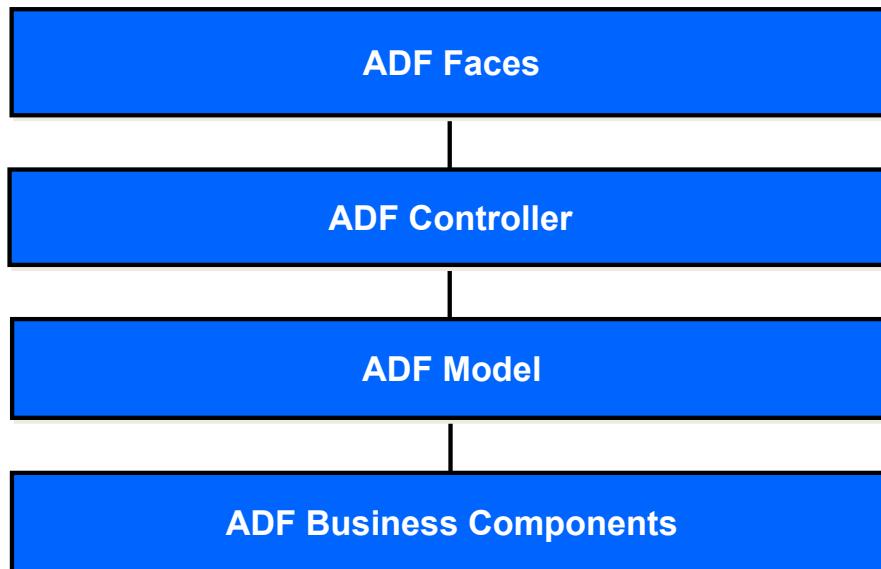
Model-View-Controller Design Pattern

A good practice when developing applications is to employ design patterns. Design patterns are a convenient way of reusing object-oriented concepts between applications and developers. The idea behind design patterns is simple: document and catalog common behavior patterns between objects. Developers can then make use of these patterns rather than re-create them. One of the frequently used design patterns is the Model-View-Controller (MVC) pattern.

In the MVC pattern, the user input, the business logic, and the visual feedback to the user are explicitly separated and handled by three types of objects. Each of these objects is specialized for a particular role in the application:

- The model manages the data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).
- The view manages the presentation of the application output to the user.
- The controller interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.

How the Oracle ADF Framework Implements MVC



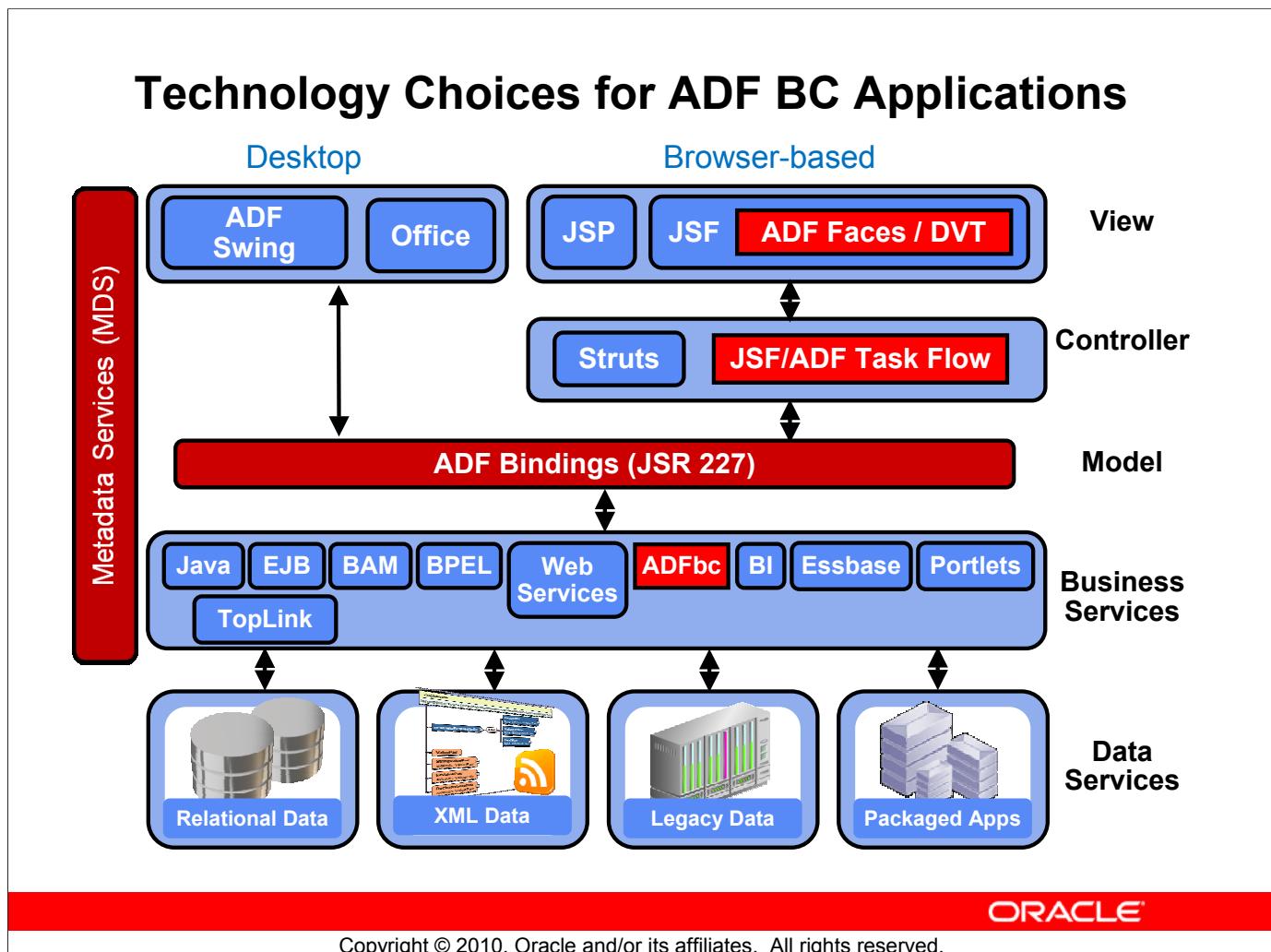
ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

How the Oracle ADF Framework Implements MVC

In addition to design patterns, developers often use architectural frameworks to build applications that perform in a standard way. ADF is such a framework, and it implements the MVC design pattern as follows:

- **Business Services:** Business services provide the back-end data model that interacts with the data source. It may be a set of Java classes or Web services, or can be Enterprise JavaBeans (EJBs), TopLink, or ADF Business Components.
- **Model:** ADF data binding is accomplished in compliance with JSR-227, a data binding and data access facility for Java EE that provides a standard for interactions between UI components and methods available on the business services. With this standard data binding, any Java UI-rendering technology can declaratively bind to any business service.
- **Controller:** Page flow and UI input processing can be implemented in Struts, JavaServer Faces (JSF), or ADF Controller. The ADF Controller enables you to define task flows, encompassing more than just page flow. The ADF Controller is explained in detail in the lesson titled “Planning the User Interface.”
- **View:** The user interface can be a rich client with ADF Swing components, or can use Java ServerFaces and ADF Faces components.

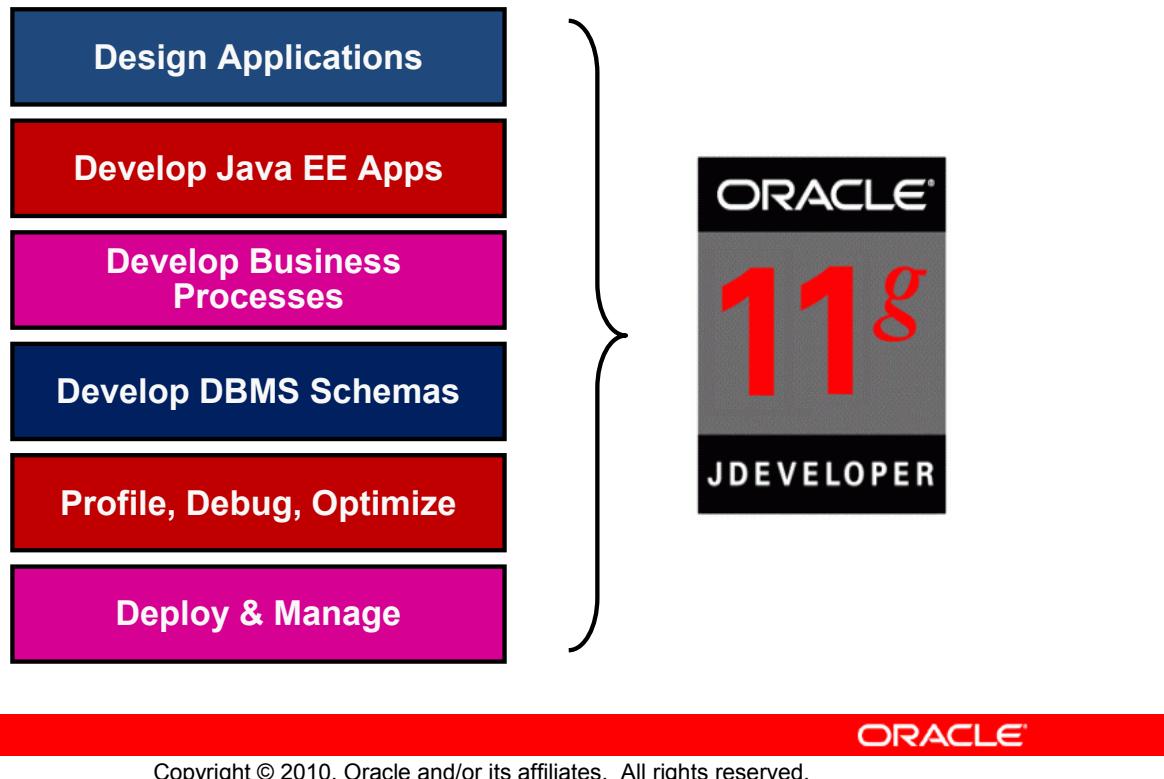


Technology Choices for ADF BC Applications

In this course, you employ the MVC architecture through the use of Oracle ADF and JavaServer Faces:

- **Business Services:** ADF Business Components (ADF BC) provides the business services that are responsible for representing database tables and persisting user input values to the database.
- **Model:** ADF data binding is accomplished in compliance with JSR-227, a standard data binding and data access facility for Java EE that provides a standard for interactions between UI components and methods available on the business services. With this standard data binding, any Java UI–rendering technology can declaratively bind to any business service.
- **Controller:** The ADF Controller is used to manage page flow and UI input processing.
- **View:** JavaServer Faces and ADF Faces provide render kits that enable the UI components defined by JavaServer Faces or ADF Faces to be rendered differently on different devices. For example, the UI can be rendered on a computer’s browser or on a mobile device such as a cell phone or PDA.
- **Metadata Service:** ADF BC applications can also use Metadata Service (MDS) for customization and personalization. This is outside the scope of this course.

Introducing JDeveloper: Oracle's Java and Web Development Tool



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

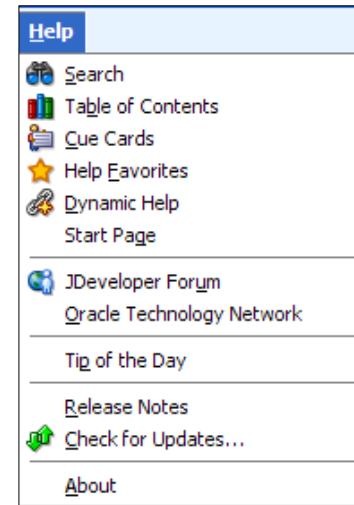
Oracle's Java and Web Development Tool

Oracle JDeveloper 11g is a comprehensive development tool that provides facilities to build Web applications using a common development tool set. This tool has evolved to provide an integrated environment that enables a developer to design, develop, troubleshoot, optimize, and deploy Java EE applications. JDeveloper is the integrated development environment (IDE) that utilizes the Application Development Framework.

Obtaining Additional Information

You can obtain more information about Oracle Fusion, Oracle ADF, and ADF Business Components from the following sources:

- Oracle Technology Network (OTN)
- Forums
- Blogs
- Oracle Magazine
- Developer's Guides
- Online Help
- Oracle Press Publications



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Obtaining Additional Information

Oracle Technology Network (Select Help > Oracle Technology Network.)

- Oracle Fusion Middleware: <http://www.oracle.com/technology/products/middleware/index.html>
- JDeveloper: <http://www.oracle.com/technology/products/jdev/index.html>
- Oracle ADF: <http://www.oracle.com/technology/products/adf/index.html>

Forums (Select Help > Ask for Help.)

- Oracle Application Server: <http://forums.oracle.com/forums/forum.jspa?forumID=44>
- JDeveloper: <http://forums.oracle.com/forums/forum.jspa?forumID=83> (from Online Help, select Ask for Help)

Web Logs (Get it straight from the Oracle experts.): You can search Oracle blogs online: <http://www.oracle.com/technology/products/jdev/howtos/index.html?msgid=6321674>

Oracle Magazine: <http://www.oracle.com/technology/oramag/index.html>

Developer's Guides (Select Help > All Books.)

- Fusion Developer's Guide for ADF
- Web User Interface Developer's Guide for ADF

Online Help: Display table of contents, index, cue cards, or perform full text search.

Oracle Press Publications: *Oracle JDeveloper 11g Handbook: A Guide to Fusion Web Development* and *Oracle Fusion Developer Guide: Building Rich Internet Applications with Oracle ADF Business Components and Oracle ADF Faces* are recent publications.

Summary

In this lesson, you should have learned how to:

- Describe the Fusion architecture
- Explain how ADF fits into the Fusion architecture
- Describe the ADF technology stack (MVC)
- Find more information about ADF



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Getting Started with JDeveloper

2

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- List the benefits that JDeveloper provides for application development
- Use the features of the JDeveloper IDE
- Define the integrated development environment (IDE) preferences
- Create applications, projects, and connections in JDeveloper



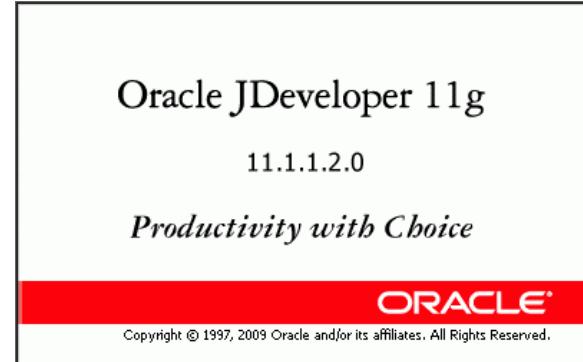
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson introduces you to the JDeveloper IDE. You learn about the features and benefits of JDeveloper and how to use its components in application development. You set preferences for the tool, create an application and project, and create a database connection.

Describing the Benefits of Using JDeveloper

- Standard, open, and extensible
- Improved productivity
 - Visual and declarative
 - Simpler development of Java EE applications
- “Productivity with Choice”
 - Technology stacks
 - Development approaches
 - Development platform
 - Data sources
 - Application servers



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JDeveloper Benefits

Oracle JDeveloper is an integrated development environment (IDE) for building service-oriented applications by using the latest industry standards for Java, XML, Web services, and SQL. It supports the complete development life cycle with features for modeling, coding, debugging, testing, profiling, tuning, and deploying applications.

The goal of JDeveloper is to make enterprise Java development simpler and more accessible, so it focuses on visual and declarative development. Oracle ADF provides JDeveloper with a Java EE framework that implements design patterns and eliminates infrastructure coding.

JDeveloper enables you to use the latest standards to develop applications that can operate across multiple hardware and software platforms. It embraces popular open source frameworks and tools, providing built-in features for Struts, Ant, JUnit, XDoclets, and Concurrent Versions System (CVS), so that you can use these open source tools to streamline the development process. JDeveloper also offers an Extension SDK so that you can add capabilities and customize your development environment.

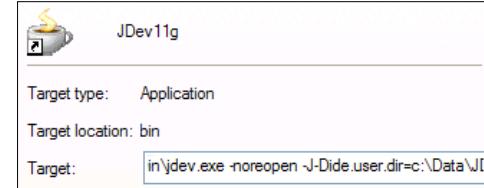
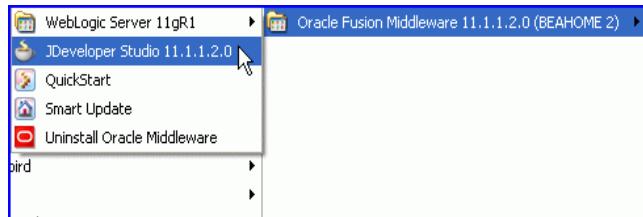
JDeveloper Benefits (continued)

JDeveloper's productive IDE enables developers to choose:

- **Technology stack:** The same productive experience is used for various technology stacks. For example, you can implement a persistence layer by using Java classes, Enterprise JavaBeans (EJB), ADF Business Components, or Web services. JDeveloper provides a declarative way to implement the chosen technology, as well as drag-and-drop mechanisms to bind UI components to any of these implementations.
- **Development approach:** JDeveloper provides a choice of development approaches that includes Model Driven Architecture (MDA), declarative development, and hand-coding. With MDA, you start by building a model of the application, which can then be implemented in different technologies. Declarative development enables you to specify requirements of the application declaratively, thus minimizing the coding that may be required. However, hand-coding is still preferred by some. Developers can choose the approach that best suits their skill levels and preferences. Various developers can work on the same files using any of these approaches.
- **Development platform:** JDeveloper is a 100% Java-based tool. Because of this, it is a cross-platform IDE that runs on Windows, Linux, Mac, and various UNIX-based systems.
- **Data source:** JDeveloper can access any JDBC-compliant database.
- **Application server:** Applications built with JDeveloper can be deployed to any Java EE-compliant application server.

Launching JDeveloper

- You can launch JDeveloper by:
 - Selecting Start > Programs > Oracle Fusion Middleware 11.1.1.2.0 > JDeveloper Studio 11.1.1.2.0
 - Starting from the command line or a shortcut pointing to `jdev.exe` in the `<JDev_Home>\jdev\bin` directory
- You can specify flags:
 - For the user directory:
`-J-Dide.user.dir=<path>`
 - For editor windows:
`-noreopen`



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Launching JDeveloper

You can launch JDeveloper in one of the following ways:

- By selecting Start > Programs > Oracle Fusion Middleware 11.1.1.2.0 > JDeveloper Studio 11.1.1.2.0 (or a different location as specified in the Installer)
- By double-clicking the JDeveloper executable in the home directory of the JDeveloper installation
- From the command line or a shortcut with the following command:
`<JDev_Home>\jdev\bin\jdev.exe`

By default, JDeveloper's user directory is the `\system` subdirectory of the `<JDev_Home>` directory, such as `C:\Oracle\Middleware\jdeveloper\system`—that is, where it stores user preferences, such as information about the appearance of the IDE. If you want to use a different directory, start JDeveloper with the `-J-Dide.user.dir` flag. The following example sets the user directory to a directory where the user customarily stores data:

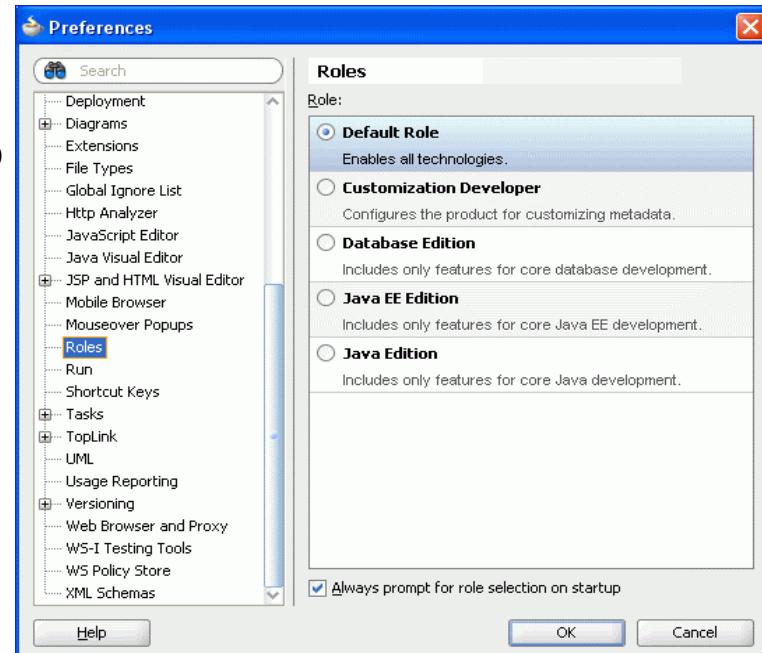
`c:\JDev\jdev\bin\jdev.exe -J-Dide.user.dir=c:\Data\JDev.`

Note: As an alternative to using this flag, you can set the `JDEV_USER_DIR` environment variable in your operating system.

Another flag that you can use when launching JDeveloper is `-noreopen`, which prevents JDeveloper from opening all the editor windows that were open the last time you closed JDeveloper. This can improve JDeveloper's startup time.

Defining Roles

When you launch JDeveloper, you need to select a developer role.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Defining a Role

To begin working in JDeveloper, you need to select a role to use. You then define an application with one or more projects. If your application requires data from a database, you also need to create a reusable database connection.

Roles and Shaping: When you start JDeveloper for the first time, you are asked to choose the role that you as a developer want to use. You can change that role later, if desired, by selecting Tools > Preferences from the JDeveloper menu.

The JDeveloper environment customizes itself based on the role of the user by removing items that are not needed from the menus, preferences, new gallery, and even individual fields in dialog boxes. This is called shaping, and it can even control the default values for fields in dialog boxes within the tool.

Using JDeveloper Features

JDeveloper features that facilitate ease of application development include:

-  Application Navigator
-  Database Navigator
-  Editor (code or visual editors)
-  Component Palette
-  Resource Palette (IDE and Application)
-  Data Controls panel
-  Structure Window
-  Property Inspector
-  Log Window



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using JDeveloper Features

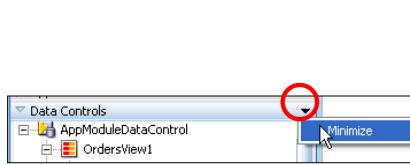
The next few slides provide an overview of using the features that are listed in this slide.

Using JDeveloper's Application Navigator

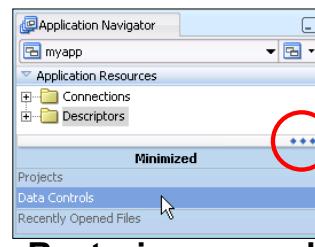


Has expandable, minimizable panels to display:

- Projects
- Application resources
- Data controls
- Recently opened files

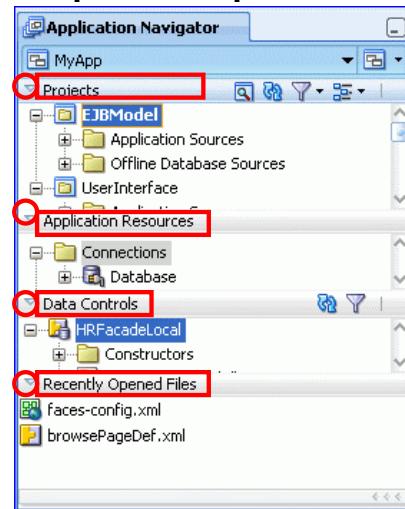


Minimizing a panel



Restoring a panel

Expandable panels



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Application Navigator

The Application Navigator presents you with a hierarchical view of all the elements in your application. At the top of the Application Navigator is a drop-down list of applications that also enables you to open an application or create a new one. To the right of the drop-down list is an arrow that displays the application menu, enabling you to select actions pertaining to the current application.

The Application Navigator contains four panels:

- **Projects:** Shows all the projects that are part of the current application, along with a hierarchical view of the project elements. The project panel contains a toolbar that enables you to display project properties, refresh, filter, or rearrange the display.
- **Application Resources:** Shows application-level resources such as descriptors and connections
- **Data Controls:** Provides a hierarchical view of all the data elements, such as view objects and methods, that are available to access in the user interface. It enables you to use the drag-and-drop functionality to create UI components on a page. It also contains a toolbar that allows you to refresh and filter the display.
- **Recently Opened Files:** Provides quick access to files in the current application that have recently been used

Using the Application Navigator (continued)

You can expand, collapse, minimize, and restore each of these four panels in the Application Navigator:

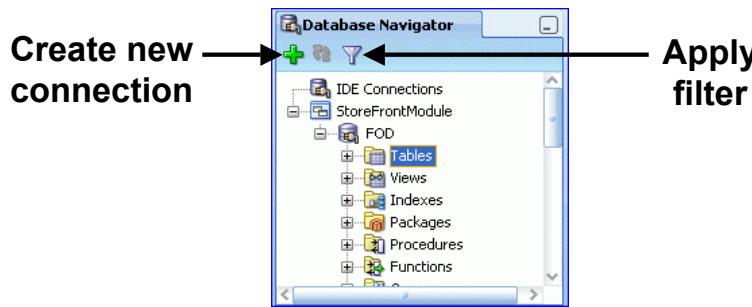
- To expand or collapse, click the arrow to the left of the panel name.
- To minimize, select Minimize from the drop-down menu at the right of the panel's title bar.
- To restore, click the ellipsis at the bottom right of the Application Navigator and select the minimized panel that you want to restore.

Using JDeveloper's Database Navigator



The Database Navigator:

- Shows both IDE and application connections
- Enables you to:
 - View database elements
 - Filter the display
 - Create new connections
 - Delete connections
 - Change connection properties
 - Open SQL Worksheet



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Database Navigator

The Database Navigator provides you with a complete editing environment for online databases. You can use it to create, update, and delete database objects. Much of the functionality available standalone in SQL Developer is also available in JDeveloper. Highlights include:

- SQL Worksheet: Script execution, explain plan, autotrace, code snippets (drag and drop), DBMS, and OWA output
- Creation of external, index organized, temporary, partitioned (range, hash, and list) tables and materialized views
- Extensive context menu options to modify objects (for example, table rename, column additions, compilation, index rebuilds, and database link testing)
- Browse, query, update, delete, sort, and filter data including CLOB and BLOB and tracking of changes through message log
- Export data in multiple formats, export DDL, import data
- Database reports

Using JDeveloper's Editors

Edit code:

Edit properties:

Visually edit task flows, diagrams, and pages:

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Using the Editors

JDeveloper provides a variety of editors with which you can:

- **Edit code:** To edit source code, double-click the file in the Application Navigator to display the contents in the appropriate editor. Highlights of the code editor include the following:
 - **Toolbar:** Enables quick access to functions such as generate accessors, surround with, override method/implementation interface, reformat, and so on
 - **Auto Code Highlight:** Automatically highlights instances of a selected item
 - **Syntax color options:** Makes code easily readable
 - Quick Javadoc
 - **Code folding:** Enables you to expand and contract sections of code
 - Show Whitespace Characters: Renders spaces, new lines, carriage returns, nonbreaking spaces, and tab characters as alternate visible characters
 - **Overview Popup:** Enables you to view the source at an overview mark by simply hovering the cursor over the mark
 - **Structure Window Popup:** View the source for a method by pressing Ctrl while the cursor is over the desired method in the Structure window.

Using the Editors (continued)

- **Quick Outline:** Provides a new method to quickly navigate to methods and fields of a class and its superclasses. The “ghost” window floats just above the code and contains a tree of the available methods and fields of the current class and its superclasses.

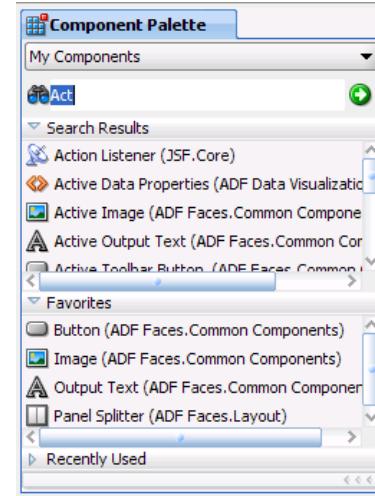
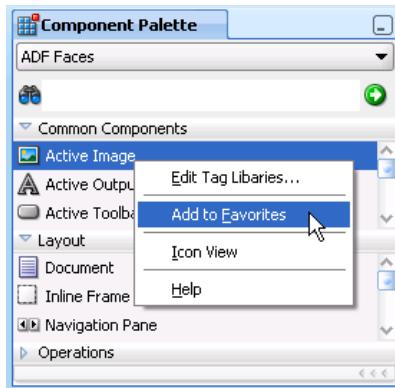
Start typing in a filter field to reduce the visible items, allowing quick and easy selection for navigation to the desired place.
- **Code Peek:** Provides the ability to view code in different files without navigating or opening new editors
- **Last Change Indicator:** Provides a marker in the left gutter of code editors provides a quick visual indicator of changes to a file since the last save
- **Edit properties:** Many editors for components are modeless editors that appear in the editor area along with the other editors. They can remain open to enable easy access to component editors while you work on other parts of the application.
- **Edit Visually**
 - **Page Flows:** Toolbar enables quick access for modifying the display and marking default activities and exception handlers.
 - **Diagrammers:** The internal diagramming framework used to build the UML tools in JDeveloper, the Page Flow diagrams, and other visual editors has been re-architected to be more flexible for internal consumers. The result for end users is more consistent behavior between the different diagrams.
 - **Pages:** The breadcrumb bar in the page editor shows the hierarchy of nodes from the current caret position up to the top of the file. Placing the cursor over a node displays some information about the node and clicking the node navigates the caret to the node location.

Using JDeveloper's Component Palette



The Component Palette enables you to:

- Display the components available to drag to visual editor
- Search for components
- Display the favorites and recently used components



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Component Palette

When you are using any type of visual editor, the Component Palette displays the components that are available to drag to the editor.

The Component Palette features are as follows:

- A drop-down list at the top of the palette enables you to select the component category to display.
- Collapsible panels and divider sections can be used to organize related components.
- A quick search field helps you to locate components.

You can add commonly used components to your Favorites list for easier access later, and another panel keeps track of your recently used components.

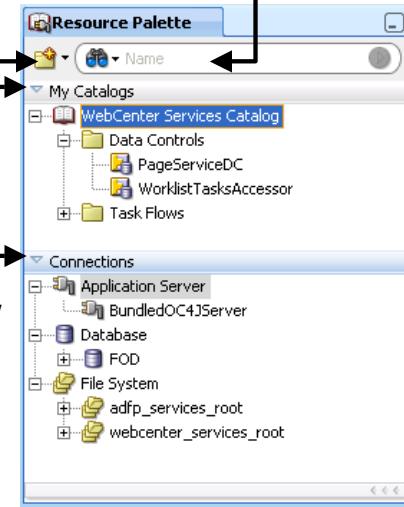
Using JDeveloper's Resource Palette



Toolbar: Create/Import or Search

The Resource Palette:

- Contains two types of resources:
 - Catalogs
 - Connections
- Provides single searchable view of and access to many types of resources
- Facilitates sharing resources among developers



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Resource Palette

The Resource Palette provides a federated view of the contents of one or more otherwise unrelated repositories in a unified search-and-browse user interface. This enables users to find the resources needed for the task at hand easily. Users can locate resources from a wide variety of repositories, search for resources and save searches, preview a resource before using it, and reuse resources by sharing catalog definitions.

The Resource Palette contains two expandable and resizable panels:

- **My Catalogs:** Resources are created and published in their source repository. Related resources are grouped into catalogs and are then exposed to the user in the Resource Palette.
- **Connections:** Many types of connections, such as to databases, file systems, or application servers, can be created and displayed in the Resource Palette for use in any application.

By using the Resource Palette's toolbar icon, you can create new or import existing catalogs and connections. You are able to search through all resources.

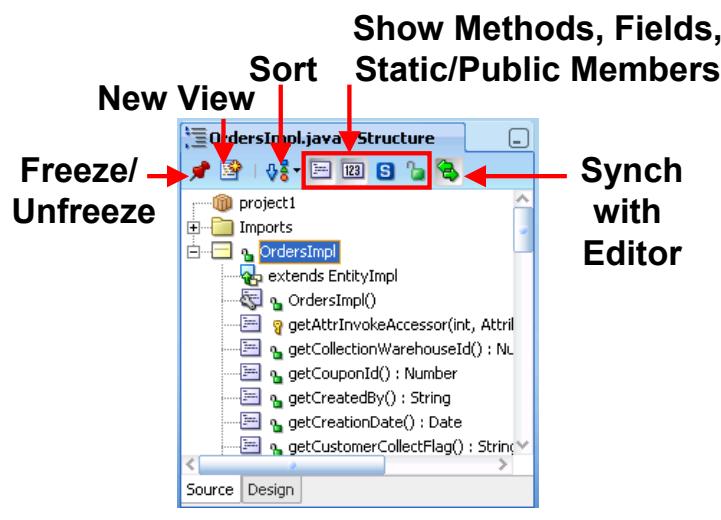
By default, the Resource Palette is displayed on the right of the JDeveloper IDE. If it does not appear, you can display it by selecting View > Resource Palette.

Using JDeveloper's Structure Window



The Structure Window provides:

- A tree view of the selected document or object
- The ability to:
 - View in different ways
 - Navigate



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Using the Structure Window

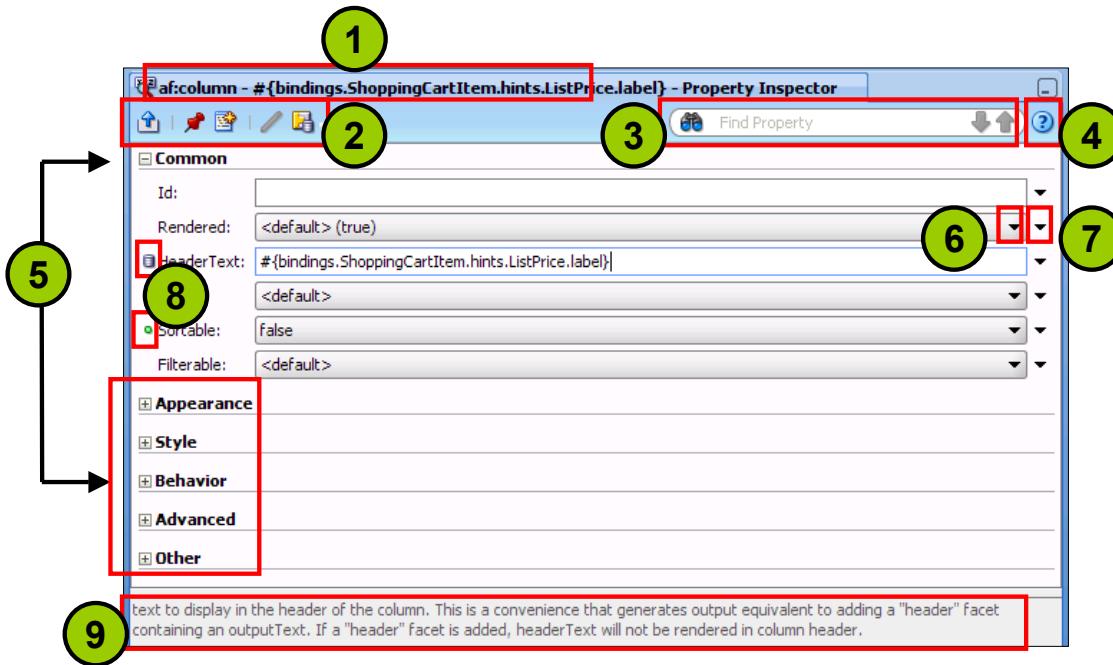
The Structure window is displayed by default below the Application Navigator. If it is not visible, select View > Structure to display it.

When you select anything in a navigator, editor, or the Property Inspector, the Structure window displays the elements of the selected document in a tree format. For example, when you select a .java source file, the classes, interfaces, methods, and variables are displayed, and you can sort and view them in a variety of ways. You can use the Structure window to quickly locate and navigate to specific areas of code in your Java source files and to browse the class hierarchy.

When you are designing a user interface with the visual editor, the Structure window displays the components of the UI and their associated event-handling methods in a hierarchical tree format. The Structure window enables you to create new components in precisely the desired location, whereas such precise placement is often difficult in the visual editor.

You can open as many new instances of the Structure window as you would like. New instances will automatically track the active selection in the active view unless frozen. You can freeze and unfreeze the selection in the window by clicking Freeze View. You can click the New View icon to open a new instance of the Structure window that appears docked with the existing window or windows.

Using JDeveloper's Property Inspector



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Property Inspector

The Property Inspector presents a collection of all settable properties for a selected element.

Features of the Property Inspector include the following:

1. When you select an element in the Structure window or Visual Editor, the Property Inspector title bar shows the name of the selected element.
2. A toolbar enables you to perform component-level actions, such as rebinding to a different control, freezing/unfreezing, or creating a new view. You can also toggle auto-extend, which enlarges the Inspector when the cursor moves over it.
3. A search function enables you to quickly locate a property on any panel.
4. A Component Help button invokes context-sensitive Help.
5. Expanding a category group on the left displays a panel containing a subset of properties.
6. Drop-down lists of values are available for many properties.
7. If an editor is available for a property, an arrow to the right of the property invokes it.
8. Icons indicate whether property values are databound or whether they are changed from the default.
9. When a property is selected, a brief description of the property is displayed at the bottom of the Inspector.

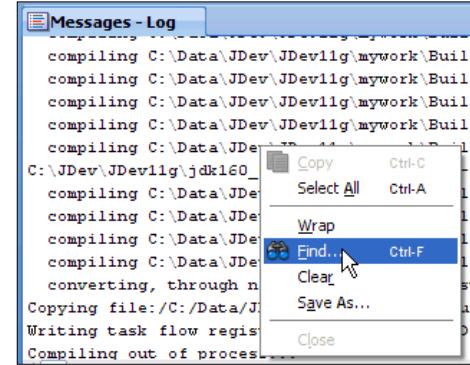
When you change a property, the value is automatically validated and applied when you navigate out of that field.

Using JDeveloper's Log Window



The Log window:

- Displays JDeveloper messages, such as logging and compilation messages
- Has one or more tabs: a Messages tab and additional tabs depending on what is occurring
- Displays standard output (such as `System.out.println()` messages)
- Has context menu for clearing, wrapping, saving output, searching, and so on



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Log Window

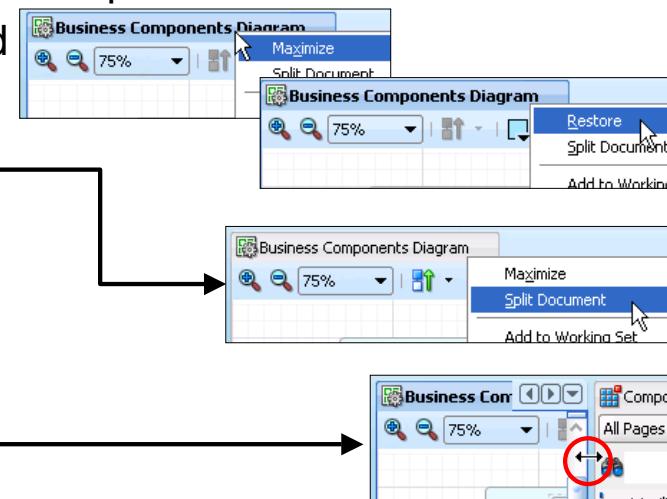
The Log window in JDeveloper enables you to view output from JDeveloper operations such as running, debugging, and profiling. The Log window consists of a Messages tab and optionally, additional tabs for displaying results.

The context menu of a log window enables you to select, copy, or save text, clear the text, wrap the entries, find a specified string in the text, or close a tab. There are other context menu options depending on the type of log window displayed on a tab. You can also close individual tabs by clicking the X that appears on the tab when you move the cursor over it, and you can close all log windows by clicking the X on the top tab. If you close all log windows, you can reopen them by selecting View > Log from the main menu.

Working with JDeveloper Windows

Many of the windows in JDeveloper can be:

- Maximized or restored
- Opened or closed
- Split or unsplit
- Repositioned
- Docked or undocked
- Resized



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Working with JDeveloper Windows

By now you can see that JDeveloper has many types of windows that display aspects of the IDE. Most JDeveloper windows can be maximized or restored by using the context menu that appears when you right-click its tab. Alternatively, you can double-click the tab to maximize or restore the windows.

Other options available from the context menu are splitting or unsplitting the window or closing the window. Another way to close the window is by clicking the X on its tab. To reopen, either double-click the corresponding node in the Application Navigator, or select the window to open from the View menu.

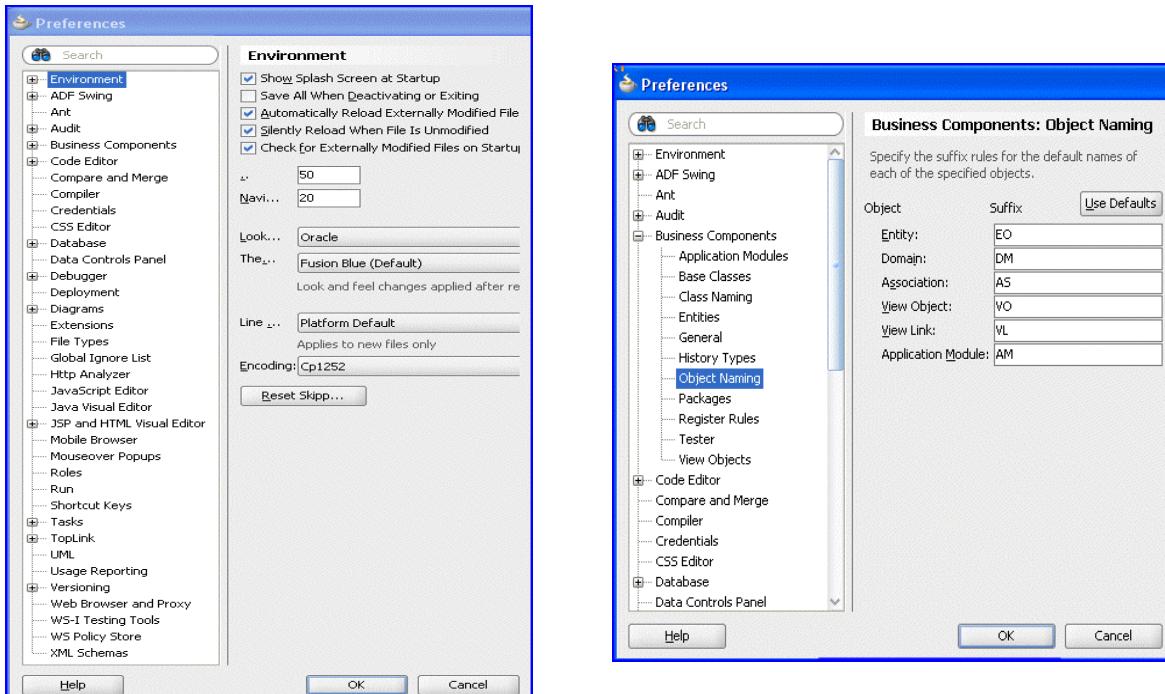
You can reposition the window by dragging its tab to a different part of the IDE. You can also undock and redock floating windows by dragging the tab.

You can use the mouse to resize all JDeveloper windows.

You restore the default arrangement of the IDE windows by selecting Window > Reset Windows to Factory Settings.

Setting IDE Preferences

Select Tools > Preferences to set preferences:



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Setting IDE Preferences

There are many preferences that you can set to affect the display and functionality of the IDE. You access the Preferences window by selecting Tools > Preferences.

For example, to change the default naming scheme for ADF Business Components objects, you expand the Business Components node in the Preferences window and select Object Naming. You can specify a suffix for each type of object.

Getting Started in JDeveloper

To build an application in JDeveloper, you need to define:

- An application 
- One or more projects 

If your application requires data from a database, you also need to define a database connection. 

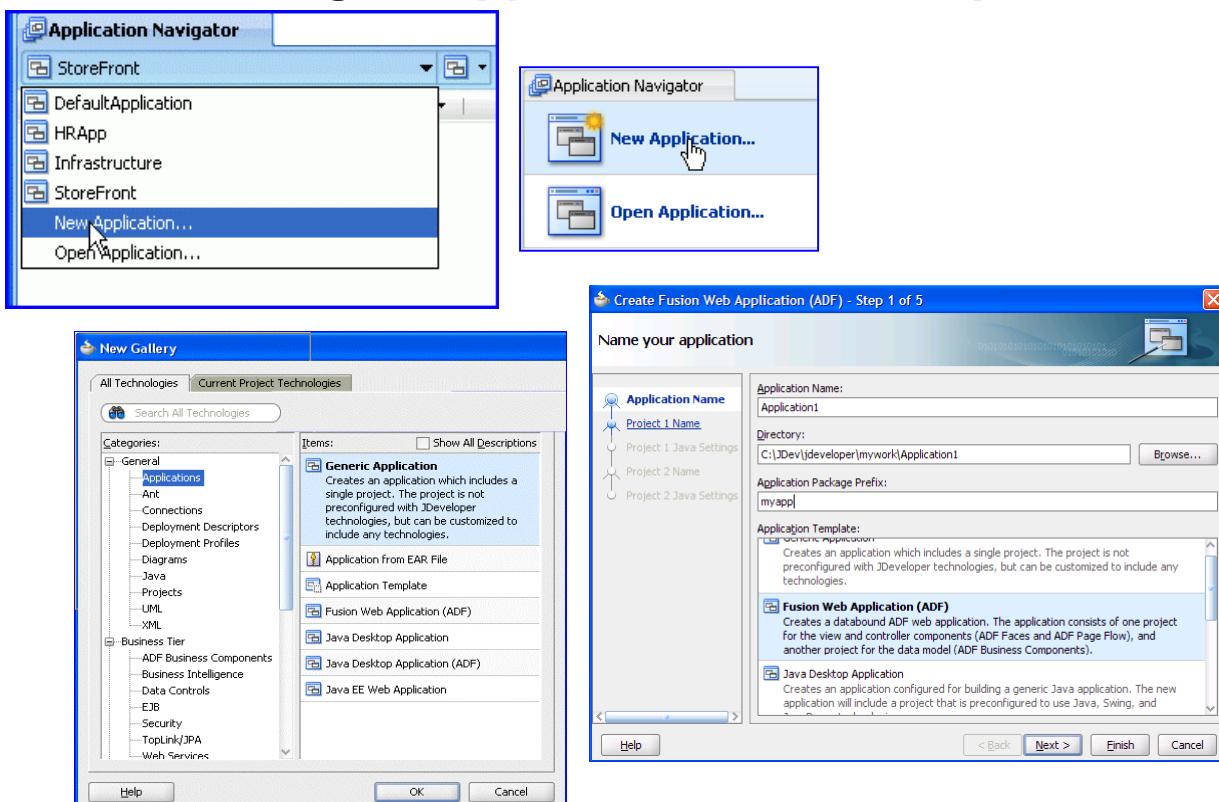


Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Getting Started in JDeveloper

The following slides explain how to create an application, a project, and a database connection.

Creating an Application in JDeveloper



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating an Application

Oracle JDeveloper uses a well-defined structure to manage Java programming applications. The structure is hierarchical and supports applications, projects, images, .html files, and so on.

The application is the highest level in the control structure. It is a view of all the objects you currently need while you are working. An application keeps track of the projects you use and the environment settings while you are developing your Java program.

You create an application by invoking the Create Application dialog box in one of the following ways:

- If there is an application already open in JDeveloper, invoke the application drop-down list and select New Application. If there is no open application, click New Application.
- Select File > New, and in the New Gallery dialog box, select General > Applications. Then select either Generic Application, or a specific type of application from the list. Click OK.

Enter values for the following properties in the Create Application Wizard:

- **Application Name:** Enter a name for the application. This name is used to categorize all the files in your application.
- **Directory Name:** Enter a top-level directory for the application or click Browse to locate one. This is where all your application files are stored during development.
- **Application Package Prefix:** Enter a prefix for all packages associated with this

Creating an Application (continued)

- **Application Package Prefix:** Enter a prefix for all packages associated with this application. The prefix you assign defines the root package for every project in the application.
- **Application Template:** Select a template for the application. Click Manage Templates to edit an existing template or to create a new one. An application template provides a way to apply best practices to create the project structure for standard applications with the appropriate combination of technologies already specified. The new application that is created from a template appears in the navigator already partitioned into projects, with technology scopes set.

Saving Applications

Whenever you save an application, you are prompted to save all the current open files. To save the open and modified files, select the Save option (or the Save All option) from the File menu. Italic style is used in the Application Navigator to indicate files that have not yet been saved.

Applications are stored in files with the extension .jws. You do not edit an application file directly, but you can view the content of an application file by using any text editor.

When you open JDeveloper, the last application used is opened by default so that you can resume your work.

Using the Application Overview Page

The Application Overview page shows details about the application's:

- Java files
- Page flows
- Web pages
- ADF Business Components
- ADF binding files
- Offline Databases

It enables you to:

- Create new objects
- Delete objects
- Edit objects



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Application Overview Page

When you first create an application, the Application Overview page appears in the editor area of the IDE. This page provides a listing of all the sources for your application, grouped into helpful categories that are displayed in resizable panels.

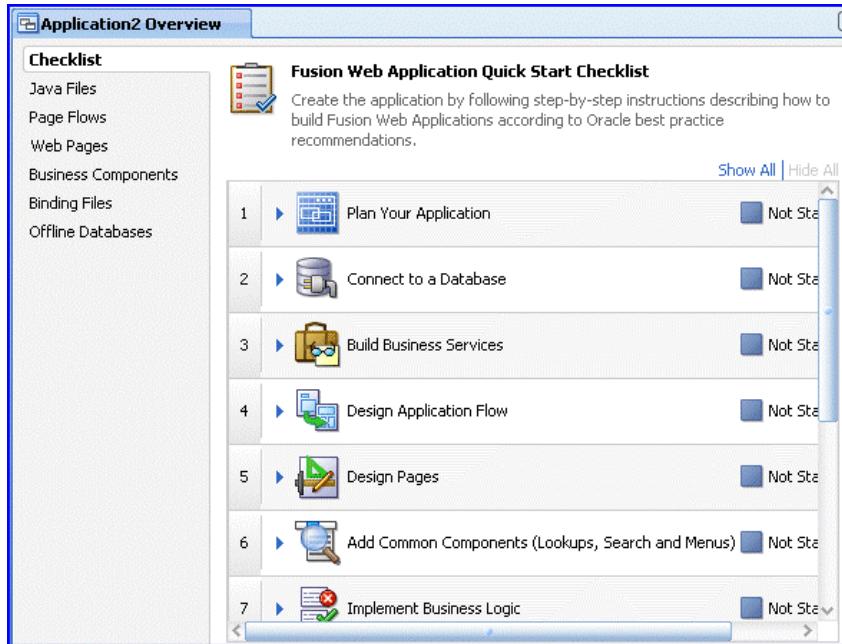
The Application Overview page gives you a sense of the overall status of your application, such as which sources are incomplete, which have errors, and so on.

When you select one of the panels, you have access to a context menu that enables you to create objects. For example, in the Java Files panel, the New menu enables you to create a Java class or interface. You can also edit and delete objects here.

In the Application Navigator, select Show Overview from the application menu to display the Application Overview page.

The Application Overview Checklist

The Application Overview Checklist is designed to guide users through typical steps in building an application.



ORACLE

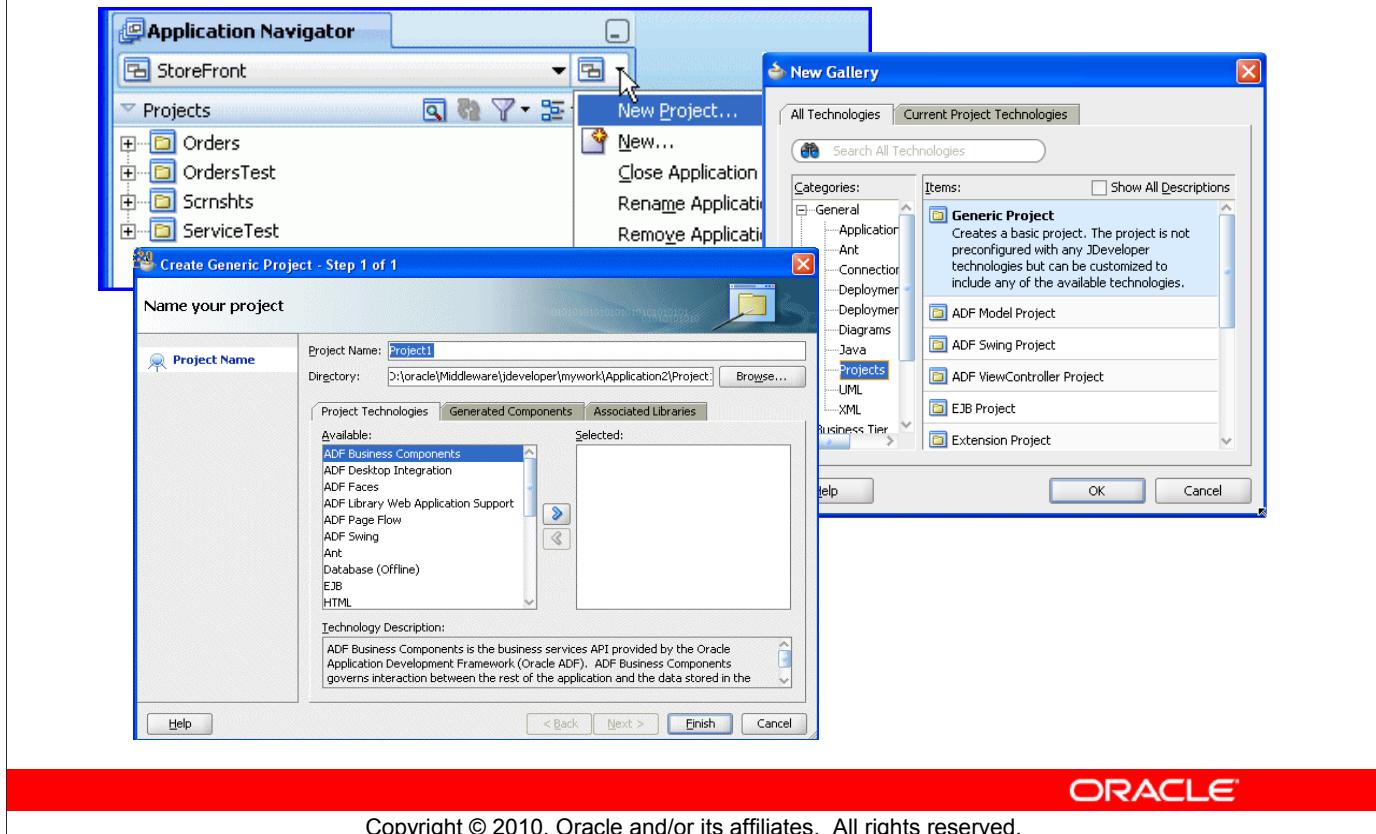
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

The Application Overview Checklist

The Application Overview Checklist is designed to help users (especially new users) by providing step-by-step instructions describing how to build the type of application they have chosen, according to Oracle best practice recommendations.

Initially, (in JDeveloper 11.1.1.2.0), there is a checklist only for new applications based on the “Fusion Web Application” application template.

Creating a Project in JDeveloper



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating a Project in JDeveloper

JDeveloper projects organize the file elements that are used to create your program. In the Applications Navigator, projects are displayed as the second level in the hierarchy under the application.

Projects manage environment variables, such as the source and output paths used for compiling and running your program. Projects also maintain compiler, run-time, and debugging options so that you can customize the behavior of those tools for each project.

A project file has the file extension `.jpr` and keeps track of the source files, packages, classes, images, and other elements that may be needed for your program. You can add multiple projects to your application to easily access, modify, and reuse your source code. You can view the content of a project file by using any text editor.

There are two ways to create a project in JDeveloper. The first way is when you create an application. Depending on the application template that you select, one or more projects are created automatically. Sometimes you are presented with a wizard page enabling you to name the project and set its technologies.

Creating a Project in JDeveloper (continued)

The second method of creating a project is to create a new one. You must first select an existing application to contain the project. Then you can create a new project from the New Gallery in one of the following ways:

- Invoke the application menu that appears to the right of the application name in the Applications Navigator, or click the down arrow at the far right of the application name, and select New Project to invoke the New Gallery.
- Select File > New from the menu.
- Click New on the Toolbar.

In the New Gallery, select the type of project to create, and then click OK.

When you are presented with the Create Project Wizard, you can name the project and specify where you want to store the project files. Depending on the type of project, you may also need to select the technologies to use, set Java or EJB settings, or define other types of project properties.

Note: You can automatically save and compile a project before running it. This preference is set in project properties. You can set default properties for new projects as follows:

1. Select Application > Default Project Properties from the JDeveloper menu.
2. In the tree at the left of the Default Project Properties dialog box, select Run/Debug/Profile.
3. With the Default run configuration selected, click Edit.
4. In the tree at the left of the Edit Run Configuration dialog box, select Tool Settings.
5. In the Before Running section, select the options that you want to perform before running, such as Make Project and Save All.
6. Click OK twice to close both dialog boxes and save your preferences.

You can edit preferences for existing projects by right-clicking the project in the Application Navigator and selecting Project Properties.

Creating Database Connections



Two types:

- IDE:
 - Available to be added to any application
 - Create in Database Navigator or Resource Palette
 - To add to an application, drag to Application Resources or to an application in Database Navigator
- Application:
 - Owned by a specific application
 - Create in Database Navigator or Application Resources

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

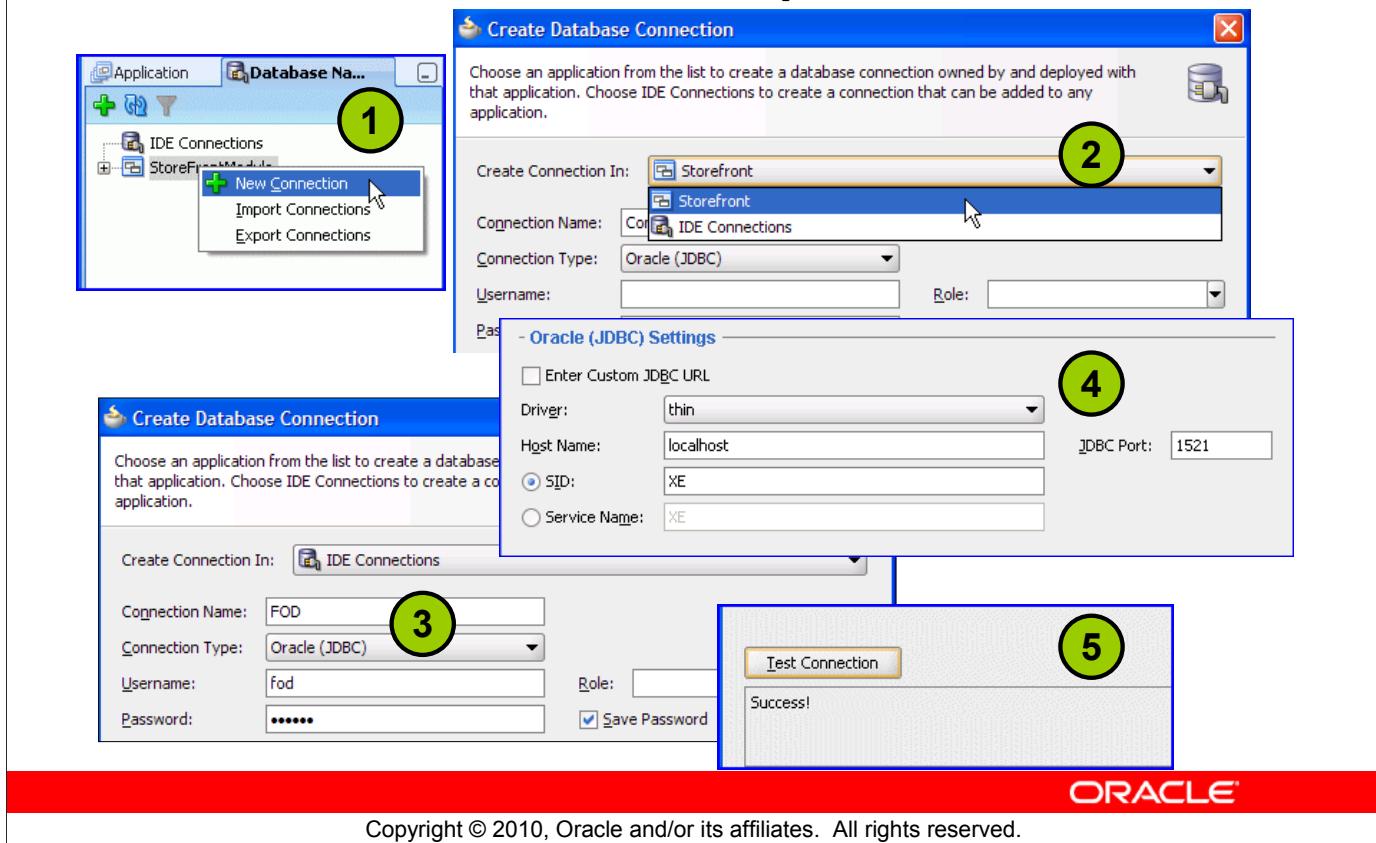
Creating Database Connections

JDeveloper enables you to define several types of connections, including connections to external data sources. Database connections are visible in the Database Navigator, and are organized by the application that owns them. They can also be added to a Resource Catalog, enabling other developers to share them.

All IDE connections are stored in the `connections.xml` file in the `system11.1.1.0.xx.xx\xo.jdevimpl.rescat2\connections` subdirectory of the JDeveloper user directory (see “Launching JDeveloper” earlier in this lesson for the location of the user directory).

The file system location for an application’s connection descriptor definition information is the `connections.xml` file in the `.adf\META-INF` subdirectory of the application itself. By default, applications are stored in the `\mywork\<Application Name>` directory under the root directory in which JDeveloper is installed, although you can choose to save them elsewhere.

Creating a Database Connection in JDeveloper



Creating a Database Connection in JDeveloper

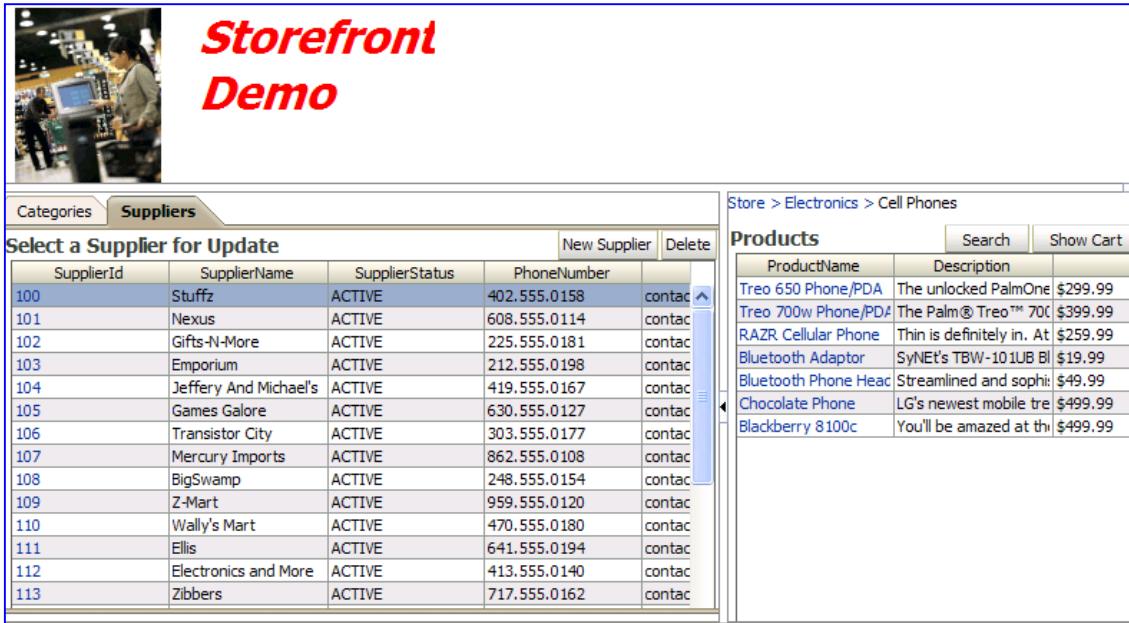
In JDeveloper's Database Navigator, you can set up and manage database connections to enable your projects to communicate with external data sources, including the Oracle database. All defined connections are accessible to any application or project. To define a new database connection, perform the following steps:

1. In the Database Navigator, click the New Connection icon at the top left of the navigator to invoke the Create Database Connection Wizard, or right-click an application and select New Connection from the context menu.
2. In the Create Connection In field, select IDE Connections if you want a connection that can be added to any application, or select an application to add the connection for that application only.
3. Provide a name and connection type, and enter a username, a password, and, optionally, a role. If you want to save the password, select the Save Password check box. This option is selected by default.
4. In the Settings section (determined by your Connection Type choice), enter the connection details as requested.

Creating a Database Connection in JDeveloper (continued)

5. Click Test Connection. JDeveloper checks the connection by using the information you provided. If the test succeeds, a success message appears in the status text area. If the test does not succeed, an error appears. Click Back and change any previously entered information as needed to correct the error, or check the error content to determine other possible sources of the error. When the test succeeds, click Finish. The new connection name appears under the owning application node (or the IDE Connections node) in the Database Navigator.

Describing the Course Application



Storefront Demo

Categories Suppliers

Select a Supplier for Update

SupplierId	SupplierName	SupplierStatus	PhoneNumber	Action
100	Stuffz	ACTIVE	402.555.0158	Contact
101	Nexus	ACTIVE	608.555.0114	Contact
102	Gifts-N-More	ACTIVE	225.555.0181	Contact
103	Emporium	ACTIVE	212.555.0198	Contact
104	Jeffery And Michael's	ACTIVE	419.555.0167	Contact
105	Games Galore	ACTIVE	630.555.0127	Contact
106	Transistor City	ACTIVE	303.555.0177	Contact
107	Mercury Imports	ACTIVE	862.555.0108	Contact
108	BigSwamp	ACTIVE	248.555.0154	Contact
109	Z-Mart	ACTIVE	959.555.0120	Contact
110	Wally's Mart	ACTIVE	470.555.0180	Contact
111	Ellis	ACTIVE	641.555.0194	Contact
112	Electronics and More	ACTIVE	413.555.0140	Contact
113	Zibbers	ACTIVE	717.555.0162	Contact

Products

ProductName	Description	Price
Treo 650 Phone/PDA	The unlocked PalmOne	\$299.99
Treo 700w Phone/PDA	The Palm® Treo™ 700	\$399.99
RAZR Cellular Phone	Thin is definitely in. At	\$259.99
Bluetooth Adaptor	SyNET's TBW-101UB Bl	\$19.99
Bluetooth Phone Head	Streamlined and sophi	\$49.99
Chocolate Phone	LG's newest mobile tre	\$499.99
Blackberry 8100c	You'll be amazed at th	\$499.99

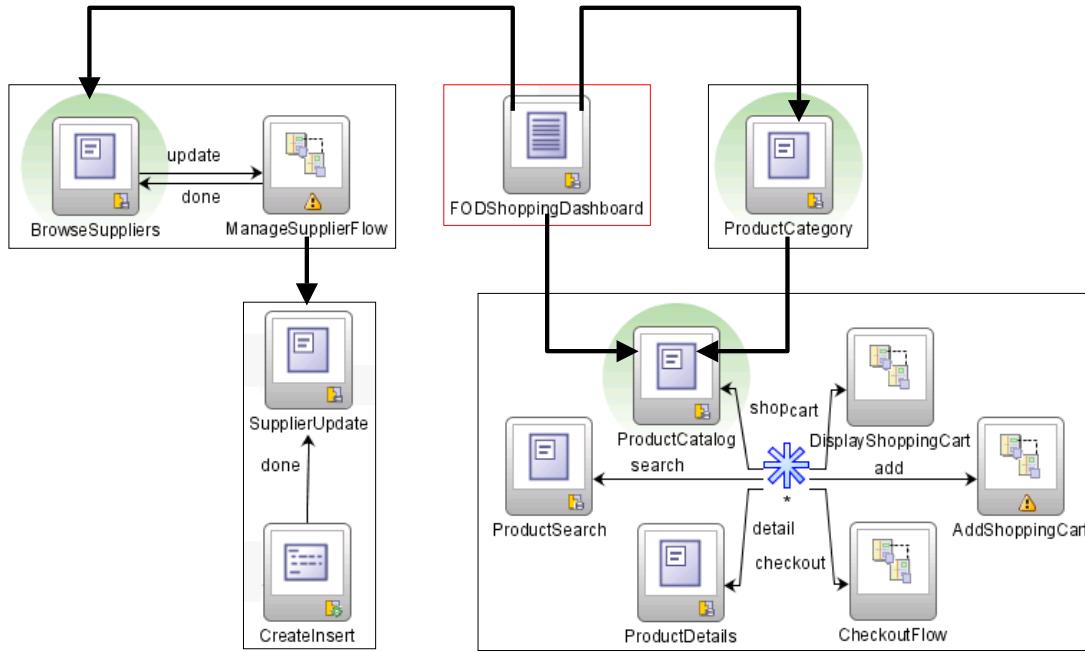
ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Describing the Course Application

In this course, you build an application that serves as a storefront where customers can select products, add to their shopping carts, and check out. They can also view suppliers, and those users with appropriate permissions can also create, update, and delete supplier records.

Presenting the Storefront User Interface



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Presenting the Storefront User Interface

The storefront user interface enables a customer to browse a selection of products and to place an order. It consists of the following high-level pages and task flows:

- **Shopping Dashboard:** The main page that contains regions to display all other pages of the application. The user has two main options here, as shown on two tabs:
 - **Shopping:** The shopping area enables users to browse or search for products, add products to their shopping carts, and check out the order.
 - **Suppliers:** The suppliers area enables users to view suppliers, and also to create, update, and delete suppliers if the user is authenticated and has permission.
- **Browse suppliers:** A list of suppliers; users with appropriate permissions can click a supplier link to update or delete, or can click a button to create a new supplier
- **Manage suppliers:** A page where users can update or create new suppliers, depending on parameters passed
- **Product search:** A search form for products
- **Product category:** A tree of categories and subcategories
- **Product catalog:** A series of tables containing categories, subcategories, and products; users can select a product to display details
- **Product details:** Details about a single product; users can add the product to the cart

There is a series of checkout pages where users confirm their information and submit the order.

Summary

In this lesson, you should have learned how to:

- List the benefits that JDeveloper provides for application development
- Use the features of the JDeveloper IDE
- Define IDE preferences
- Create applications, projects, and connections in JDeveloper



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 2 Overview: Using JDeveloper

This practice covers the following topics:

- Setting IDE Preferences
- Creating a JDeveloper Application and Project
- Initializing the Project and Creating a Database Connection
- Examining the Course Application



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 2 Overview: Using JDeveloper

In the practices for this lesson, you start JDeveloper, set preferences for the IDE, and create a JDeveloper application, project, and database connection. You then view the course application in a browser and identify the functionality of the pages.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Osi S.R.L. use only

Building a Business Model with ADF Business Components



ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Describe the role of ADF Business Components (ADF BC) in building a business service
- Explain the architecture of ADF BC
- Identify the types of components that cooperate to support the business service implementation
- Use JDeveloper design-time facilities for ADF BC
- Explain how ADF BC components are used in a Web application



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson provides students with an overview of ADF Business Components and how they are used as business services to support a Web application. The lesson introduces many topics for which additional details are provided in later lessons. Students create a simple master-detail data model.

Describing ADF Business Components (ADF BC)

ADF Business Components (ADF BC) characteristics:

- Provides data interaction and business logic execution
- Maps to a data source, such as an Oracle database
- Enables 4GL development
 - Wizard-based or visual development
 - Implemented in metadata, not code
- Enables business logic development
 - Predefined Java methods for any event
 - Declarative business rules



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ADF Business Components

Oracle ADF Business Components (ADF BC) provides building blocks that help you to create the business services part of your application. That is, it governs interaction between the rest of the application and the data stored in the data source, providing validation, specific services, and other business logic.

With ADF BC, you can:

- Write and enforce business application logic in a centralized way
- Author and test business logic in components that automatically integrate with databases
- Reuse business logic in multiple applications and application tasks
- Access updatable views of business data that are customized to specific tasks
- Access and update the views from browser, desktop, mobile, and Web service clients
- Maintain and modify the business functionality in layers, without requiring modification of the delivered application

ADF Business Components are implemented in metadata that you write by using wizards and that you edit declaratively. You can optionally expose framework code to add functionality if needed.

ADF BC Implementation Architecture

Characteristics of the Business Components layer of ADF:

- Is based on standard Java and XML
- Works with any application server or database
- Implements popular design patterns
- Organizes components into packages
- Provides prebuilt code in two main packages:
 - oracle.jbo
 - oracle.jbo.server
- Provides metadata-driven components with optional Java code



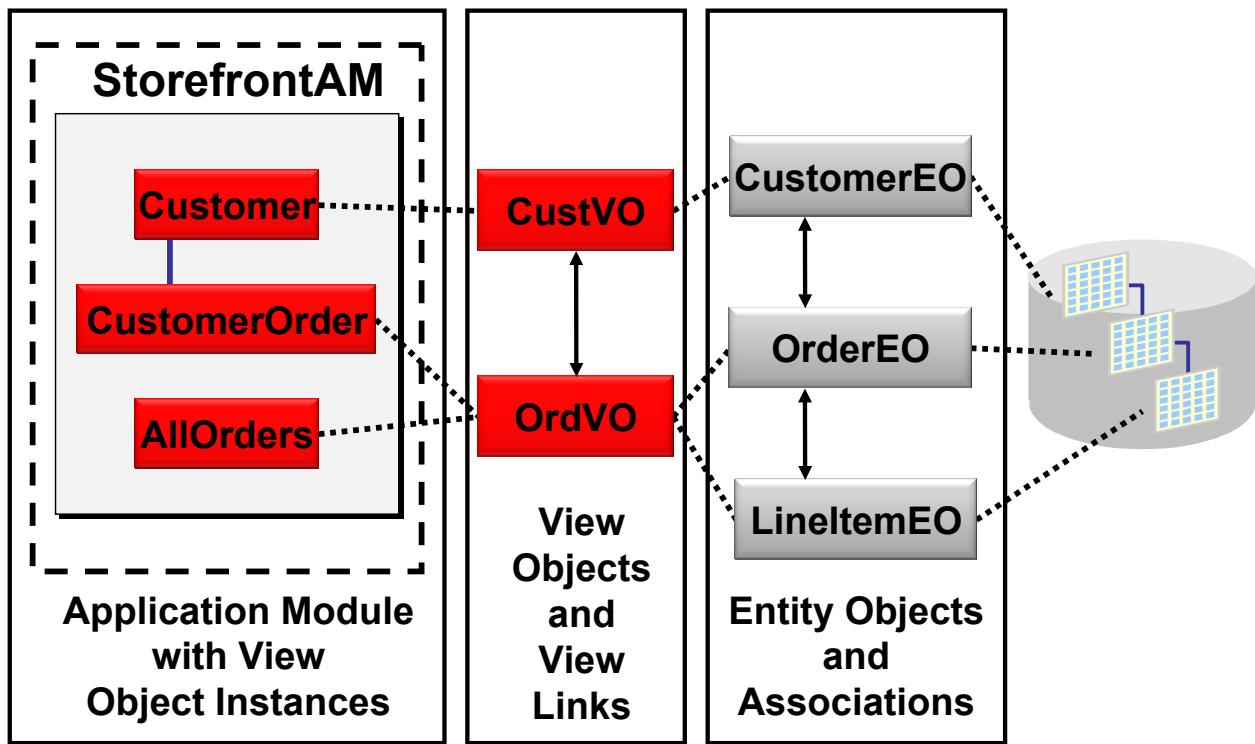
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ADF BC Implementation Architecture

Before entering into a detailed discussion on each of the key components in subsequent lessons, it is helpful to understand a few guiding principles that have gone into the design and implementation of this layer of Oracle ADF:

- It is based on standard Java and XML. Each component's run-time behavior is configured in an XML file, whereas the Java source code for the framework is extensible.
- It is database and application server agnostic. You can use any environment with a JVM.
- It implements all the popular Java EE design patterns, such as Model-View-Controller, Interface/Implementation Separation, and Session Facade.
- It organizes components into packages. To ensure that your code does not clash with reusable code from other organizations, you should choose package names that begin with your organization's name or Web domain name.
- Most of the classes and interfaces comprising the ADF BC prebuilt code are in the `oracle.jbo` or the `oracle.jbo.server` package.
- Components are metadata driven, but you can optionally generate a Java class where you can add code.

Types of ADF Business Components



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Types of ADF Business Components

There are two types of Oracle ADF Business Components objects that represent features of your data source. If you have a well-designed data source, the structure of these objects should reflect the structure of the data source. Data sources are represented by the following types of components:

- Entity objects, which represent objects in the data source (usually tables, views, and synonyms in a database)
- Associations, which represent relationships between these objects (such as foreign key relationships)

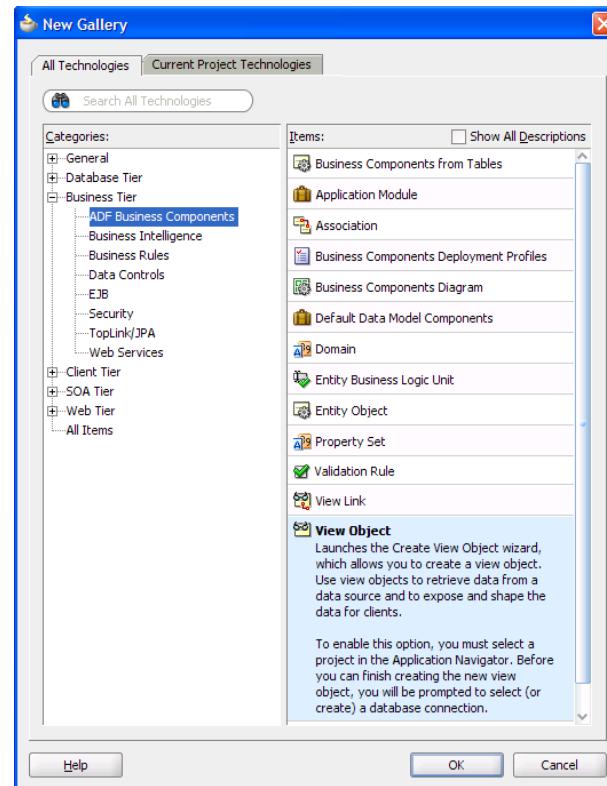
Other types of ADF BC objects collect data and present it to the client model. They should be designed on the client's specific data needs. For this reason, they are not as reusable as entity objects and associations. There are three such components:

- View objects, which collect data from the data source, usually by a SQL query
- View links, which represent relationships (such as master-detail relationships) between view object result sets
- Application modules, which provide a single point of access to the view objects and view links

Creating ADF Business Components

You can create ADF Business Components by:

- Creating a business components diagram
- Using the wizards



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating ADF Business Components

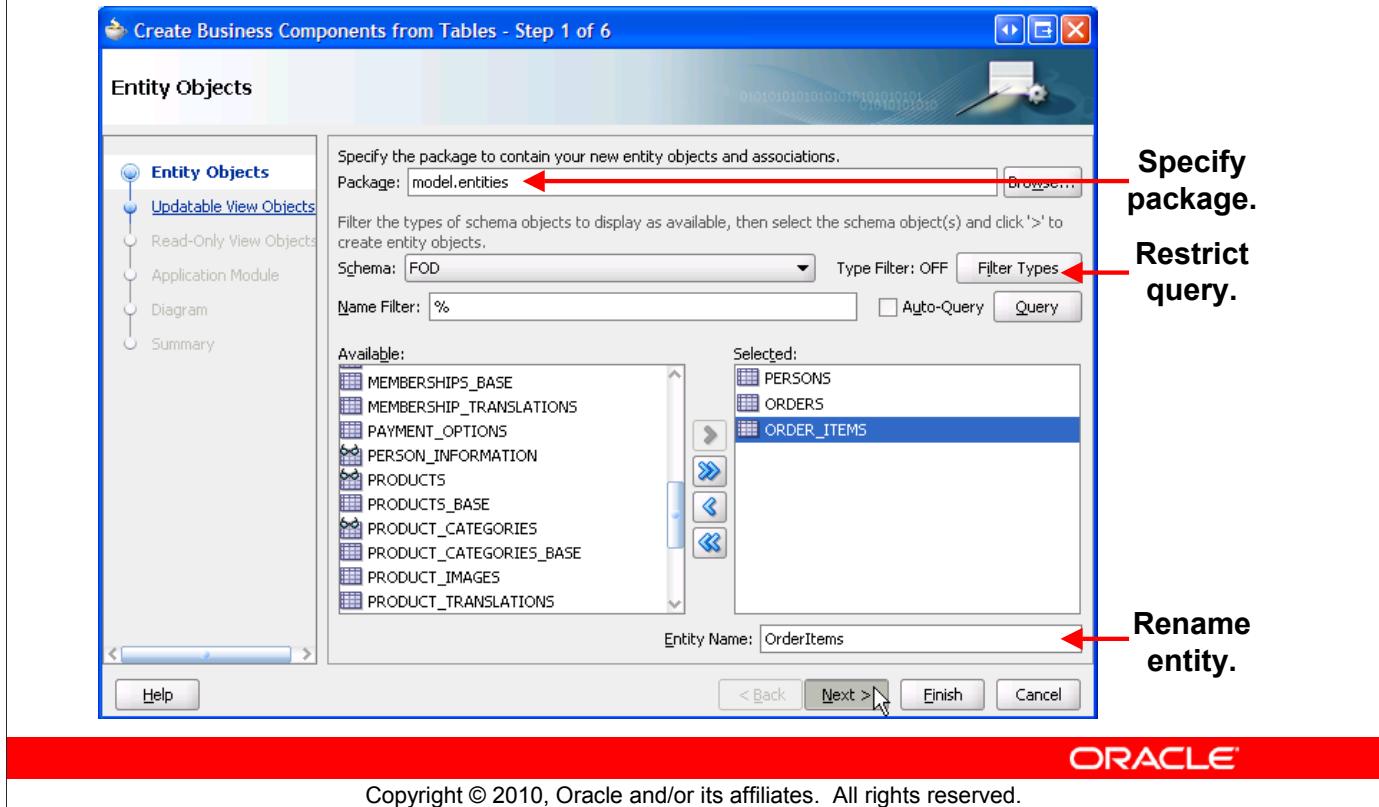
To create components, right-click the project that you want to contain the components and select New from the context menu. The New Gallery enables you to create several components in a variety of ways by invoking either a modeler or a wizard.

You can create business components by creating a business components diagram. Dragging database tables from the Database Navigator to the diagram creates entity objects for those tables. You can then use the Component Palette to create other types of objects. This enables you to visualize the relationships between the business components as well as to create and modify them visually. This course does not use the business components diagram, but you may want to experiment with it on your own.

Another way to create business components is by using wizards. You can use individual wizards, such as the Create Entity Object Wizard, to create one type of object at a time. Alternatively, you can use the Create Business Components from Tables Wizard to create several types of components at once, which is probably the quickest way to get a data model up and running. The next few slides show the pages in the “Create Business Components from Tables” Wizard. Subsequent lessons show you some of the other wizards.

Note: JDeveloper automatically invokes the “Create Business Components from Tables” Wizard when you create a new Business Components project.

Create Business Components from Tables Wizard: Entity Objects



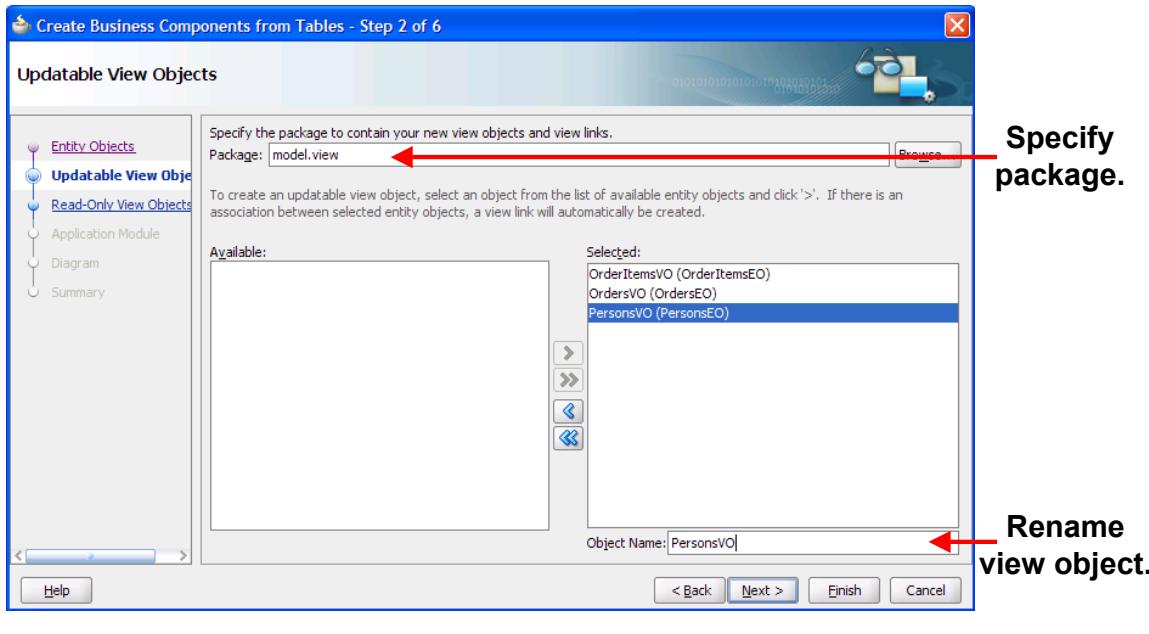
Create Business Components from Tables Wizard: Entity Objects

On the Entity Objects page of the “Create Business Components from Tables” Wizard, you can create default entity object definitions for the specified database objects. The wizard also creates associations based on all relevant foreign key relationships, and domains for all needed Oracle object types. This page enables you to create a large number of simple business components quickly and easily. Later you can edit any default business components that you create, and you can add new business components that may be needed.

On this page, you specify the name of a package to contain the entities. It is recommended that you keep the business domain components (entities and associations) in a separate package from the data model components.

Select the database schema that you want to use, and then click Query to display the Tables, Views, and Synonyms in that schema, or enter a name filter to display only certain ones. Then you can shuttle from the Available list to the Selected list the tables for which you want to create entity objects. You also can change the entity name if desired.

Create Business Components from Tables Wizard: Updatable View Objects



ORACLE

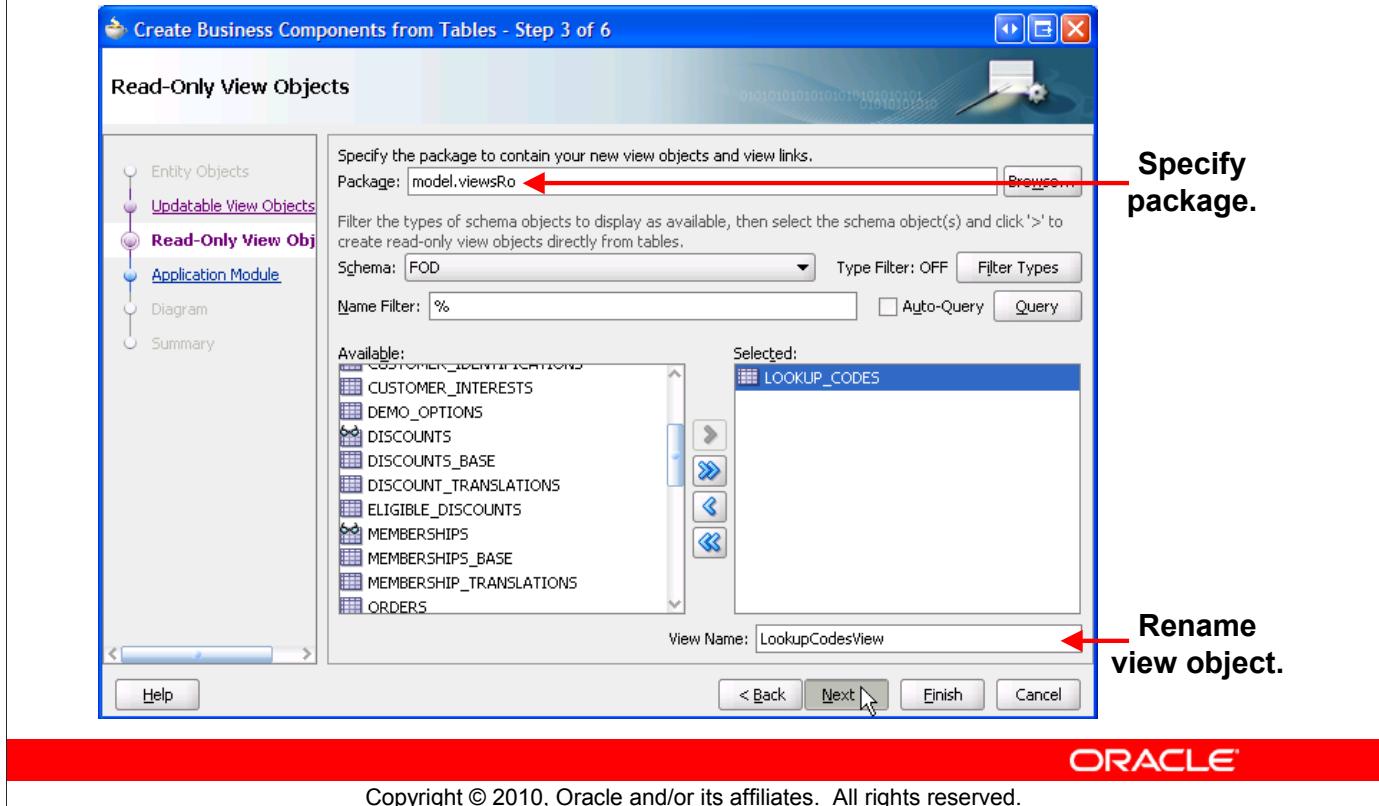
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Create Business Components from Tables Wizard: Updatable View Objects

On the Updatable View Objects page of the wizard, you can create default view object definitions that are updatable because they are based on the specified entity objects. The wizard also creates view links for all relevant association definitions. The view object definitions that you create on this page expose all attributes in the entity object.

On this page, you specify the name of a package to contain the view objects. Then you can shuttle from the Available list to the Selected list the entities for which you want to create view objects. You also can change the view object name if desired.

Create Business Components from Tables Wizard: Read-Only View Objects

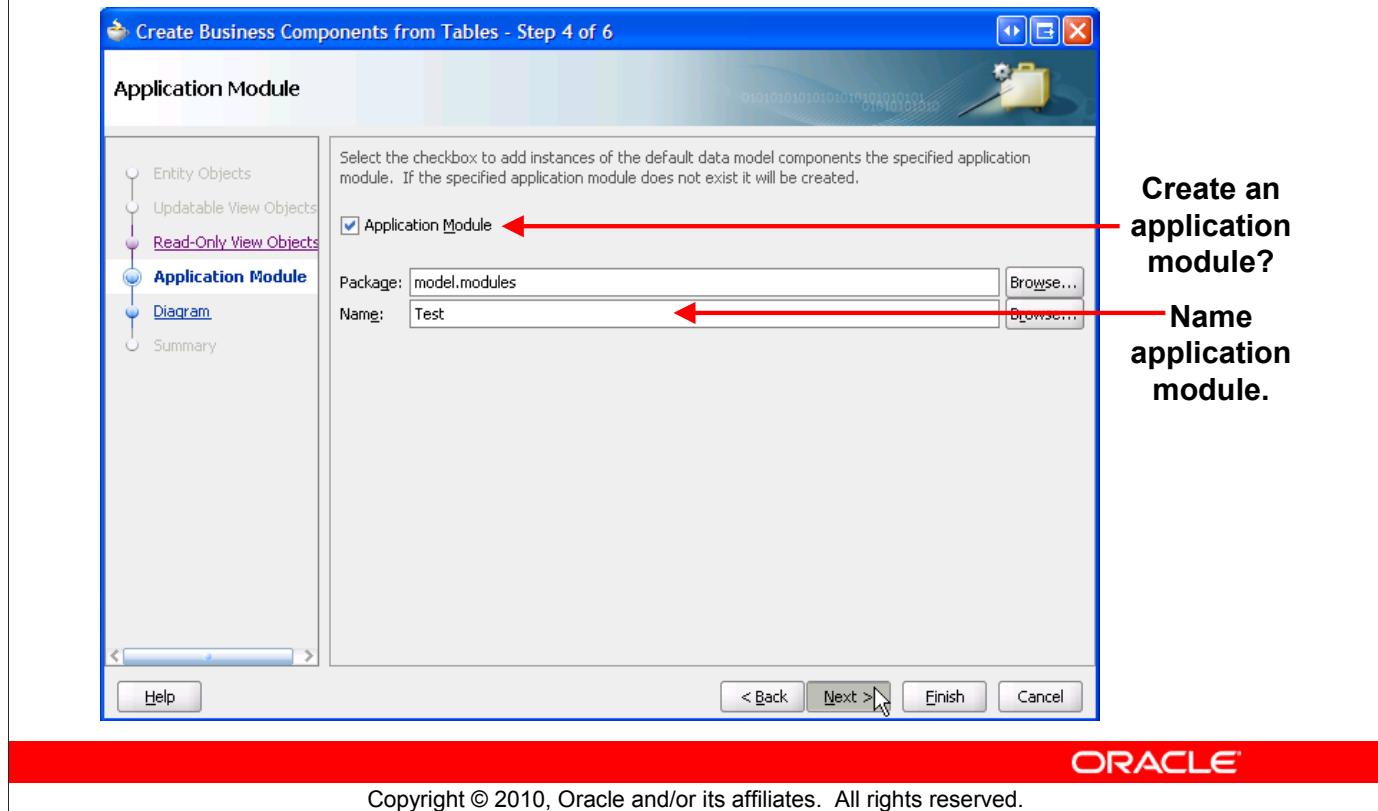


Create Business Components from Tables Wizard: Read-Only View Objects

On the Read-Only View Objects page of the wizard, you can create read-only view objects and view links for the specified database objects. Note that these are table names, not entity objects as was the case when you were creating updatable view objects.

On this page, you specify the name of a package to contain the view objects. Then as with entity objects, you select the database schema that you want to use and click Query to display the Tables, Views, and Synonyms in that schema, or enter a name filter to display only certain ones. Then you can shuttle from the Available list to the Selected list the tables for which you want to create read-only view objects. You also can change the view name if desired.

Create Business Components from Tables Wizard: Application Module



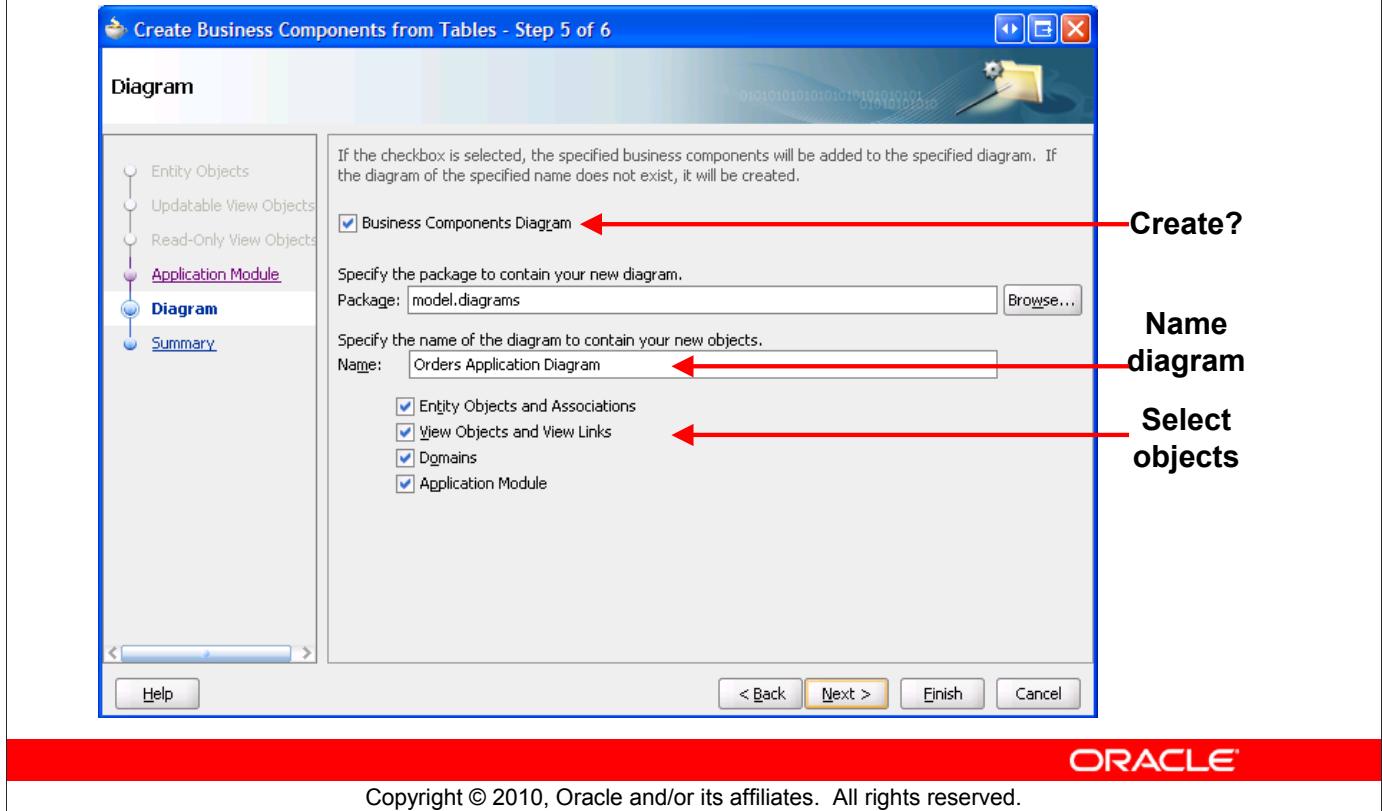
Create Business Components from Tables Wizard: Application Module

On the Application Module page of the wizard, you designate whether to add the business components that you are creating to an application module. Application modules enable you to test business components and also to expose them to a user interface.

If you choose to use an application module, you can create a default one or specify an existing one to contain instances of all the view objects, joined in a master-detail hierarchy in every possible way by the view link instances that the wizard is creating.

If you are creating a new application module, you specify the name of a package to contain it, and you give the application module a name.

Create Business Components from Tables Wizard: Diagram



Create Business Components from Tables Wizard: Diagram

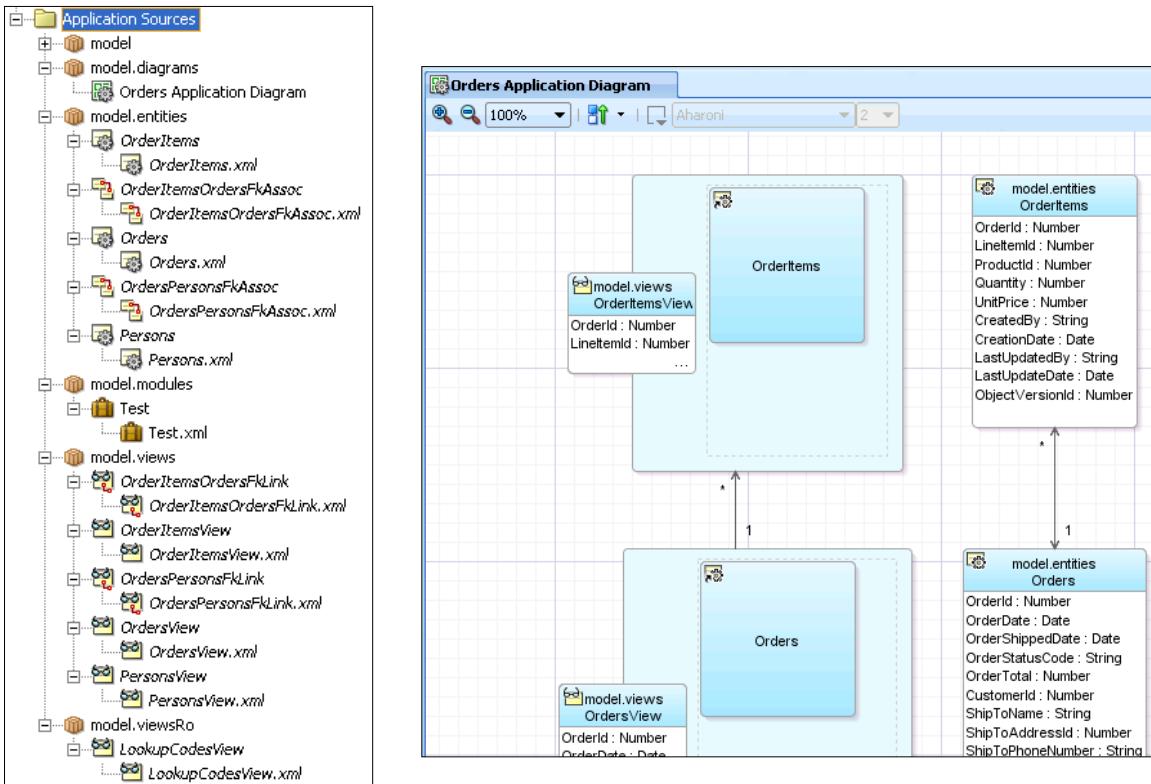
On the Diagram page of the wizard, you can choose to create diagram elements on a new or existing business components diagram. Diagramming is optional.

If you choose to create a diagram, on this page, you specify the diagram's package and name. You also can select which types of objects to depict on the diagram. Note that you cannot create diagram elements for objects that do not exist in the package. For example, you cannot create an application module on the diagram unless there is an application module in your business components package.

The next page of the wizard displays a summary of what you have specified for the wizard to create. Click Finish to create the business components.

The individual wizards are similar to this one, except that each creates only a specific type of component. They enable you to more specifically define each component to customize them to the needs of your application.

Examining Created Objects



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

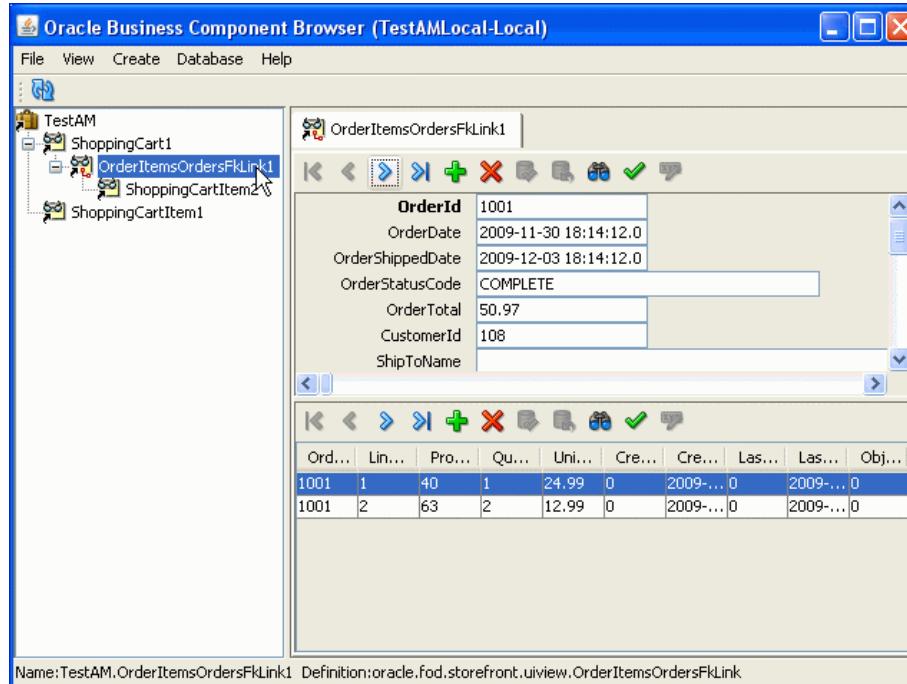
Examining Created Objects

The business components and the diagram that were created are shown in the Application Navigator, and if a diagram was created, it opens in the editor.

Note that the Application Navigator shows the packages that were specified in the wizard. In addition, each component that was created has an associated XML file to contain the metadata pertaining to that component.

By default, no Java files are created. You can create Java files, if desired, for those components to which you want to add code. This is discussed in the lesson titled “Programmatically Customizing Data Services.”

Testing the Data Model



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Testing the Data Model

JDeveloper provides a way to test the data model without creating a client application. You can perform this testing by using the Business Components Browser, sometimes also referred to as the Business Components Tester or the BC Tester.

To use the Business Components Browser to test the data model, right-click the application module in the Application Navigator and select Run from the context menu.

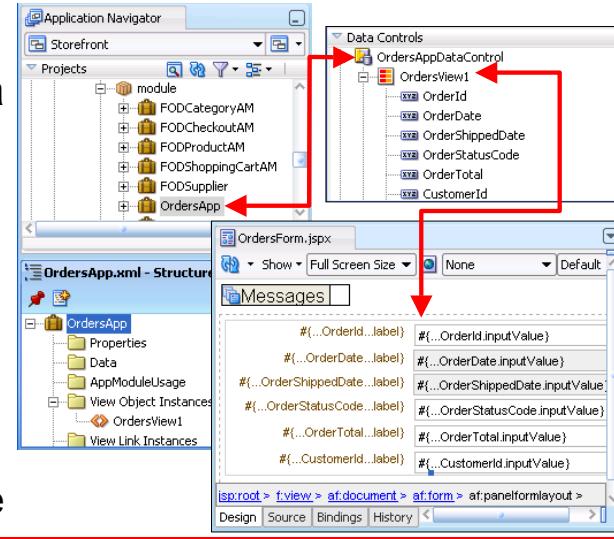
A Connect dialog box appears, where you select your database connection and click Connect. After the connection is established, the Business Components Browser displays a tree with the application module and all of its view object and view link instances. You can double-click a top-level view object instance to view it alone, or double-click a view link instance to view both master and detail in a hierarchy, with the master row displayed in form style and the detail rows displayed in tabular style.

You learn more about ADF Business Components and how to test them in subsequent lessons.

Exposing the Application Module to the User Interface

Oracle ADF Model:

- Abstracts service implementation from clients:
 - EJB, Web services, TopLink, Java classes, ADF BC
 - JSP, mobile
- Provides declarative data binding:
 - Implements JSR 227
 - Uses Expression Language (EL)
 - Separates view from business service
 - Creates data control from application module



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Exposing the Application Module to the User Interface

Oracle ADF Model abstracts the UI data binding from the technology that is used to implement the back-end business services.

ADF Model implements the JSR-227 service abstraction called the data control, and provides data control implementations for the most common business service technologies—EJB, Web services, TopLink, Java classes, and ADF Business Components. Whichever implementation you choose, JDeveloper and ADF work together to provide you a declarative, drag-and-drop data binding experience as you build the user interface for your application.

When you create an ADF BC application module, it is automatically exposed as a data control, including all the view object and view link instances that it contains.

The JDeveloper Data Control Palette exposes an application's data controls in the IDE, and enables you to use the drag-and-drop functionality to create UI components on a page. The UI components created by the Data Control Palette use declarative data binding, which means that the data binding expressions are automatically configured and that, in most cases, you do not have to write any additional code. The data bindings use Expression Language (EL).

It does not matter which technology you have used to build your business services—EJB, Web services or business components—the Data Control Palette looks the same and works in exactly the same way.

Summary

In this lesson, you should have learned how to:

- Describe the role of ADF Business Components in building a business service
- Explain the architecture of ADF BC
- Identify the types of components that cooperate to provide the business service implementation
- Use JDeveloper design-time facilities for ADF BC
- Explain how ADF BC components are used in a Web application



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 3 Overview: Building a Business Model

This practice covers the following topics:

- Creating Default Business Components
- Testing the Business Model



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 3 Overview: Building a Business Model

This exercise is to familiarize you with the default capabilities of JDeveloper and ADF Business Components.

In this practice, you create a business model using the wizards that are built into JDeveloper. You see how to build a default model without any coding. You then run the model in the Business Components Browser, a built-in application for testing business components.

Querying and Persisting Data

4

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Describe the characteristics of an ADF BC view object
- Create ADF BC view objects that can be used for data queries in a Web application
- Modify SQL statements in view objects
- Explain how entity objects relate to database tables
- Describe the persistence mechanism of entity objects
- Create entity objects from database tables
- Create associations between entity objects
- Create updatable view objects based on entity objects
- Link view objects to one another
- Refactor business components

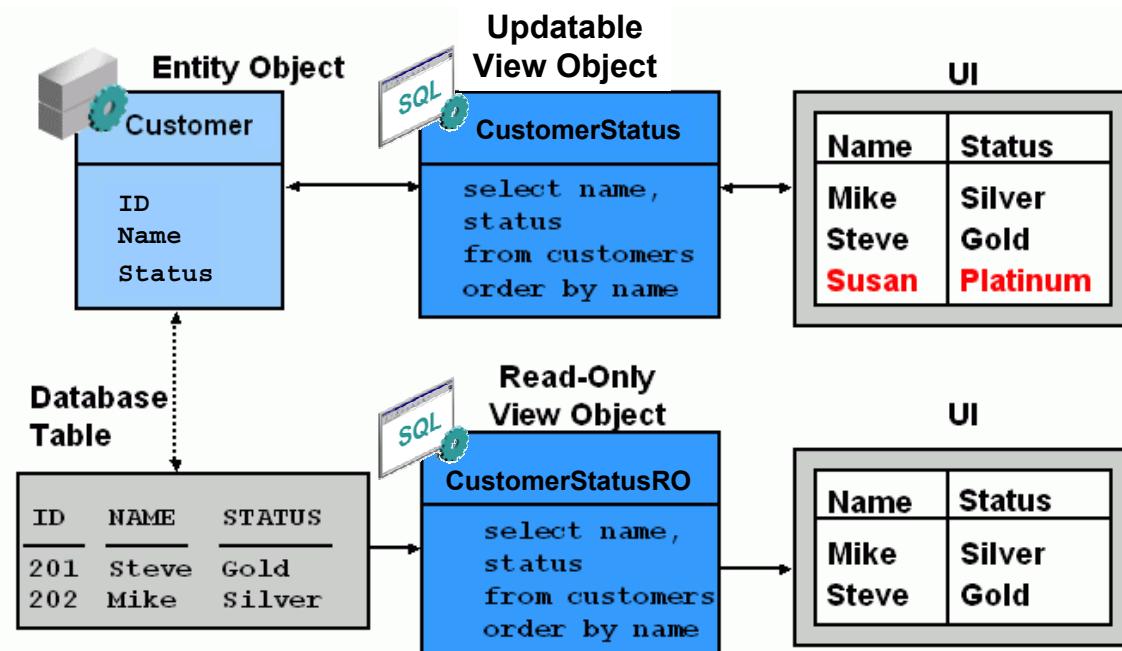


Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson teaches you how to enable Web applications to query, insert, update, or delete data in the database. You learn how to use ADF BC read-only view objects to provide a reusable mechanism for defining database queries. The entity object is the ADF BC mechanism for persisting data. You create entity objects from selected tables in the database schema, and then create updatable view objects based on them.

Using View Objects



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using View Objects

View objects provide the ability to query data in ADF BC applications. There are two types of view objects that you can use to retrieve data from the database:

- **Updatable View through Entity Objects:** Using entity object references enables view object instances to update data, use validation and other business rules from the entity object definition, and immediately synchronize data with other view object instances.
- **Read-only Access View:** SQL-only view objects bypass entity cache population and are faster for many applications. Note that view objects of this type save changes in memory only, and are not persisted in the database.

You can also populate the rows of a view object programmatically or by using a static list, but these types of view objects are not covered in this course.

Characteristics of a View Object (VO)

View objects:

- Represent a query
- Are used for joining, filtering, projecting, and sorting business data
- Enable you to have a view of data that is specific to one part of your application
- Can be constructed from a SQL statement, static values, or populated programmatically
- Can also be based on any number of entity objects



ORACLE

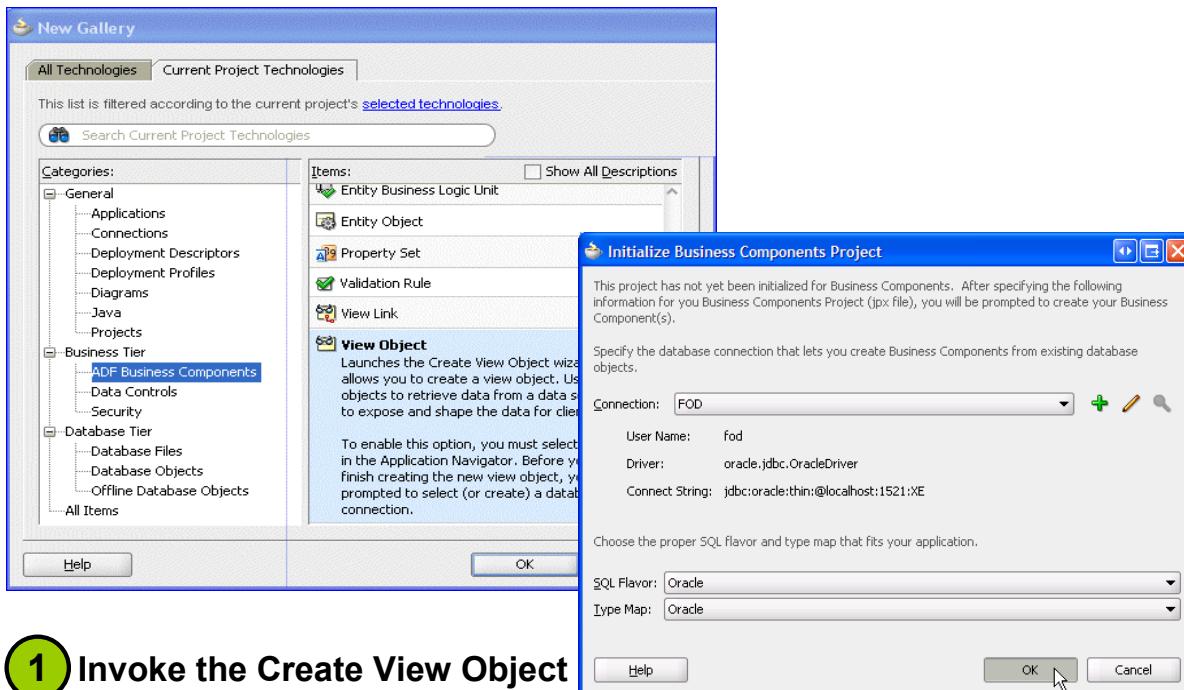
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Characteristics of a View Object (VO)

A view object is a reusable Oracle ADF business component that encapsulates a SQL query and simplifies working with its results. View objects have the following characteristics:

- They are defined by providing a SQL query or underlying entity object.
- They simplify the job of executing a query and working with the results.
- They can be linked to one or more other view objects to create master-detail hierarchies.
- They execute the query at run time to produce a row set of rows through which you can iterate by using a row set iterator.
- You can filter the row set a view object produces by applying a set of query-by-example criteria.
- You can create a view of data that is specific to your application or to a part of your application.
- You use view object instances in the context of an application module that provides the database transaction for the view object queries.

Creating View Objects for Queries



1 Invoke the Create View Object Wizard.

2 Initialize project for Business Components.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating View Objects for Queries

To create a view object, use the Create View Object Wizard. The wizard is available from the New Gallery in the Business Tier > ADF Business Components category. If it is the first component that you are creating in the project, the Initialize Business Components Project dialog box appears. In this dialog box, you can select a database connection and specify the SQL flavor and type map for your application.

The **SQL Flavor** setting controls the syntax of the SQL statements that your view objects will use and the syntax of the DML statements that your entity objects will use. If JDeveloper detects that you are using an Oracle database driver, it defaults this setting to the Oracle SQL flavor. The supported SQL flavors include:

- **Oracle** – The default, for working with Oracle
- **Olite** – For the Oracle Lite database
- **SQLServer** – For working with a Microsoft SQL Server database
- **DB2** – For working with an IBM DB2 database
- **SQL92** – For working with any other supported SQL92- compliant database

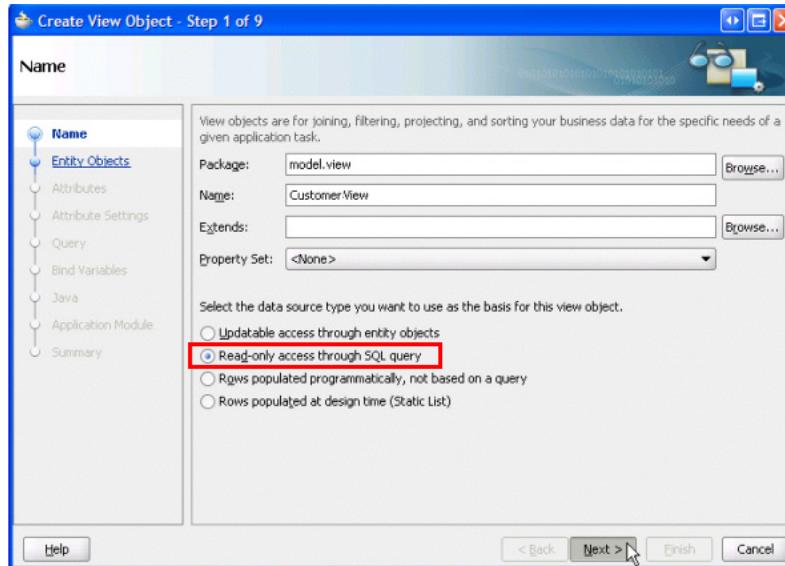
Creating View Objects for Queries (continued)

The **Type Map** setting controls whether you want this project to use the optimized set of Oracle data types, or use only the basic Java data types. If JDeveloper detects that you are using an Oracle database driver, it defaults this setting to the `Oracle` type map. The supported type maps are:

- `Oracle` – Use optimized types in the `oracle.jbo.domain` package
- `Java` – Use basic Java types only

Note: You can change the database connection information later by editing project properties, but the SQL flavor and type map cannot be changed after the initialization of the project. Therefore, if you plan to have your application run against both Oracle and non-Oracle databases, you should select the `SQL92` SQL flavor when you begin building your application, not later. Although this makes the application portable to both Oracle and non-Oracle databases, it sacrifices using some of the Oracle-specific optimizations that are inherent in using the Oracle SQL flavor.

Creating View Objects for Queries



3 Specify package, name, and kind of data.

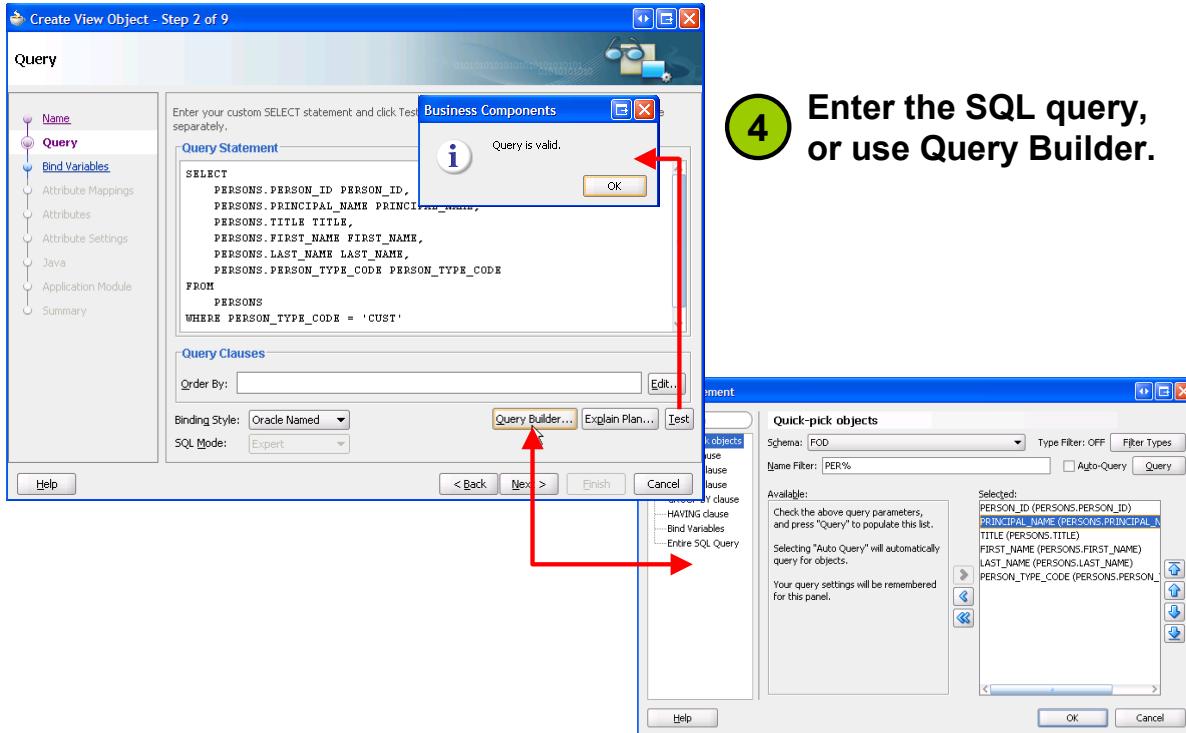
ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating View Objects for Queries (continued)

The Name page of the Create View Object Wizard enables you to supply the view object name and the package to contain it, as well as the kind of data you need the view object to manage. For the read-only view object, select “Read-only access through SQL query.”

Creating View Objects for Queries



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating View Objects for Queries (continued)

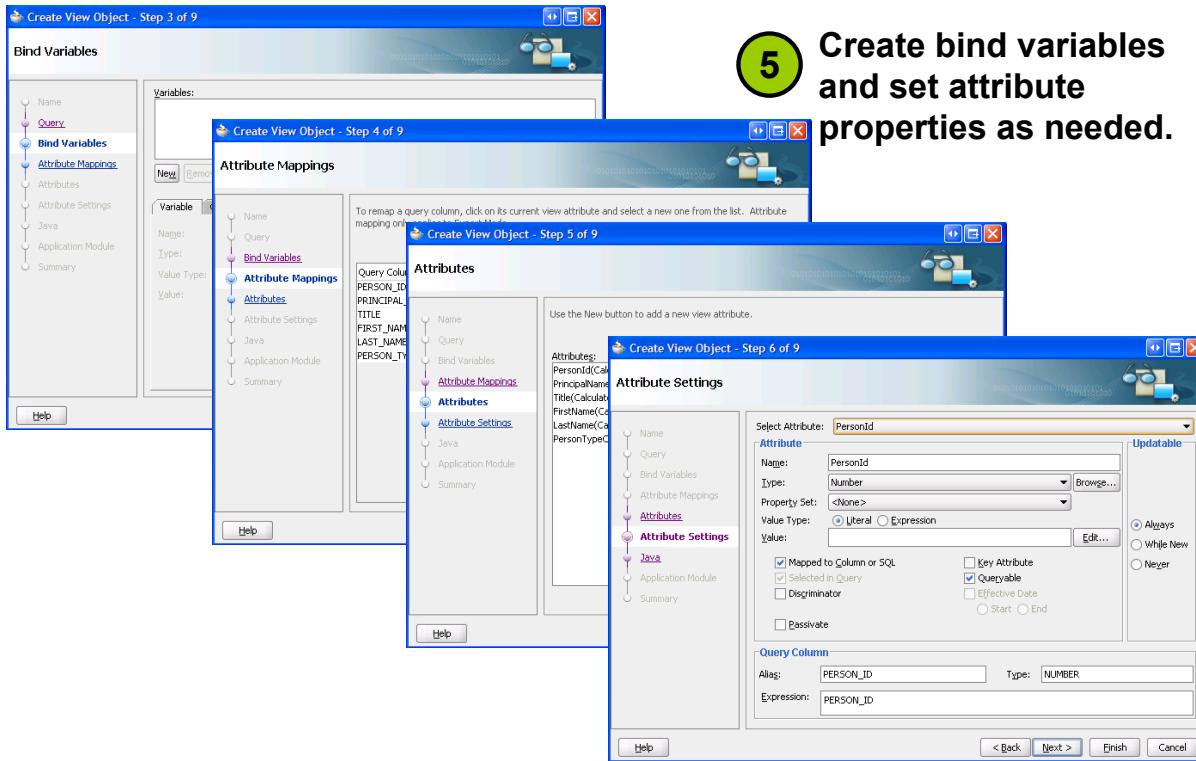
The SQL Statement page of the wizard enables you to enter a custom SQL statement and to check its syntax. You also can specify an ORDER BY clause. There are buttons to invoke a query builder, to perform an explain plan, and to test the syntax of the query.

You can type in the query if desired, but clicking the Query Builder button enables you to build the query and helps to avoid typing mistakes. In the SQL Statement dialog box of the Query Builder, you can select Quick-pick objects in the tree at the left. Then select the schema and click Query to show the available tables or views, which you can expand to show the attributes. You select the objects from the Available list and shuttle them to the Selected list to populate the query with the desired attributes and FROM clause. Selecting attributes under a foreign key ensures that tables are joined based on the foreign key. You can also define aliases on the Quick-pick objects page.

Selecting other elements from the tree at the left displays other pages that enable you to define various clauses or bind variables for the query.

After you have written the query, click Test for a syntax check.

Creating View Objects for Queries



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

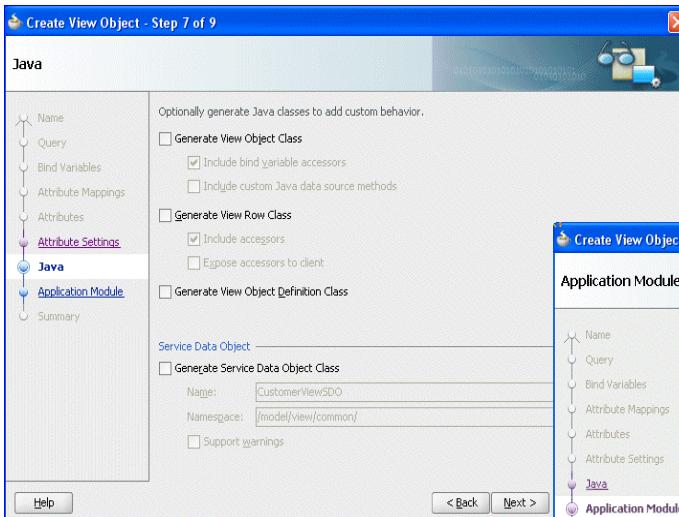
Creating View Objects for Queries (continued)

On the next few pages of the Create View Object Wizard, you can define:

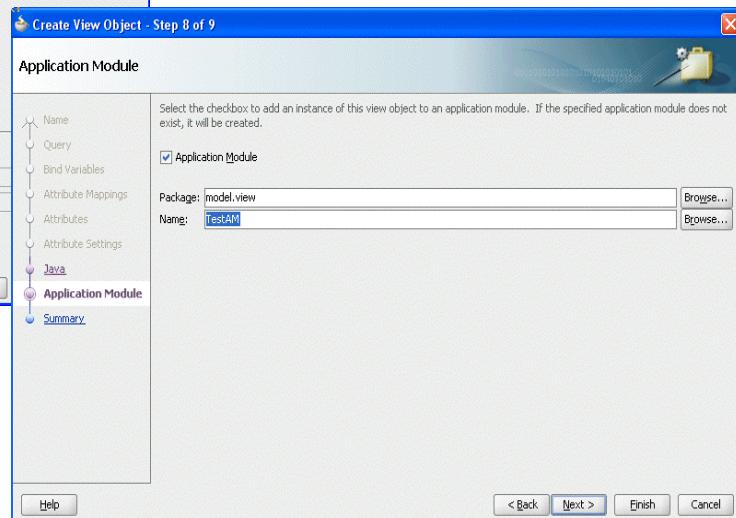
- Bind variables
- Attribute mappings
- Attributes
- Attribute settings

These pages enable you to create bind variables and set properties on attributes. Bind variables are discussed in a later lesson. If you create the view object without setting values for these pages, you can edit the view object later to set them.

Creating View Objects for Queries



6 Optionally, create Java classes (if adding programmatic business logic).



7 Optionally, create a new application module or add to an existing one.

ORACLE

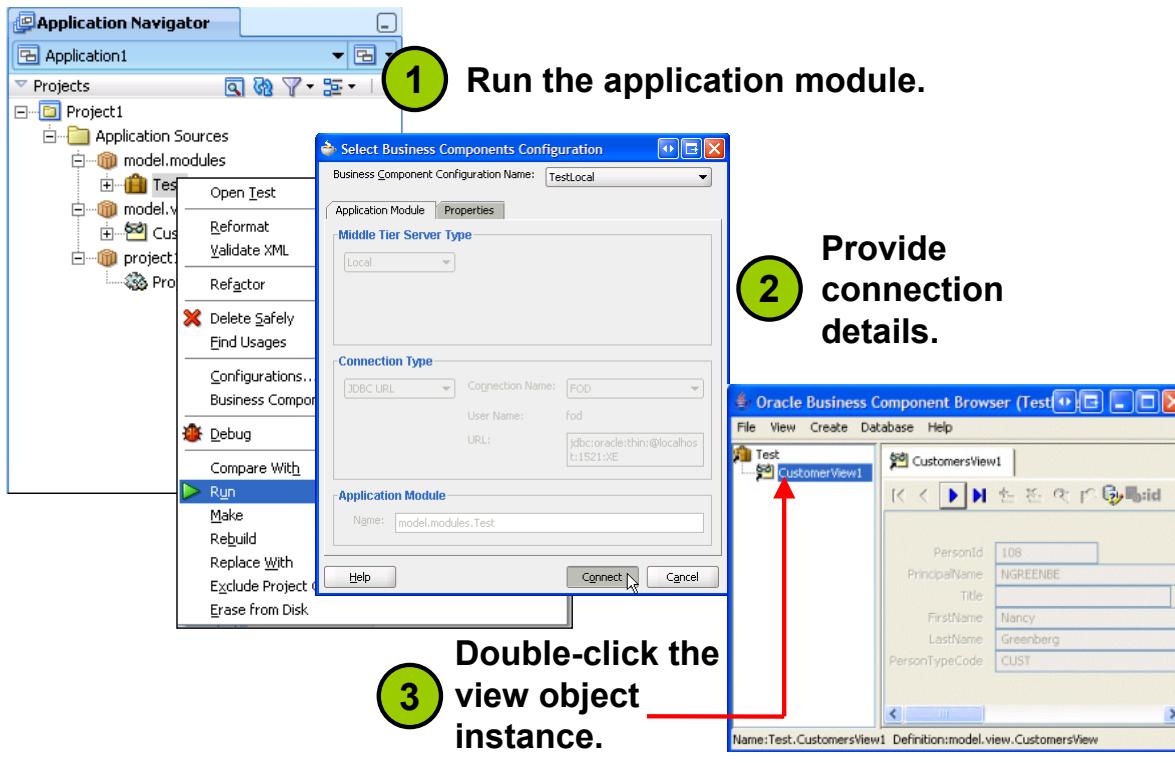
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating View Objects for Queries (continued)

Business components by default are defined in .xml files. However, the Java page of the wizard enables you to optionally create .java files where you can add custom code if desired. If you create the view object without creating the Java files, you can create them at a later time.

Application modules are discussed in a later lesson, but for now you should understand that application modules provide a way to expose business components to applications. To be able to test your view object, it must be contained in an application module. The last step of the Create View Object Wizard enables you to optionally put the view object into an application module. If you choose not to do so now, you may add the view object to an application module later.

Testing View Objects with the Business Components Browser



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Testing View Objects with the Business Components Browser

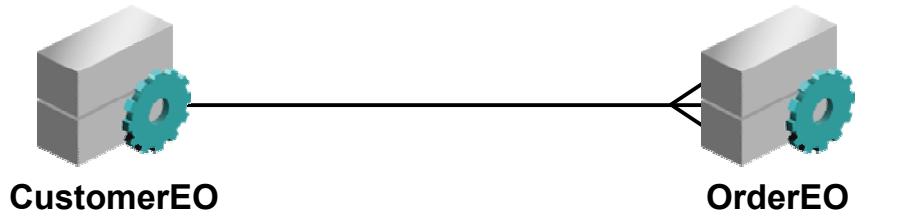
As explained previously, a view object must be contained in an application module before you can test it. To test a view object, perform the following steps:

1. Right-click its application module in the Applications Navigator and select Run from the context menu.
2. In the Business Component Browser Connect dialog box, select the connection to use, and then click Connect.
3. In Business Component Browser, double-click the view object to execute it. You can then use the toolbar buttons to navigate between records, to set values for bind variables, or to specify view criteria.

Characteristics of an Entity Object (EO)

Entity objects:

- Represent a row in a database table or other data source
- Handle database caching
- Contain attributes representing the database columns
- Encapsulate attribute-level and entity-level validation logic
- Can contain custom business methods



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

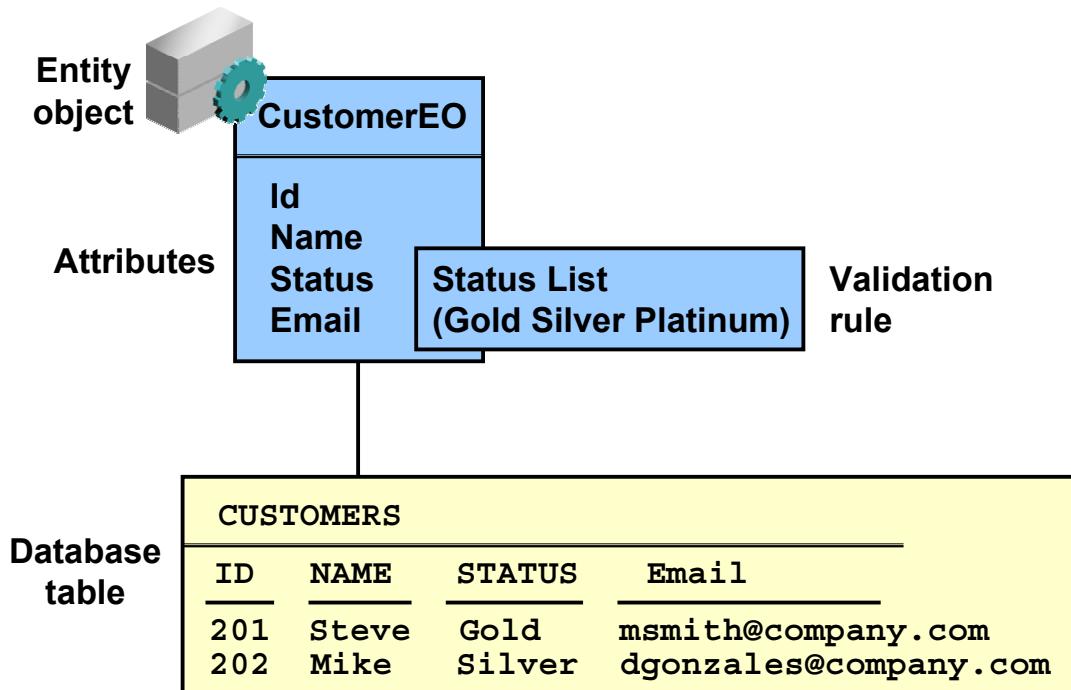
Characteristics of Entity Object (EO)

An entity object is the ADF BC component that represents a row in a database table and simplifies modifying its data. It enables you to encapsulate domain business logic for those rows to ensure that your business policies and rules are consistently validated.

Each entity object maps to a data source, usually a database table or database view. Entity objects handle database caching, so that changes to data are cached in the entity object before being committed to the database.

Entity objects are the foundation of the Business Components technology. Each entity object represents a business object, or business entity, in your application. Entity objects handle business rules and validation and can contain custom business methods.

Using Entity Objects to Persist Data



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Entity Objects to Persist Data

Entity objects contain attributes, business rules, and persistence information that apply to a specific part of your business model. Entities are used to define a logical structure of your business, such as product lines, departments, divisions, and so on. They can also define physical items, such as warehouses, employees, and equipment. For example, CustomerEO is an entity object.

- An entity object is based on a data source; the example in the slide is based on the Customers table.
- An entity's attributes map to the columns of the data source; the CustomerEO entity has attributes called ID, Name, Status, and Email that map to the corresponding columns of the CUSTOMERS table.
- You can attach validation rules to an entity object; the Status attribute has a validation rule that restricts status values to a list of valid values.

You retrieve and modify entity rows in the context of an application module that provides the database transaction. When your application module creates, modifies, or removes entity object rows and commits the transaction, changes are saved automatically.

Creating Entity Objects

The Create Entity Object Wizard:

Name page

This page allows you to specify the name and package for the entity object, select the database schema, and choose the schema object on which to base the entity object. It also includes sections for Entity objects, Data Source, and Extends Entity.

Attributes page

This page lists the attributes for the entity object, including their names, descriptions, and IDs. It provides buttons for creating new attributes and removing existing ones.

Attribute Settings page

This page provides detailed settings for each attribute, such as type, value type, and persistence. It also includes sections for Database Column and Refresh After.

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Creating Entity Objects

You can create entity objects with the Create Entity Object Wizard that you invoke from the New Gallery (File > New > Business Tier > ADF Business Components > Entity Object). This wizard contains the following pages:

- **Name:** Specify the name and package for the entity object; select the database schema to use; select one or more check boxes for the types of schema objects to display; select the schema object on which to base the entity object.
- **Attributes:** Add or delete attributes from the entity object.
- **Attribute Settings:** Change the following entity object settings for attributes that you select from a drop-down list:
 - **Name:** It is a valid Java identifier.
 - **Type:** It is a Java data type.
 - **Property Set:** Select a named property set to apply to this attribute. A property set is a version of an existing domain that is XML-only (no Java) and does not enforce a data type.
 - **Value Type:** It indicates whether the default value is a literal or an expression.
 - **Value:** This is the default value (optional). This value is not propagated from the default value, if any, in the database table.
 - **Persistent:** Select this option if the attribute is persistent; deselect for transient attributes.

Creating Entity Objects (continued)

- **Mandatory:** This is selected by default for columns with NOT NULL constraints.
- **Change Indicator:** Select this option if the column is a change indicator, such as a timestamp, to be used to indicate that a row has been changed by another transaction. (If no columns are specified as change indicators, ADF BC does a column-by-column comparison.)
- **Derived from SQL Expression:** This indicates that this attribute is derived from a SQL expression. When selected, you can enter the column Type and an Expression.
- **Discriminator:** Select this option if this is a discriminator column for a polymorphic entity object.
- **Primary Key:** This is selected by default for columns with PRIMARY KEY constraints.
- **Unique:** Specify that the corresponding table column is to be generated with a UNIQUE constraint.
- **Queryable:** Specify that this attribute can occur in the WHERE clause of a view object (selected by default except for LOBs).
- **Effective Date:** The attribute is either the start or end date of a date range for which the entity row is effective (used for a point of time snapshot to answer such questions as what an employee's salary was on January 30, 2004.)
- **History Column:** Select to use this column to log changes to the database. You can log changes only if you have implemented Java Authentication and Authorization Service (JAAS) authentication, the selected attribute is persistent, you have not selected Primary Key, Mandatory, or Discriminator, and the attribute is Char, Character, String, Date, Timestamp, or Number. After you have selected History Column, you can select the history column type (*created on* or *modified on* for a Date or Timestamp attribute, *created by* or *modified by* for a Char, Character, or String attribute, or *version number* for a Number attribute).
- **Updatable:** You can enable the attribute to always or never be updatable, or enable it to be updated only before the entity is first posted (while new).
- **Refresh After:** Specify whether to retrieve the value from the database after an update or insert.
- **Database Column:** This indicates the name and SQL data type of the column to which the attribute is mapped.
- **Java:** Specify whether to create Java files to implement custom logic for validation and default values in the entity class or to override base class methods. You can opt to create Java files for any or all of the entity collection class, the entity object class, or the entity definition class. Otherwise, only XML files are generated.
- **Generate:** It enables you to create a default view object and application module from this entity object.
- **Summary:** It presents a summary of the selected options.

Creating Entity Objects from Tables, Views, or Synonyms

When you create an entity object, JDeveloper:

- Interrogates the data dictionary for information
- Infers primary key, or creates one from RowID
- Creates implicit validators for database constraints
- Creates the XML component definition file (*<EO_name>.xml*)
- Creates optional Java files if selected, such as entity object class *<EO_name>Impl.java*
- Generates associations based on foreign keys, if applicable (*<Association_name>FkAS.xml*)



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating Entity Objects from Tables, View, or Synonyms

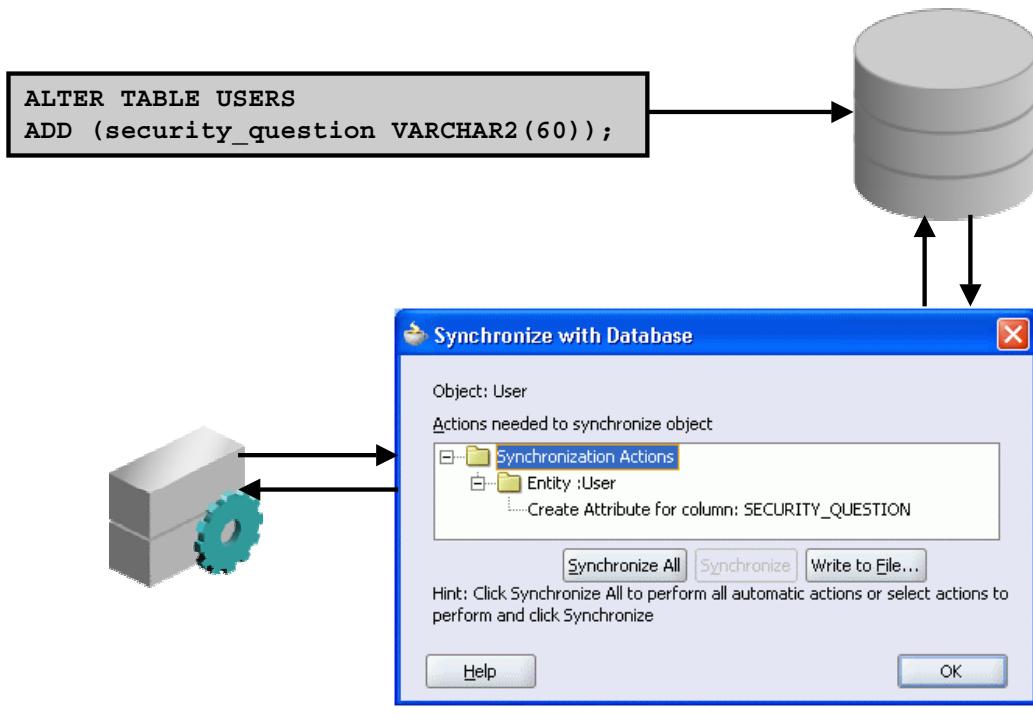
When you create an entity object from an existing table, first JDeveloper uses the online database connection to interrogate the data dictionary to infer the following information:

- The Java-friendly entity attribute names from the names of the table's columns (for example, DEPT_ID becomes DeptId)
- The SQL and Java data types of each attribute based on those of the underlying column
- The length and precision of each attribute
- The primary and unique key attributes (If the table has no primary key, it uses RowID.)
- The mandatory flag on attributes, based on NOT NULL constraints
- The relationships between the new entity object and other entities based on foreign key constraints

JDeveloper then creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package.

If you create an entity object from a view, or from a table without a primary key, by default RowID is used as a primary key. If you create an entity object from a synonym, that entity object behaves as if it is created on the underlying table or view.

Synchronizing an Entity Object with Changes to Its Database Table



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Synchronizing an Entity Object with Changes to Its Database Table

If you alter a table for which you have already created an entity object, the existing entity is not disturbed by the presence of additional attributes in its underlying table. However, if you want to access the new table column in your application, you first need to synchronize the entity object with the database table. To perform this synchronization from JDeveloper, right-click the entity object in question and select “Synchronize with Database” from the context menu.

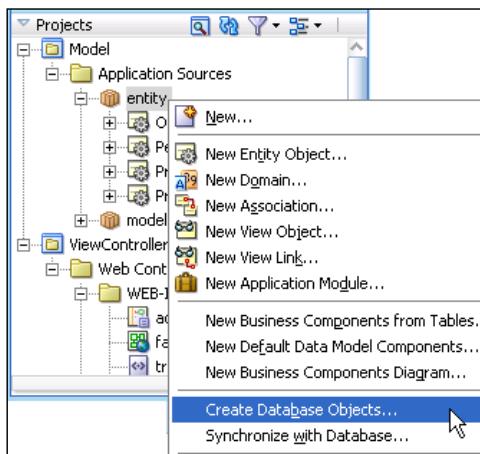
For example, suppose you had issued the following SQL*Plus command to add a new SECURITY_QUESTION column to the USERS table:

```
ALTER TABLE USERS ADD (security_question VARCHAR2(60));
```

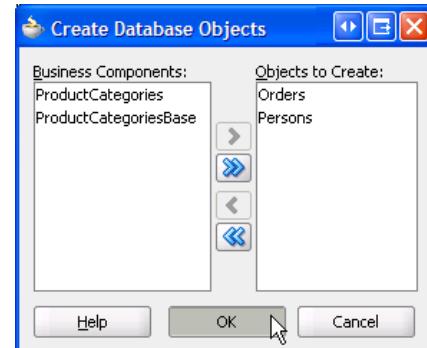
When you use the synchronize feature on the existing User entity, the “Synchronize with Database” dialog box proposes the changes that it can perform for you automatically. By clicking the desired button, you can carry out the synchronization.

Generating Database Tables from Entity Objects

Right-click package and select Create Database Objects.



In the Create Database Objects dialog box, select objects to create.



Use with caution!

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

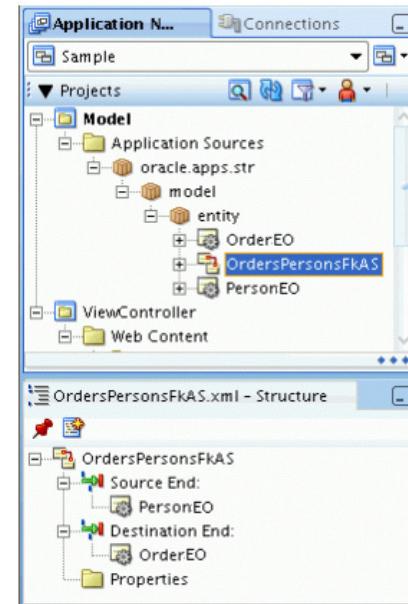
Generating Database Tables from Entity Objects

You can create database tables from entity objects. Right-click the package that contains the entity objects and select Create Database Objects from the context menu. Then in the Create Database Objects dialog box, select the object to create. Another dialog box confirms whether you want to do this before proceeding.

Use caution when performing this generation, especially on entities that are based on existing tables. The feature does not generate a script to run later, but performs its operations directly against the database and drops existing tables.

Characteristics of Associations

- Define a relationship between EOs
- Facilitate access to data in related entity objects
- May be based on database constraints
- May be independent of database constraints
- Are used in defining validations and LOV metadata
- Consist of a source (master) and a destination (detail) entity



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Associations

As stated earlier, the “Create Business Components from Tables” Wizard in JDeveloper automatically creates associations between entities that represent tables that are joined by a foreign key constraint. Associations are like master-detail relationships in a relational database. However, you can also create an association manually, as follows:

1. Right-click the package name.
2. Select New Association from the Context menu.

This displays the Create Association Wizard.

Viewing the Components of an Association

When you select an association in the Application Navigator, the Structure window displays the source (master) entity and the destination (detail) entity. In this example, the PersonEO entity is the source in the association and the OrderEO entity is the destination. That means that each person can have multiple orders, and that every order must belong to one person.

The Usefulness of Associations

Associations can be useful in helping you to create a more comprehensive representation of the business. For example, they allow you to “walk” between entity objects, using the find-by-primary-key query. When you traverse entity associations in your model, if the entities are not already in the cache, the ADF Business Components framework performs the query to bring the entity (or entities) into the cache.

Associations (continued)

They can also perform autolookups of reference information when foreign key values change. In addition, when working in declarative SQL mode, the view object's metadata can cause the ADF Business Components run time to generate table joins for you.

Generated Files

JDeveloper generates only one file for each association: *<Association>.xml* (for example, *OrdersPersonsFkAS.xml*), which contains all the metadata for the association.

Creating Associations

Use the Create Association Wizard:

The screenshot shows the Oracle JDeveloper Create Association Wizard interface with four pages:

- Name page:** Shows the package as "model.entities.associations", name as "PersonToAddressFkAS", and extends as "".
- Entity Object page:** Shows the selection of source and destination attributes. The source attribute is "Person.PrimaryAddressId" and the destination attribute is "Address.AddressId". A red arrow points from this page to the Association Properties page.
- Association Properties page:** Shows the association properties. Under "Source Accessor", Entity Object is "Person" and Accessor Name is "Person1". Under "Destination Accessor", Entity Object is "Address" and Accessor Name is "Address1". Other options like "Composition Association" and "Lock Level" are also shown. A red arrow points from the Entity Object page to this page.
- Association Query page:** Shows the query clauses. The source query clause is ":Bind_PrimaryAddressId = Address.ADDRESS_ID" and the destination query clause is ":Bind_AddressId = Person.PRIMARY_ADDRESS_ID". Buttons for "Test" and "Explain Plan..." are available for both clauses. A red arrow points from the Association Properties page to this page.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

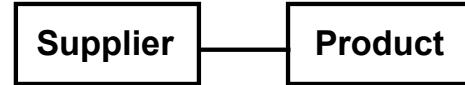
Creating Associations

Note that when creating associations, JDeveloper can also optionally generate accessors in the associated entities. It may be appropriate to turn this off if you do not own both entities involved.

Association Types

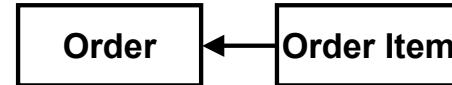
- Association

- Entities are related but not completely dependent.
- Either end of the association can exist without the other.
- Either can be deleted without deleting the other.



- Composition

- Destination entity is completely dependent on source entity.
- Source entity owns destination entity.
- No destination entity can be created without the owning entity existing first.
- Source entity cannot be deleted without deleting all its associated destination entities.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Types of Associations

Association: An association is a relationship between two entities that are not completely dependent on each other. Each end of the association may exist without the other, and either may be deleted without deleting the other. In Unified Modeling Language (UML), an association is represented as a line.

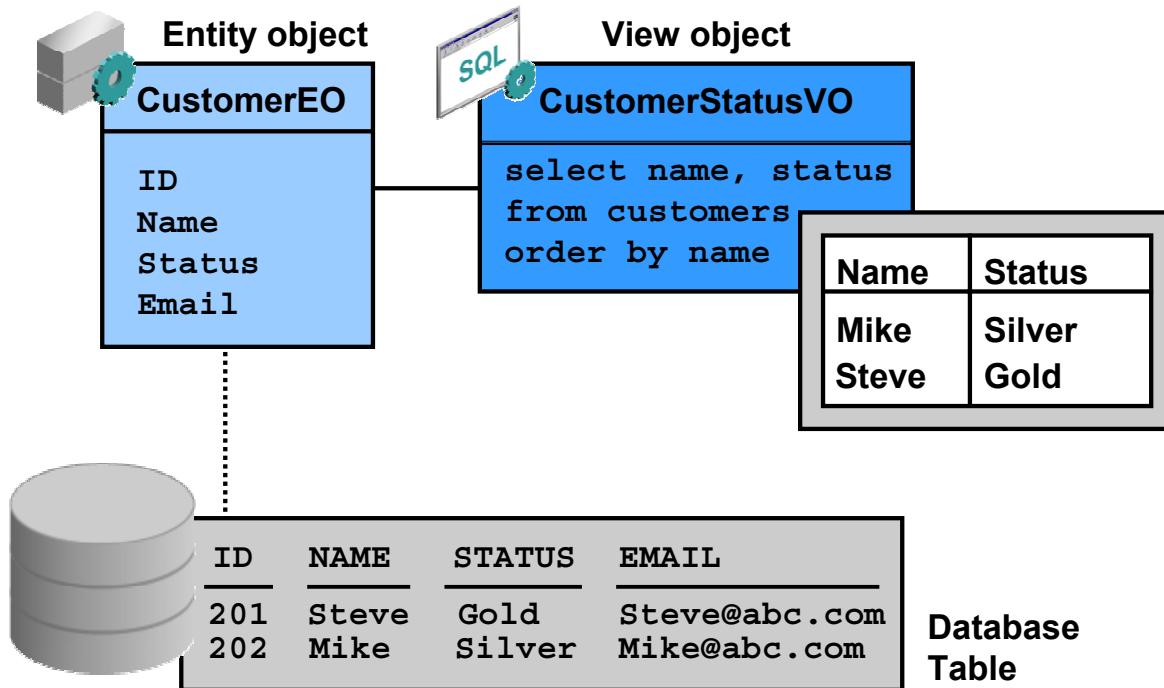
Composition: A composition relationship means that the source entity owns the destination entity object. The destination entity object cannot exist without the owning entity object existing first. The source entity object is a container for the destination entity object. For example, an order owns the order lines associated with it. In a composition, the source cannot be deleted until all the destination items are deleted. In UML, a composition is represented as a line with a solid diamond shape at the source end.

Compositions and Validation

When a change is made to the destination entity, a validate message is sent to the source. The source entity's `validate()` method is called when the transaction is committed.

The composition relationship also dictates the sequence of various operations at run time. For example, child validation is forced before parent. Parent posting is forced before child posting. You can control many additional composition-related features through settings on the Association Properties page of the Create Association Wizard, or via the Association Editor.

Characteristics of Updatable View Objects



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Characteristics of Updatable View Objects

So far you have learned to create view objects that are based on a SQL query. Such view objects can be used by client applications to present data in read-only mode.

You have also learned that the entity object is the ADF BC mechanism to interact with the database, enabling inserts, updates, and deletes. However, entity objects are not visible to client applications.

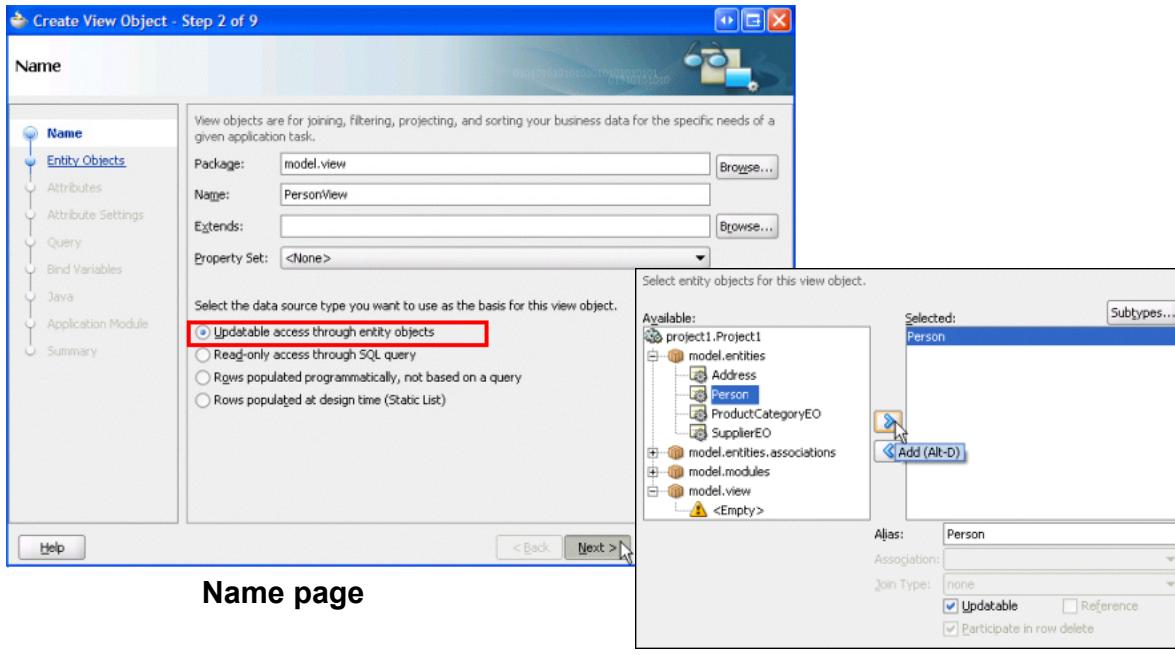
To retrieve and present to the client application the records to be updated, you need a different type of view object, one that is based on an entity object rather than on a SQL query. This is called an updatable view object.

The slide shows an example of a view object, called CustomerStatusVO. An updatable view object is bound to an entity object; this one is based on the CustomerEO entity object. A view object's attributes map to the attributes of the underlying entity object; the CustomerStatusVO view object has attributes that map to the Name and Status attributes of the underlying entity object.

Even though an updatable view object is based on an entity object, it still uses a SQL query (select) to sort and filter data, so you can specify ORDER BY and WHERE clauses for it.

Creating Updatable View Objects

Use the Create View Object Wizard:



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Creating Updatable View Objects

The process of creating an updatable view object begins with selecting the view object from the New Gallery to invoke the Create View Object Wizard, just as you did with read-only view objects. However, on the Name page of the wizard, you specify that you want the view object to have updatable access through entity objects, instead of read-only access as you do to create a read-only view object.

After you have selected to base the view object on entity objects, the next page of the wizard, the Entity Objects page, enables you to select from existing entity objects. If you select more than one, you can specify the join type and whether the second entity object is updatable or only for reference. These choices are discussed in more detail later.

Creating Updatable View Objects: Attributes and Settings

The screenshot shows two pages of a wizard for creating updatable view objects.

Attributes page: This page lists attributes available in the PersonView entity. On the left, under "Available:", there is a tree view with nodes like PersonView, Person, PersonId, PrincipalName, Title, FirstName, LastName, PersonTypeCode, SupplierId, and PrimaryAddressId. On the right, under "Selected:", there is a list box. Between them are arrows for moving attributes between the lists. A button labeled "Add All (Alt-L)" is also present. A "New..." button is located at the bottom right of the list box.

Attribute Settings page: This page provides detailed settings for a selected attribute, "PersonId". The "Attribute" section includes fields for Name (PersonId), Type (Number), Property Set (<None>), Value Type (Literal), and Value. It also includes checkboxes for Mapped to Column or SQL, Selected in Query, Discriminator, and various query-related options like Key Attribute, Queryable, Effective Date, Start, and End. The "Query Column" section defines an alias (DEPARTMENT_ID) and type (NUMBER(4, 0)). To the right of the main settings area, there is a "Updatable" section with radio buttons for Always, While New, and Never.

Attribute Settings page

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating Updatable View Objects (continued)

The Attributes page presents all the attributes of the entities that you selected and enables you to select the ones that you want in the view object, thus restricting the columns returned by the query. You can use the arrows at the right to rearrange the order of the attributes, and you can use the New button to create calculated or transient attributes that are not part of the entity (more about these in the lesson titled “Declaratively Customizing Data Services”).

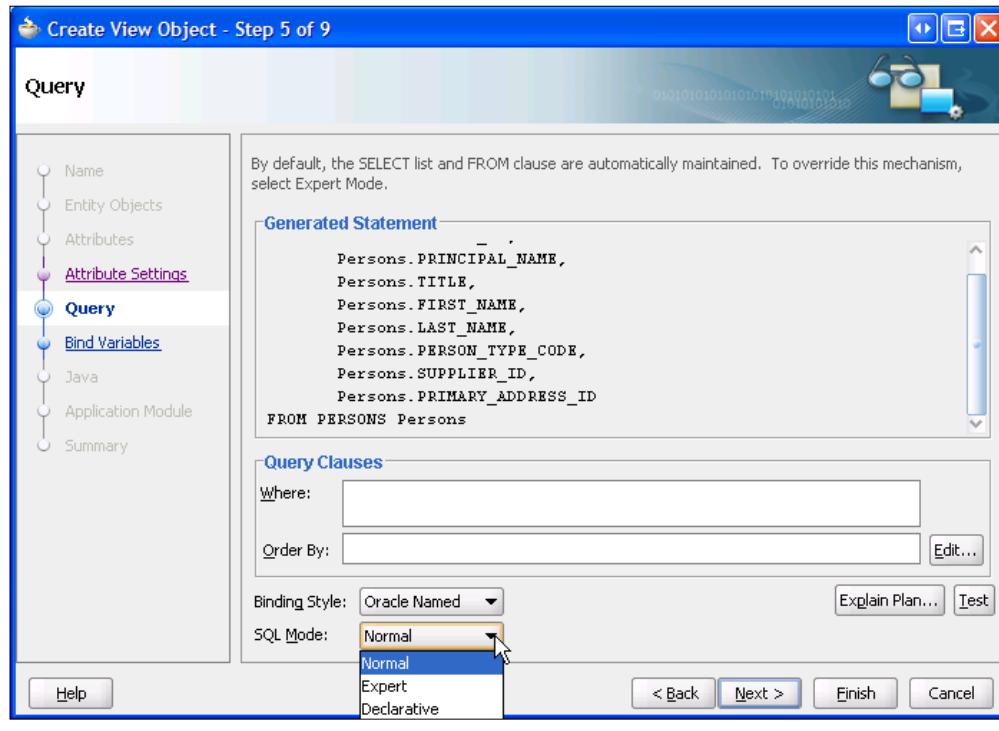
The Attribute Settings page of the wizard is similar to the one in the Create Entity Object Wizard. However, these settings apply to this view object only, and if different, override the attribute settings in the entity object. Different views may present data differently. You can specify the following settings for attributes (select the attributes from the drop-down list):

- **Name:** It is a valid Java identifier.
- **Type:** It is a Java data type.
- **Property Set:** Select a named property set to apply to this attribute. A property set is a version of an existing domain that is XML-only (no Java) and does not enforce a data type.
- **Value Type:** It indicates whether the default value is a literal or an expression.
- **Value:** It is the default value (optional).
- **Mapped to Column or SQL:** If selected, this attribute is mapped to a column or SQL statement. If deselected, this attribute is not yet mapped to a table. However, you can select the check box and provide the data in the Query Column group below.

Creating Updatable View Objects (continued)

- **Selected in Query:** When selected for a transient attribute, this attribute appears in the view object's SELECT statement. SQL-derived attributes always appear in the view object's SELECT statement.
- **Discriminator:** Select this option if this is a discriminator column for a polymorphic view object; this type of view object is discussed in the course *Build Applications with ADF II*.
- **Key Attribute:** When selected, this attribute forms part of the view object's key. Entity attributes defined as primary keys will automatically be selected as a Key Attribute when view objects based on the entity are created. You may need to modify this property for view objects based on more than one entity object, where multiple primary keys are defined.
- **Queryable:** It is selected if this attribute can occur in a view object's WHERE clause. It is selected by default except for LOBs. This option cannot be selected for LOBs.
- **Effective Date:** It is available only for view attributes that are based on entity attributes for which the effective date is defined. The attribute is either the start or end date of a date range for which the view row is effective (used for a point of time snapshot to answer such questions as what an employee's salary was on January 30, 2004.)
- **Updatable:** You can enable the attribute to always or never be updatable, or enable it to be updated only before the underlying entity is first posted (while new).
- **Query Column:**
 - **Alias:** Aliases are optional, but can be useful in SQL-derived attributes to prevent naming conflicts.
 - **Type:** This defines the SQL data type of the query column.
 - **Expression:** An expression is required for a SQL-derived attribute.

Creating Updatable View Objects: Query



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating Updatable View Objects (continued)

The Query page of the wizard enables you to see the SQL query that is generated and to set WHERE and ORDER BY clauses for it. There are buttons to perform an explain plan and to test the query.

The Query page has three modes of operation:

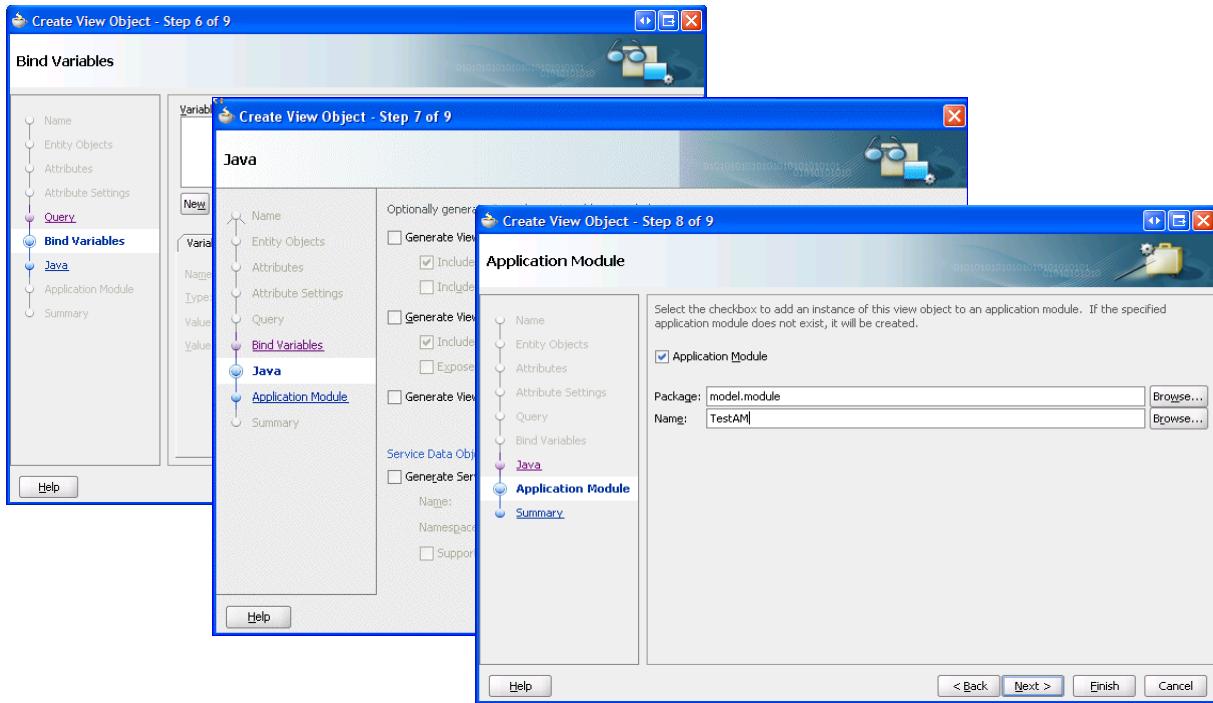
- **Normal mode** is easier and more foolproof, because most of the query is taken care of for you. The SELECT and FROM portions of the SQL statement are automatically created, based on the entity objects that the view object is based on, and on view attributes with the “Selected in Query” setting. You can add the WHERE and ORDERBY portions of the query as needed.
- **Expert mode** is useful if:
 - You want complete control over the SQL query, such as the SELECT and FROM portions. In this mode, you are responsible for ensuring that the mapping of SQL result columns to entity attributes is correct in the Attribute Mappings page. You need to remap these items if you deleted, added, or switched positions of columns in your custom SQL query, because the mappings might no longer be correct.

Creating Updatable View Objects (continued)

- You want to remove a SQL query from the view object. You might want to remove the SQL query if you are only inserting data, and want to save the startup time association with the initial query; or you want to define temporary collections of data using the same consistent business components interface and easily bind data to the user interface.
- **Declarative mode:** This mode is valid only for entity-based view objects. It requires no knowledge of SQL and defers SQL generation until the run-time execution of the view object. The SELECT and FROM lists are generated at run time based entirely on the entity object attributes that are exposed by a databound component in the user interface. You can provide the functionality of WHERE and ORDERBY clauses, if needed, by defining view criteria and sort criteria. These criteria are converted at run time to their corresponding SQL clauses.

Warning: If you switch modes, any SQL that you have defined in a previous mode is lost.

Creating Updatable View Objects: Additional Settings



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

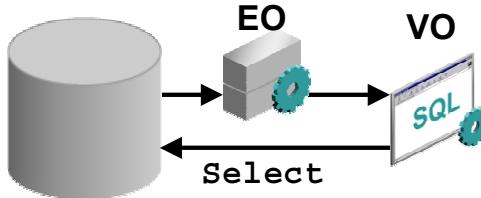
Creating Updatable View Objects (continued)

The next three pages of the wizard enable you to create bind variables, generate Java files, and create or use an existing application module, the same as you could do for read-only view objects.

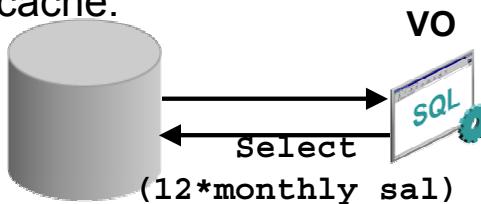
Finally, a summary page shows the options that have been selected. Clicking Finish creates the view object.

Interaction Between Views and Entities: Retrieving Data

- The view object queries the database directly.
- Data retrieved by the query is saved to the entity object's cache.



- Nonpersistent attributes are stored and retrieved from the view object's cache.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Interaction Between Views and Entities: Retrieving Data

A view object typically gets its data by using a SQL query, and a view object is usually bound to an entity object. This slide should help to explain these two apparently conflicting pieces of information. View objects and entity objects work together when a view object retrieves data:

- The view object queries the database, ensuring that its data is current.
- Data retrieved by the query is saved to the entity object's cache.

Calculated Attributes

A calculated attribute is based only on a database query, not on an entity object's attribute.

Calculated attributes get their data directly from a column expression in the SQL query; the data is stored in the view object's cache.

Entity Objects

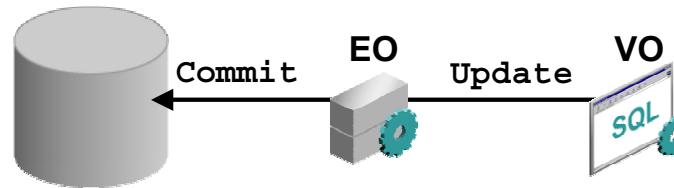
The data is passed to the entity object to validate business rules stored in the object. The view object determines what data is retrieved. The entity object guarantees that the data follows the defined business rules.

Synchronization

Values in view object rows and entity object rows remain coordinated.

Interaction Between Views and Entities: Updating Data

- The view object updates the entity object's cache.
- The entity object commits to the database.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Interaction Between Views and Entities: Updating Data

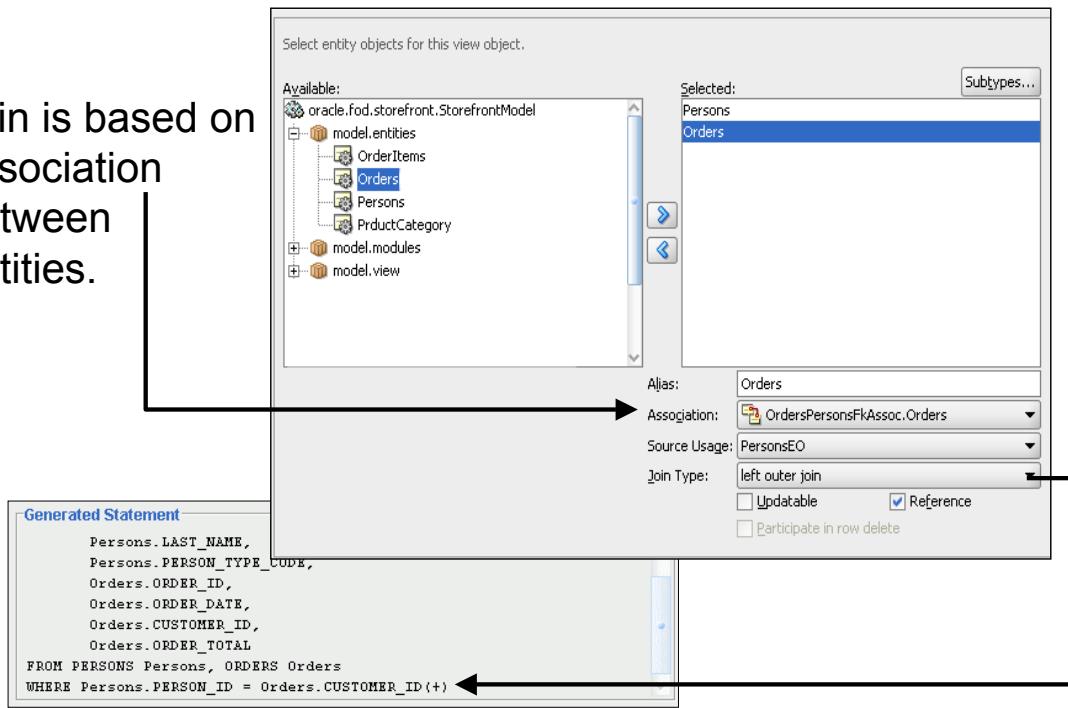
When you update data in the database, the process described in the previous slide is not quite reversed:

- The view updates the entity object's cache.
- When the transaction is committed, the entity object updates the database.

As stated on the preceding page, the entity object guarantees the validity of the data that is being committed to the database.

Creating a Join View Object

Join is based on association between entities.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating a Join View Object

A join view object is one that is based upon multiple entity objects that are related by one or more associations. When you create a join view object, you can select from the following join types:

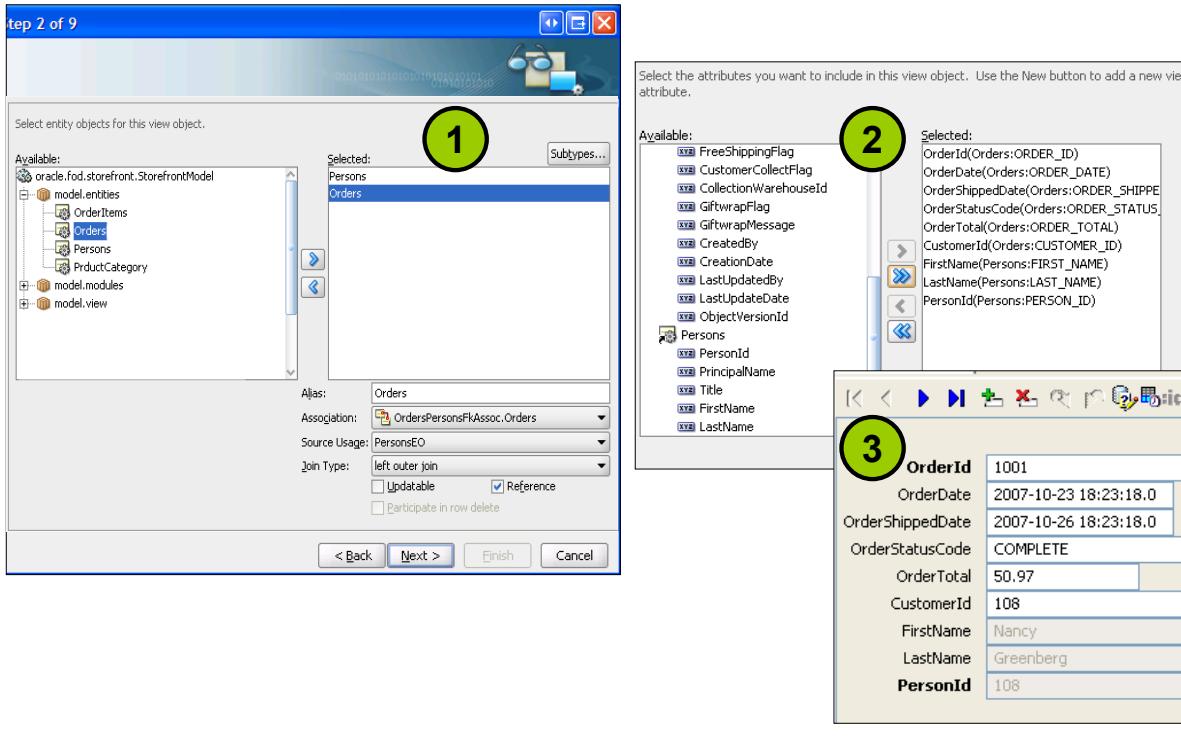
- **Inner join (default):** Rows that meet query criteria are returned only if related rows exist in both entities. For example, only Person rows with at least one order and only Orders rows that are associated with a person are returned.
- **Left outer join:** All rows that meet the criteria from the first entity object are returned by the query, regardless of whether there are rows in the second entity. For example, all Person rows that meet query criteria are returned, regardless of whether they have an associated order.
- **Right outer join:** All rows that meet the criteria from the second entity object are returned by the query, regardless of whether there are rows in the first entity. For example, all Order rows that meet query criteria are returned, regardless of whether they have an associated person.

Shaping the Data

It is important to realize that the type of join view object selected will determine how the data will be shaped, specifically as to how it will be used in either the business services or the UI.

Therefore, different types of view objects can be used to represent different facets of the data.

Including Reference Entities in Join View Objects



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Including Reference Entities in Join View Objects

It is extremely common in business applications to supplement information from a primary business domain object with secondary reference information to help the end user understand what foreign key attributes represent. For example, when displaying employees, rather than displaying the department number, it may be more meaningful for users if the department name is displayed. The department name is in a separate entity from the employees' entity.

When you create a join view object, the additional entity is most commonly used as a nonupdatable reference entity. To include a reference entity, when you create the view object (or when editing it later), perform the following steps:

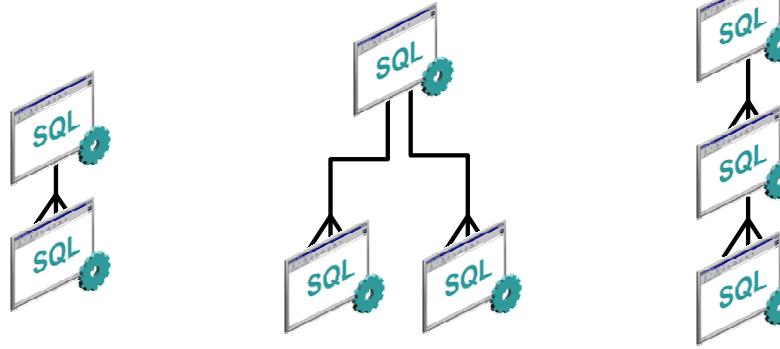
1. Add the secondary entity to the view object as a Reference entity, choosing the association and join type to use.
2. Add the referenced attributes to the selected attributes for the view object.
3. When the application module is run, the referenced attributes are displayed in read-only mode.

You can include multiple referenced entities in the same view object. In fact, in most cases all but one of the entities in a view object are reference entities.

The example in the slide shows the Orders entity as the primary entity for the view object. The FirstName, LastName, and PersonId attributes are added from the reference entity, Persons, to be displayed as read-only attributes.

Creating Master-Detail Relationships with View Objects

Different types of master-detail relationships are supported.



Master to Single Detail

Master to Multiple Details

Cascading Master-Detail to Any Depth

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating Master-Detail Relationships with View Objects

You can use view links to create master to multiple detail and cascading master-detail relationships between view objects. To create view objects with these characteristics, you need to create view links.

Linking View Objects

Select each pair of source and destination view object attributes that define the view link, then click Add.

Cardinality: 0..1 to *

Select Source Attribute:

- CustomerVO
 - FirstName
 - LastName
 - OrdersPersonsFkAssoc
 - PersonId**
 - PersonToAddressFkAs
 - PersonTypeCode
 - PrimaryAddressId
 - PrincipalName
 - SupplierId

Select Destination Attribute:

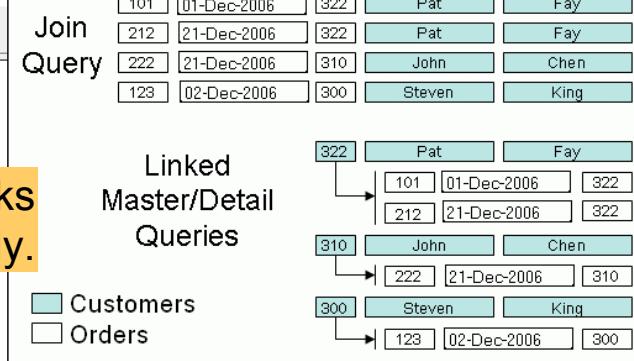
- OrderVO
 - CustomerId**
 - FirstName
 - LastName
 - OrderDate
 - OrderId
 - OrdersAddressesFkAssoc
 - OrderShippedDate
 - OrderPersonsFkAssoc
 - OrderStatusCode

Add Remove

Source Attribute(s)	Destination Attribute(s)
CustomerVO.PersonId	OrderVO.CustomerId

Use the Create View Link Wizard to create a view link between CustomerVO and OrderVO view objects.

Joined queries and view links display information differently.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Linking View Objects

You have learned to use associations to define a relationship between entity objects. You have also learned how to join multiple tables in a view object query, producing a single view object that contains attributes from multiple tables. View links enable you to join multiple view objects in a master-detail hierarchy. When your needs call for showing the user a set of master rows, and for each master row a set of coordinated detail rows, then you can create view links to define how you want the master and detail view objects to relate. Note, however, that a query of the detail view object via the master only occurs as long as the view link is included as part of the application module.

To define a view link, you can first create a view object that queries the master table and another that queries the detail table. Then you can create a new view link, selecting the attributes of the source (master) and destination (detail) view objects to link on and clicking Add to create the source/destination pair. If multiple attribute pairs are required to define the link between master and detail, you can repeat these steps to add additional source/destination attribute pairs.

Comparing Join View Queries with View Links

Join view queries:

- Contain all attributes from main and referenced entities in a single row
- Use a WHERE clause to relate a key from the main table with a foreign key in referenced tables

View link queries:

- Use a bind variable to relate detail records to key for selected master record
- Append WHERE clause with bind variable to the base SQL query for the source or destination view object



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Comparing Join View Queries with View Links

A join view query for the example in the previous slide may be as follows:

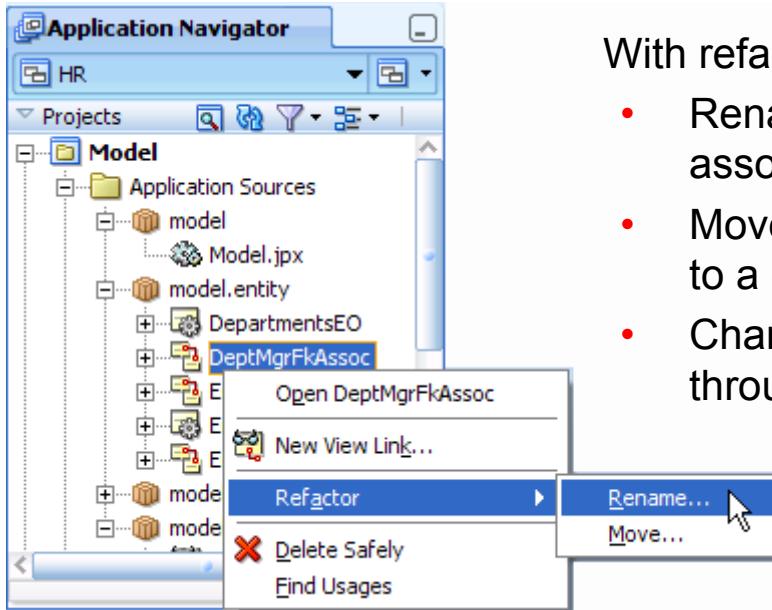
```
SELECT Orders.ORDER_ID,  
       Orders.ORDER_DATE,  
       Persons.PERSON_ID,  
       Persons.FIRST_NAME,  
       Persons.LAST_NAME  
  FROM PERSONS Persons, ORDERS Orders  
 WHERE Persons.PERSON_ID = Orders.CUSTOMER_ID
```

A query for a view link uses a bind variable in a WHERE clause that is appended to the base query for the source or destination view object. In the example in the previous slide, the WHERE clause that is appended would be something like the following:

```
:Bind_PersonId = Orders.CUSTOMER_ID
```

This would query all the orders for the current customer, displaying orders and customers in a master-detail fashion.

Refactoring Objects



With refactoring you can:

- Rename objects, such as associations and view links
- Move objects or packages to a different package
- Change all references throughout the application

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Refactoring Objects

You can use refactoring to rename or move an object:

- **Renaming:** When you use a wizard to create entity objects and view objects, associations and view links are created automatically. They are by default given names that are based on the name of the foreign key in the database. It is often desirable to use a more meaningful phrase to rename them. To do so, you right-click the association or view link in the Application Navigator and from the context menu, select Refactor > Rename. A refactoring dialog box appears to enable you to rename the association or view link.
- **Moving:** When you create an object, you are given the opportunity to specify which package to place it in. If you later want to change the package where an object resides, you can right-click the object and select Refactor > Move. A dialog box opens to enable you to specify a different package; this package gets created if it does not already exist.

When you use refactoring, all references to the object are modified throughout the application.

Summary

In this lesson, you should have learned how to:

- Describe the characteristics of an ADF BC view object
- Create ADF BC view objects that can be used for data queries in a Web application
- Modify SQL statements in view objects
- Explain how entity objects relate to database tables
- Describe the persistence mechanism of entity objects
- Create entity objects from database tables
- Create associations between entity objects
- Create updatable view objects based on entity objects
- Link view objects to one another
- Refactor to move and rename associations and links



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 4 Overview: Creating Entity Objects and View Objects

This practice covers the following topics:

- Creating Read-Only View Objects
- Creating Multiple Read-Only View Objects at Once
- Creating Associations
- Creating Updatable View Objects
- Refactoring Associations and Links



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 4 Overview: Creating Entity Objects and View Objects

In this practice, you begin to create components that are more customized for your application. First you determine the LOVs that your application needs, and you create read-only view objects to support those LOVs. Next you create entity objects for tables that the application will update, and you also create updatable view objects based on these. You also create and refactor associations and view links. For now, you place all of the view objects and links in the same application module so that you can test them; in the next set of practices you create custom application modules that are designed for your application.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Osi S.R.L. use only

Exposing Data

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Explain the role of application modules
- Describe the characteristics of application modules
- Create an application module
- Explain how application modules can manage:
 - Business components transactions
 - Application state
- Explain the role of the ADF Model



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

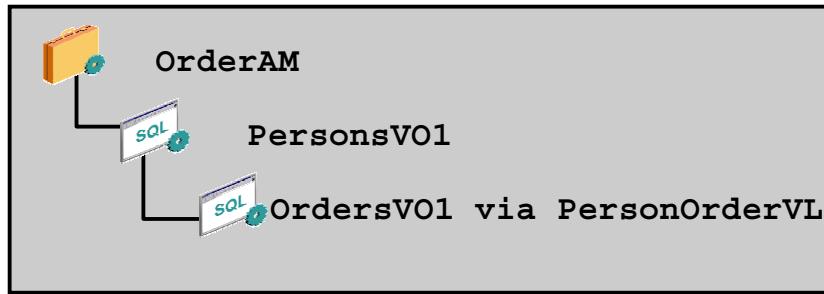
Lesson Aim

In this lesson, you learn how to expose ADF BC data services to an application. You learn that the application module enables the data services to be used on a Web page, and you learn about how ADF Model abstracts the data for binding to user interface components. You create application modules and test them with the Business Components Tester.

Oracle ADF Application Module (AM)

An Oracle ADF Application Module:

- Represents a logical unit of work related to an end-user task
- Encapsulates the active data model and business service methods for that task



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Oracle ADF Application Module (AM)

Oracle ADF application modules are business components that represent particular application tasks. The application module provides a data model for the application task by acting as a sort of “wrapper” for the view object and view link instances required for the task. Although conceptually an application module is a wrapper for the entities, views, associations, and view links in an application task, its data model contains only views and view links. The underlying entities and associations are included by implication. The application module also contains services that help the client accomplish the task. For example, an application module can represent and assist with tasks such as:

- Updating customer information
- Creating a new order
- Processing salary increases

Restricted and Unrestricted Views

In the application module shown in the slide, `PersonsVO1` is an unrestricted view and `OrdersVO1` is a restricted view. When you run the application module, `PersonsVO1` can display all rows of person data, but `OrdersVO1` can display only the order data associated with the current person. When you add a view to your application module’s data model, you can add it as an unrestricted view or as a restricted view.

Characteristics of an Application Module

- Represents the data model that the client uses and has one connection to the database
- Provides transactional context for the application
- Enables you to gather data customized to a client interface so that data can be retrieved in one network round-trip
- Can contain other application modules, called nested application modules
- Keeps track of all changes that affect data in the database
- Provides remotely accessible methods to implement application module behavior
- Is deployable in multiple configurations
- Can be easily reused in business logic tiers of other applications



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

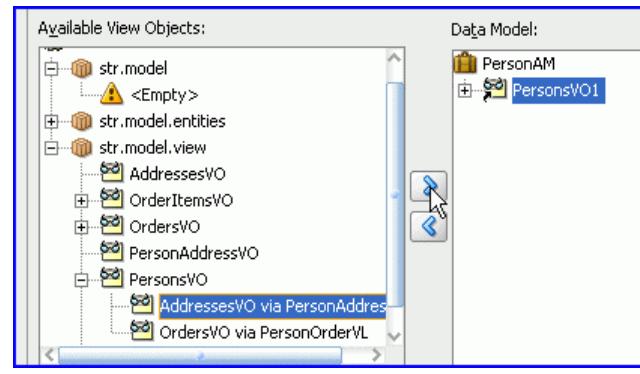
Characteristics of an Application Module

The application module acts as a wrapper for the views and entities in your business model. It also handles all database transactions and performs tasks that are specific to the application. An application module performs a specific application task—for example, logging service requests or processing user information. An application module has the following main characteristics:

- It represents the data model that your client uses and has one connection to the database.
- It enables you to gather data customized to a client interface (such as a form), so that data can be retrieved in one network round-trip instead of multiple trips.
- It can contain other application modules, called nested application modules, enabling you to separately develop, test, and then reuse the application module functionality.
- It keeps track of all changes that affect data in the database and defines the transactional model.
- It provides remotely accessible methods, which implement the application module behavior.
- You can deploy the same application module in multiple configurations.
- As discrete units, application modules are easily reused in the business logic tiers of other applications.

Creating an Application Module

1. In the New Gallery Business Tier, select ADF Business Components > Application Module to invoke the Create Application Module Wizard.
2. Specify a name for the application module and identify the package where it should belong.
3. Define the data model for the application module by selecting view objects from a tree of available view objects.



ORACLE

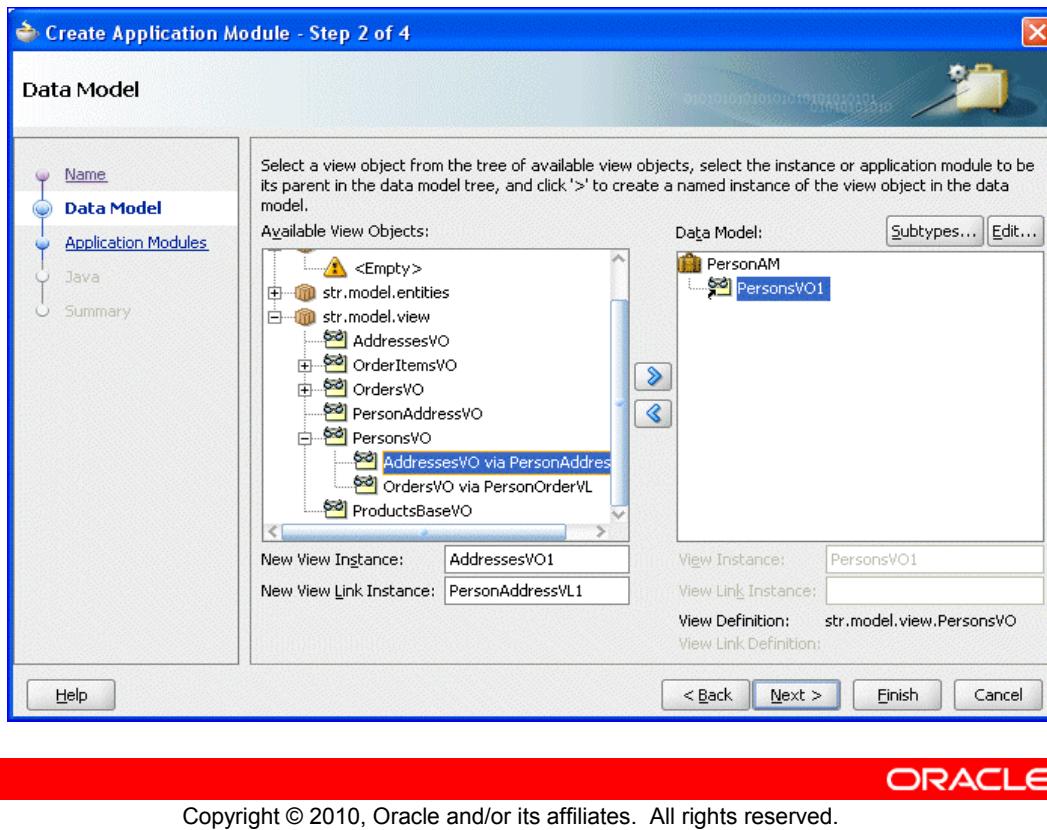
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating an Application Module

When you create an application module, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, given an application module named PersonAM in the `oracle.model` package, the XML file created is `./oracle/model/PersonAM.xml` under the project's source path. This XML file contains the information needed at run time to re-create the view object instances in the application module's data model. You can view the contents of this XML file by selecting the view object in the Application Navigator and looking in the corresponding Sources folder in the Structure Window. Double-clicking the name of the XML file opens it in an editor so that you can inspect it.

After you have created your application module, you can edit any of its settings by using the Application Module Editor. To launch the editor, select Edit from the context menu in the Application Navigator, or double-click the application module.

Defining the Data Model for the Application Module



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

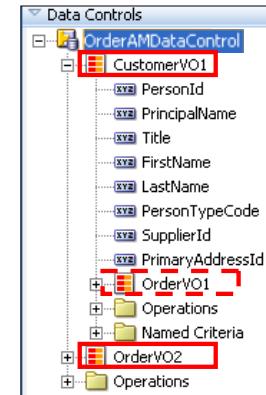
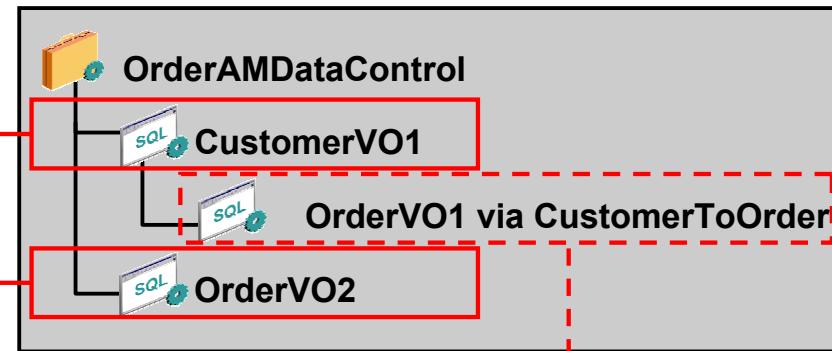
Defining the Data Model for the Application Module

The application module contains an active data model of the view objects and view link instances required for the task that the application module performs. When you create the application module, you need to define all the view objects that are part of the application module. All view objects contained within the current package are available to be included. Select the view objects required and click the right arrow to include them in the data model. To remove a view object from the data model, select it in the Data Model section and click the left arrow.

By default, the wizard creates instance names for the data model components based on their corresponding view object names. You might want to change the instance name based on its function or hierarchical placement. In the example in the slide, you might rename the nested instance to "AddressesPerPerson," because the wizard generates a less clear instance name, "AddressesVO1 via PersonAddressVL1" by default.

Using Master-Detail View Objects in Application Modules

First level: Independent VO instances



Subordinate levels: VO instances accessed via a view link display both master and detail.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Master-Detail View Objects in Application Modules

When you run an application module in the Business Components Browser, or when you use an application module's data control in a user interface, you need to be aware of the levels of VO instances that are shown in the BC Browser or the Data Controls panel:

- The first level, displayed just under the application module in the hierarchy, shows independent instances of view objects.
- Any subordinate levels show instances of view objects that are accessed via a view link. These are child VO instances of the independent VO instance on the first level, and therefore contain only rows related to the current row of the master VO. There may be multiple detail levels, each showing related records of its direct master VO.

In the example in the slide, OrderVO1 contains only the orders belonging to the current row in CustomerVO1. OrderVO2 contains all orders, unless restricted by a WHERE clause or view criteria.

Determining the Size of an Application Module

Is it better to have one big application module, or several little ones?

- An application module is a logical unit of work.
- Let use cases drive application module decisions:
 - Can be grouped by domain business objects involved
 - Grouped by the user-oriented view of business data required
- Consider the possibility of reuse of the application module.
- Consider service or transaction flow.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Sizing an Application Module

An application module represents the logical data model to accomplish a specific user task.

In the early analysis phases of application development, architects and designers often use UML use-case techniques to iteratively refine a high-level description of the different kinds of business functions that the system needs to support. Each high-level, end-user use case identified during the design phase should generate the following considerations:

- **The domain business objects involved:** What core business data is relevant to the use case?
- **The user-oriented view of business data required:** What subset of columns, what filtered set of rows, sorted in what way, grouped in what way is needed to support the use case?

Business Components Transactions

- Application modules:
 - Handle transaction and concurrency support
 - Use a single database connection
 - Provide transaction context for updates, deletes, and inserts for all view objects in the application module, so all are committed or rolled back at once
- For nested application modules, the outermost application module provides the transaction context for the others.
- No coding is required unless you want to modify the default behavior.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Business Components Transactions

An application module is a transactional container for a logical unit of work, so any updates, inserts, or deletes for all view objects in the application module are committed or rolled back at once. The outermost, or root, application module provides the transaction context for all nested application modules.

At run time, the application module delegates transaction management to a companion `Transaction` object, which provides an entity cache as a work area to hold entity rows involved in the current user's transaction. Each entity cache contains rows of a single entity type, so a transaction involving two or more entity objects holds the working copies of those entity rows in separate caches. The entity cache is also the place where new entity rows wait to be saved.

When an entity row is created, modified, or removed, it is automatically enrolled in the transaction's list of pending changes. When you call `commit()` on the `Transaction` object, it processes its pending changes list, validating new or modified entity rows that might still be invalid. When the entity rows in the pending list are all valid, `Transaction` issues a database `SAVEPOINT` and coordinates saving the entity rows to the database. If all goes successfully, it issues the final database `COMMIT` statement. If anything fails, `Transaction` performs a `ROLLBACK TO SAVEPOINT` to allow the user to fix the error and try again.

Business Components Transactions (continued)

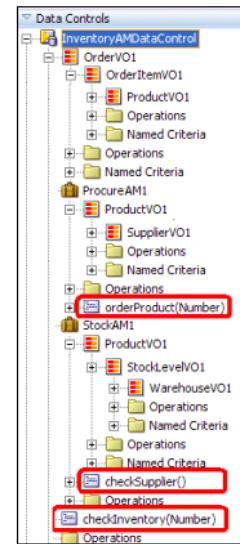
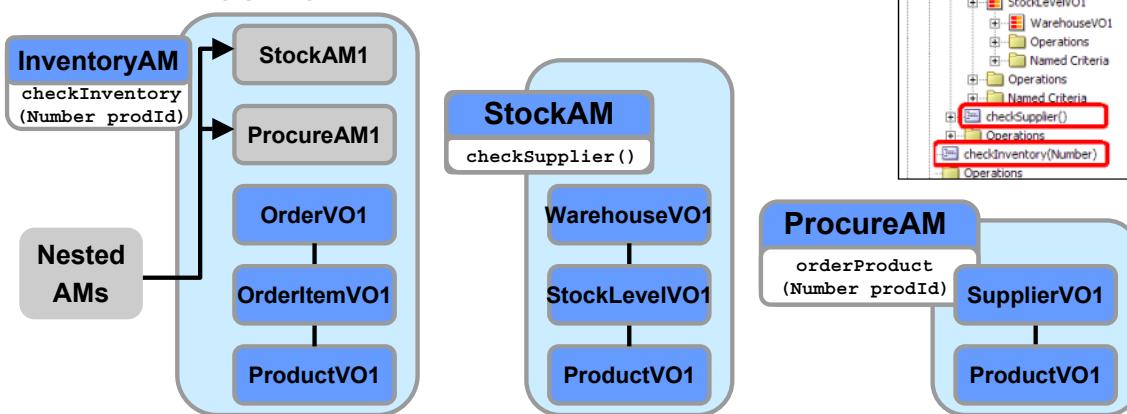
When a root application module contains other nested application modules, they all participate in the root application module's transaction and share the same database connection and a single set of entity caches. This sharing is handled for you automatically by the root application module and its `Transaction` object. You should keep this transaction context in mind when designing the application. If you want separate transactions, do not nest the application modules.

For example, you may have an `InventoryManagerAM` application module that has methods to add or subtract from inventory. It would make sense to nest this application module inside a customer-facing `ShoppingAM`. When the customer submits an order, you could subtract the ordered items from the inventory as a part of the same transaction. If the commit fails, the order does not get submitted, and inventory changes are rolled back as well. It would not make sense, however, for you to nest the inventory management application module inside one that manages human resources. HR transactions, such as hiring or salary changes, should be completely separate transactions from inventory management.

Using Nested Application Modules

With nested AMs:

- Root AM provides transaction context, database connection, and a single set of entity caches
- You aggregate VOs and service methods



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Nested Application Modules

Application modules support the ability to create software components that mimic the modularity of your use cases, for which your higher-level functions might reuse a subfunction that is common to several business workflows. You can implement this modularity by defining composite application modules that you assemble using instances of other application modules. This task is referred to as application module nesting. That is, an application module can logically contain one or more other application modules, as well as view objects. The outermost containing application module is referred to as the root application module.

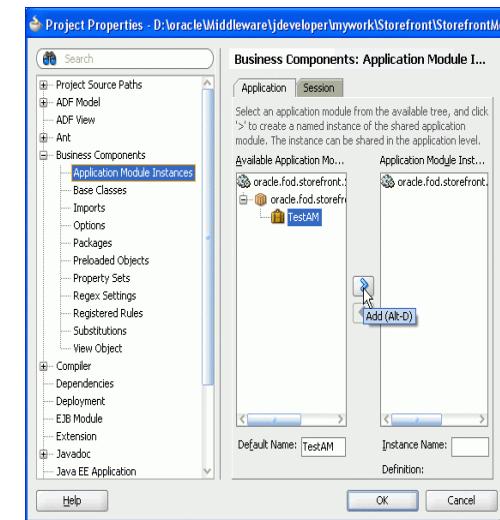
Any application module can be used as a root application module, but it would usually be one that maps to a more complex use case. When a root application module contains other nested application modules, they all participate in the root application module's transaction and share the same database connection and a single set of entity caches. This sharing is handled for you automatically by the root application module and its `Transaction` object.

When you nest an instance of one application module inside another, you aggregate not only the view objects in its data model, but also any service methods it defines. This feature of nesting, or reusing, an instance of one application module inside of another is one of the most powerful design aspects for implementing larger-scale, real-world application systems.

The example in the slide shows that the **InventoryAM** application module contains nested application module instances of **StockAM** and **ProcureAM**, in addition to view object instances.

Shared Application Modules

- Shared AMs provide a reusable data service.
- They allow requests from multiple sessions to share a single application module instance.
- Any user session is able to access the same view instances contained in the shared AM.
- You specify sharing in the Project Properties dialog box.
- Shared AMs can be “application scoped” or “session scoped.”



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Shared Application Modules

Web applications often utilize data that is required across sessions and does not change much. An example of this type of static data might be displayed in a lookup list. Accessing static data can incur an unnecessary overhead when static data caches are repopulated from the database for each application session on each request. In order to optimize performance, a common practice when working with ADF Business Components is to cache the static data for reuse across sessions and requests.

Shared application modules allow requests from multiple sessions to share a single application module instance which is managed by the application pool for the lifetime of the Web server virtual machine. In the case of *application-level* sharing, any user session will be able to access the same view instances contained in the shared application module. In contrast, the life cycle of the *session-level* shared application module extends to a user session. In this case, view instances of the application module will be accessible by other components in the same user session.

You specify that an application module is to be shared in the Project Properties dialog box:

1. Double-click the name of the project that contains the application module, to invoke the Project Properties dialog box.
2. In the panel on the left, expand the Business Components node and select Application Module Instances.

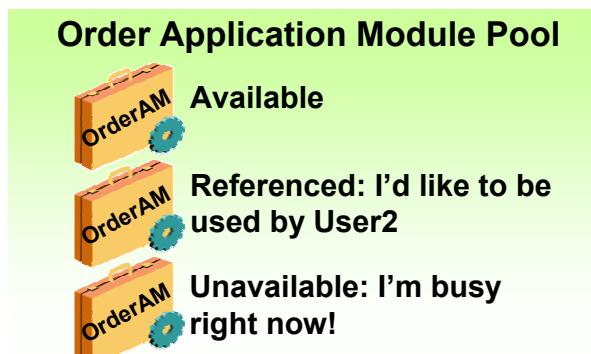
Shared Application Modules (continued)

3. Click the Application or Session tab to define whether the instance is to be application scoped or session scoped.
4. Select the AM to be shared in the Available tree on the left and use the arrow to shuttle it to the Application Module Instances pane on the right.
5. You can change the name of the instance if required. Click OK to create the instance.

Application Module Pooling

Application module pooling:

- Enables users to share application modules
- Manages application state
- Provides the same instance or one with an identical state when requested by an application with managed state



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Application Module Pooling

An application module pool is a collection of application module instances of the same type, such as an Orders application module or a Human Resources application module. This pool of application module instances is shared by multiple browser clients. The amount of time between submitting Web pages enables a smaller number of application module components to serve a larger number of active users. This reduces memory usage and improves performance.

At any one time, the pool may contain application module instances that are partitioned into three groups, based on their state. When the processing of a current HTTP request completes, the application module instance is checked back into the pool. If the AM instance has managed state, the pool keeps track that the AM is referenced by that particular session.

An application module instance in the pool may be:

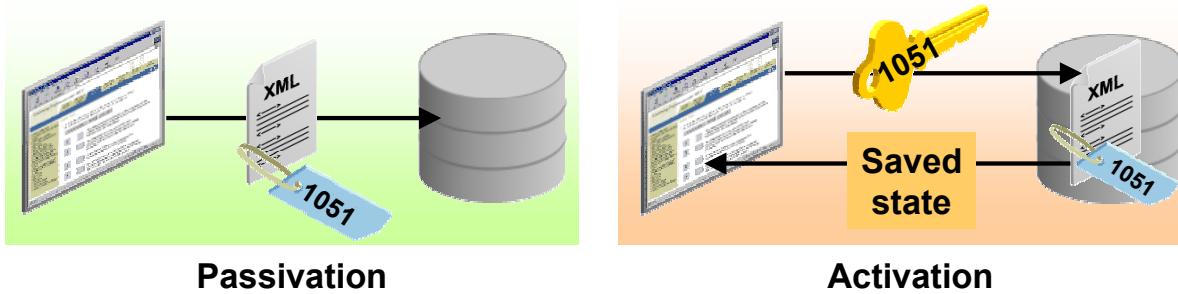
- Unconditionally available for use
- Available, but referenced for preferred reuse by a certain active session because that would be more efficient due to the managed state of the application module
- Unavailable because it is being used at that very moment by a Web container thread

You can configure application module pool behavior, sizing, and cleanup behavior. For more information about this, refer to the *Fusion Developer's Guide for ADF* in online Help.

Note: Pools are created only for root application modules, not for nested ones that users access indirectly through a root application module.

Managing Application State

- AM passivation saves transaction state in an XML document stored in database.
- AM activation retrieves saved transaction state.
- Passivation and activation are performed automatically when needed.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

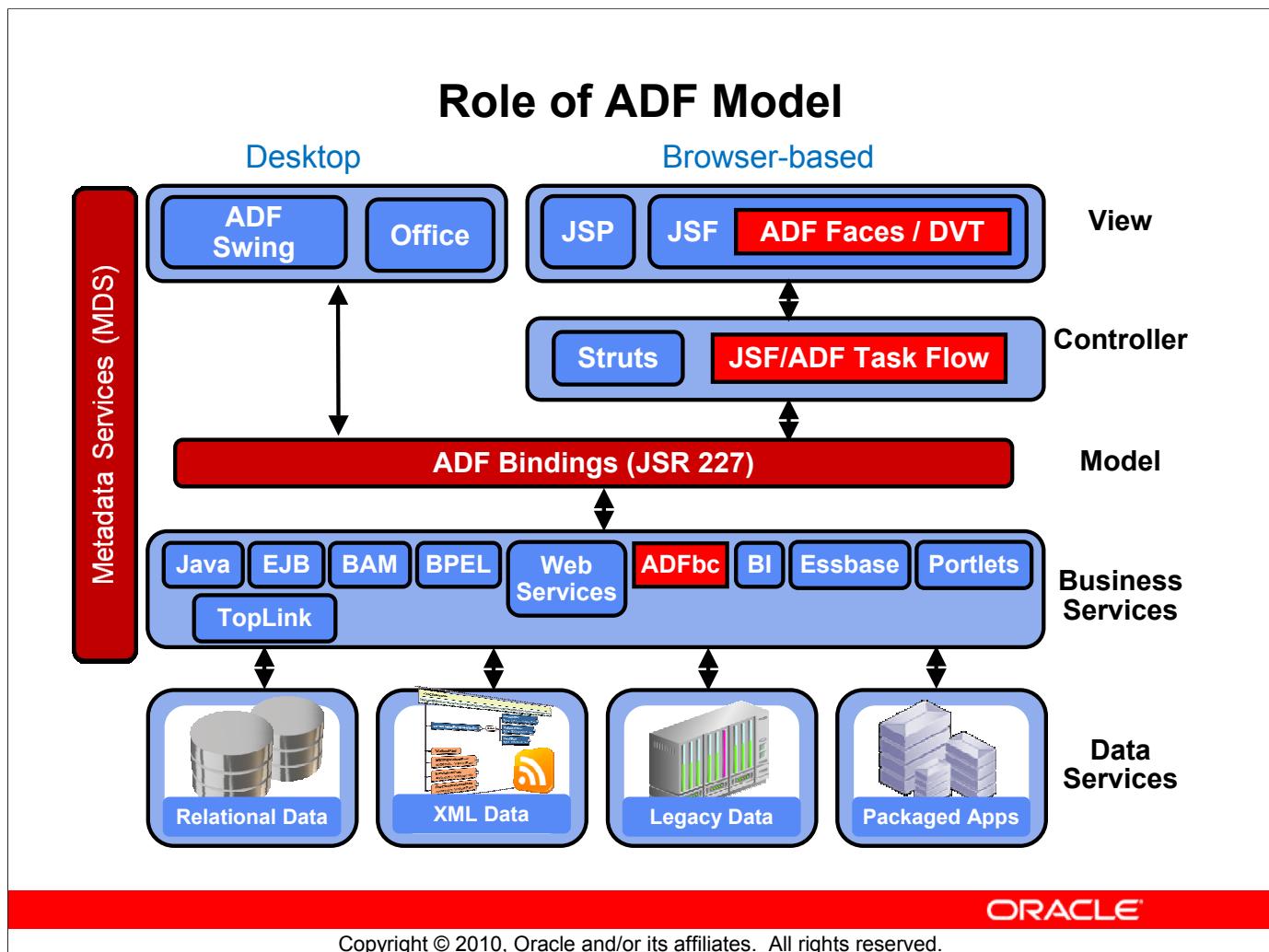
Managing Application State

HTTP is a stateless protocol. ADF model supports both stateless applications as well as those involving a unit of work that may span multiple pages.

An application module supports passivating its pending transaction state to an XML document, which is stored in the database in a single, generic table, keyed by a unique passivation snapshot ID. It also supports the reverse operation of activating pending transaction state from one of these saved XML “snapshots.”

You can demonstrate passivation in the Business Components Browser by making changes without saving them, and then selecting File > Save Transaction State. A dialog box displays the transaction key ID. Then close and reopen the BC Browser; it displays the original data without your changes. Select File > Restore Transaction State to restore the pending changes, which you can then commit to the database if desired.

Passivation and activation are performed automatically in an ADF BC application by the application module pool. This automatic state management provides the simplicity of a stateful programming model that is nearly as scalable as a completely stateless application.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

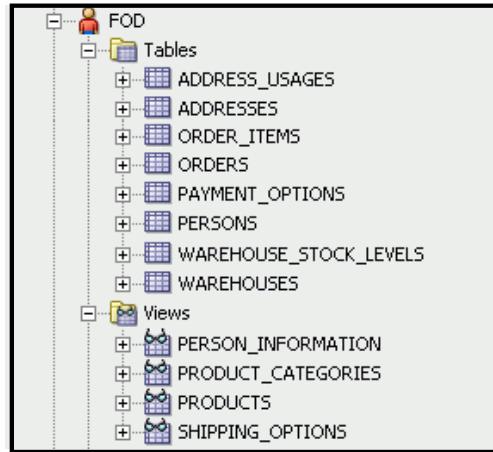
ADF Model

Developers need a standard way to bind services to Java UIs and to simplify the use of different service implementations in the model. As a developer, you need to know and understand the specifics of the technology that is used to persist data in your application. It is time consuming to write boilerplate code to create interactions between UI components and services.

ADF Model provides a generalized approach. ADF Model offers a layer of abstraction, describing the middle-tier business services to the view, so that the developer is insulated from the specifics of the underlying service implementation. Instead of binding the view to the business services, you bind the view to the model, and the model is bound to the business service by a data control and exposed to the developer through a binding. In short, ADF Model provides a common API to bind data from a business service to a UI. Out-of-the-box, ADF Model supports data controls for services based on Java classes, EJB session beans, Web services, and ADF Business Components.

Data controls are an ADF Model layer that represents the back-end business services. Each time you create an ADF BC application module, a data control is automatically created that contains all the view object instances and client methods that are exposed in the application module. Data control is what enables you to bind UI components to the back-end data.

Describing the Course Application: Database Objects



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Describing the Course Application: Database Objects

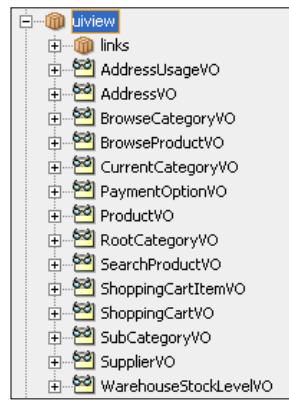
In this course, you build a shopping portal similar to any that you see on the Internet.

The main entity objects that are used in the course application are based on the following database tables and views:

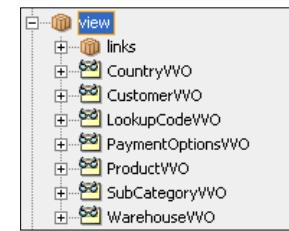
- Addresses
- Address_Usages
- Orders
- Order_Items
- Payment_Options
- Person_Information (customers)
- Persons (customers)
- Products
- Product_Categories
- Warehouse_Stock_Levels
- Warehouses

Describing the Course Application: View Objects

View objects exposed in application modules



View objects used for LOVs in the UI



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Describing the Course Application: View Objects

The view objects that are exposed to the UI are in the `uiview` package. They include:

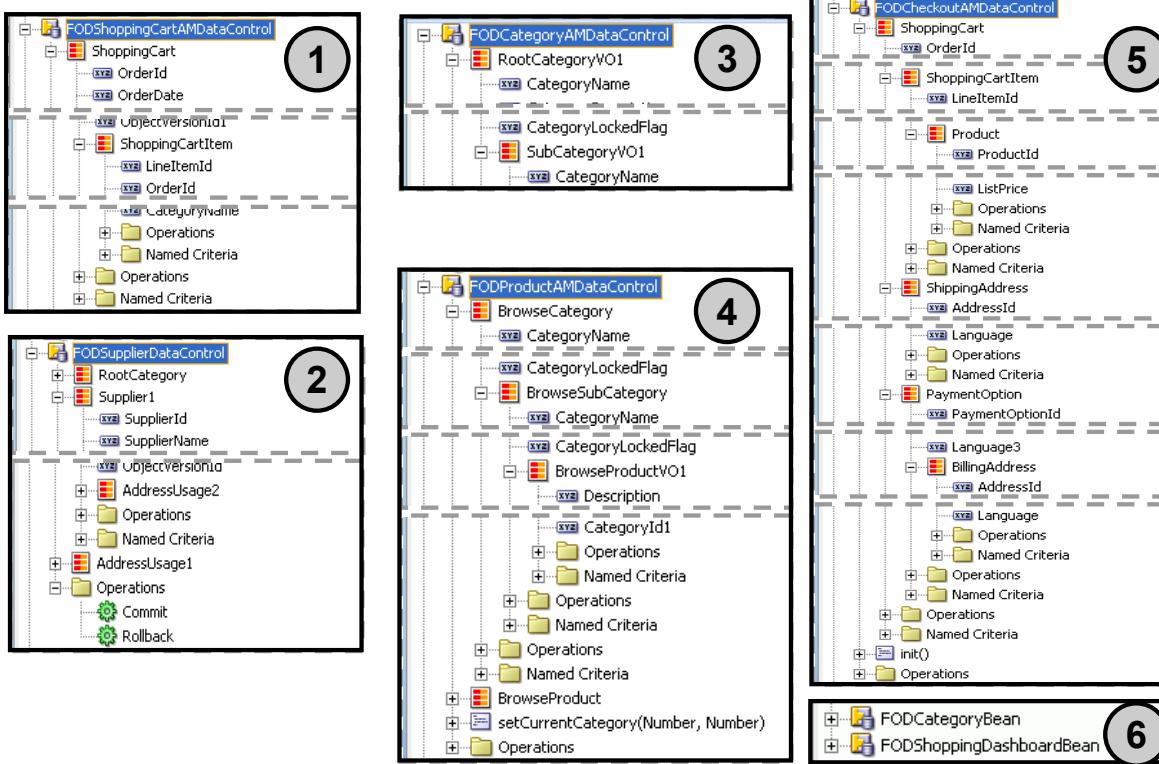
- `AddressUsageVO`
- `AddressVO`
- `BrowseCategoryVO`: Is based on `Product_Categories`
- `BrowseProductVO`: Accesses products of a specified category
- `CurrentCategoryVO`: Category IDs and names of specified category and its parent category
- `PaymentOptionVO`
- `ProductVO`
- `RootCategoryVO`: Accesses product categories whose parent category ID is null; in other words, the root categories
- `SearchProductVO`: Is based on `ProductCategoryVO` with reference to `ProductEO`
- `ShoppingCartItemVO`: Is based on `OrderItemEO`, with references to `ProductEO` (via view link to `OrderItemEO`) and `ProductCategoryEO` (via view link to `ProductEO`)
- `ShoppingCartVO`: Is based on `OrderEO` with a reference to `PersonEO` (related customers)
- `SubcategoryVO`: Is related via a view link to `RootCategoryVO`; accesses subcategories of specified root category
- `SupplierVO`
- `WarehouseStockLevelVO`: Is based on `WarehouseStockLevelEO`, with reference to `WarehouseEO` (related warehouses)

Describing the Course Application: View Objects (continued)

In addition, there are several view objects that support lists of values in the UI. These VOs are in a separate package called view, and their suffix is VVO instead of VO. These read-only view objects include:

- CountryVVO
- CustomerVVO
- LookupCodeVVO
- PaymentOptionsVVO
- ProductVVO
- SubCategoryVVO
- WarehouseVVO

Describing the Course Application: Data Controls



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Describing the Course Application: Data Controls

The main application modules are:

1. **FODShoppingCartAM:** Shopping cart (order) > Shopping Cart Items (order items)
2. **FODSupplierAM:** Suppliers > AddressUsages
3. **FODCategoryAM:** Root categories > subcategories
4. **FODProductAM has two views on the main level:**
 - Browse categories > browse subcategories > browse products
 - BrowseProduct
5. **FODCheckoutAM:** Shopping cart, which has three views on the second level:
 - ShoppingCartItem > Product
 - ShippingAddress (from AddressVO)
 - PaymentOption > BillingAddress (from AddressVO)

Data controls are automatically created when you create an application module, and therefore there is one data control per application module.

6. **In addition, there is a data control for each of two beans:**
 - **FODCategoryBean:** Is used with a tree to determine whether a clicked node is a root category or a subcategory
 - **FODShoppingDashboardBean:** Stores some information that is used to refresh a region in the UI

Summary

In this lesson, you should have learned how to:

- Explain the role of application modules
- Describe the characteristics of application modules
- Create an application module
- Explain how application modules can manage:
 - Business components transactions
 - Application state
- Explain the role of the ADF Model



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 5 Overview: Defining Application Modules

This practice covers the following topics:

- Creating an Application Module to Display Categories
- Creating an Application Module to Display the Shopping Cart



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 5 Overview: Defining Application Modules

In this practice, you take some of the view objects you have created and add them to application modules. You create two application modules in this practice. The first application module displays categories and subcategories. The second application module is for displaying and managing a customer's shopping cart. After you have created the application modules, you test them in the Business Components Browser.

Declaratively Customizing Data Services



ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Describe how to declaratively change data behavior
- Declaratively modify view objects, entity objects, and application modules
- Create view accessors
- Create LOVs



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson teaches you to customize business components to change the data behavior for the application. You learn to make various declarative modifications to the data model. You then test the modified functionality with the Business Components Tester.

Using Groovy

Groovy:

- Is a Java-like scripting language that is dynamically compiled and evaluated at run time
- Enables you to use declarative expressions, instead of writing Java code, for the following types of values in the ADF BC data model:
 - Bind variables
 - Calculated attributes
 - Attribute default values
 - View criteria
 - View accessor bind variables
 - Validation
 - Validation message token binding



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Groovy

Groovy is an agile, dynamic language for the Java platform, defined as JSR 241. It has many features that were inspired by languages such as Python, Ruby, and Smalltalk, making them available to Java developers with a Java-like syntax. It interoperates seamlessly with any Java class, and can be compiled and interpreted without disturbing normal operations.

The latest release of JDeveloper provides integrated support for Groovy. You can use Groovy expressions for all sorts of declarative values, such as bind variables and attribute default values. You can use a Groovy script that returns true or false for declarative validation. You can also use Groovy expressions in error messages. You see examples of using Groovy expressions when these different topics are presented in this lesson and later lessons.

Groovy can simplify expressions and make them more concise in that it supports object access via dot-separated notation, so it supports syntax such as `Empno` instead of `getAttribute(EMPNO)`.

You can find out more about Groovy at <http://groovy.codehaus.org/> and <http://radio.weblogs.com/0118231/2007/05/22.html#a829>.

Note: Groovy is still a fairly new technology and is missing features such as code completion and debugging capabilities. This makes it difficult to use for large segments of code.

Using Groovy Syntax in ADF

Java Code	Equivalent Groovy script
<code>((Number)getAttribute("Sal")).multiply(new Number(0.10))</code>	<code>Sal * 0.10</code>
<code>((Date)getAttribute("PromotionDate")).compareTo((Date)getAttribute("HireDate")) > 0</code>	<code>PromotionDate > HireDate</code>

Reserved name `adf` gets objects from the framework:

- `adf.context`
- `adf.object`
- `adf.error`
- `adf.currentTimeMillis`
- `adf.currentDate`



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Groovy Syntax in ADF

The current object is passed into the script as the “this” object. So, to reference any attributes inside the current object, simply use the attribute name. For example, in an attribute-level or entity-level Script Expression validator, to refer to an attribute named “Salary,” the script may say simply reference `Salary`.

There is one top-level reserved name, `adf`, to get to objects that the framework makes available to the Groovy script. These objects include:

- `adf.context`: To reference the `ADFContext` object
- `adf.object`: To reference the object on which the expression is being applied
- `adf.error`: In validation rules, to access the error handler that allows the validation expression to generate exceptions (`adf.error.raise`) or warnings (`adf.error.warn`)
- `adf.currentTimeMillis`: To reference the current date and time
- `adf.currentDate`: To reference the current date with time truncated
- `adf.currentDateTime`: To reference the current date and time

All the other accessible member names come from the context in which the script is applied:

- **Bind Variable:** The context is the variable object itself. You can reference the `structureDef` property to access other information as well as the `viewObject` property to access the view object in which the bind variables participate.

Using Groovy Syntax in ADF (continued)

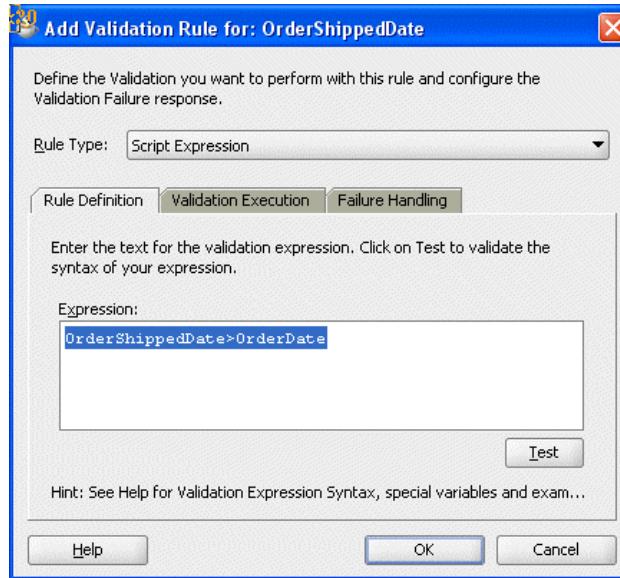
- **Transient Attribute:** The context is the current entity or view row. You can reference all attributes by name in the entity or view row in which it appears, as well as any public method on that entity or view row. To access methods on the current object, you must use the `adf.object` keyword to reference the current object like this:
`adf.object.yourMethodName()`. The `adf.object` keyword is equivalent to the `this` keyword in Java. Without it, in transient expressions, the method is assumed to exist on the dynamically compiled Groovy script object itself.
- **Expression Validation Rule:** The context is the validator object (`JboValidatorContext`) merged with the entity on which the validator is applied. You can reference keywords such as:
 - `newValue`, in an attribute-level validator, to access the attribute value being set
 - `oldValue`, in an attribute-level validator, to access the current value of the attribute being set
 - `source`, to access the entity on which the validator is applied in order to invoke methods on it (accessing simply by using attribute values in the entity does not require using the `source` keyword)

All Java methods, language constructs, and Groovy language constructs are available in the script. Some additional tips to keep in mind:

- You can use built-in aggregate functions on ADF RowSet objects by referencing the functions `sum()`, `count()`, `avg()`, `min()`, and `max()`. They accept a string argument, which is interpreted as a Groovy expression that gets evaluated in the context of each row in the set as the aggregate is being computed:
`rowSetAttr.sum("GroovyExpr")`
such as `employeesInDept.sum("Sal")`
or `employeesInDept.sum("Sal!=0?Sal:0 + Comm!=0?Comm:0")`
- Use the `return` keyword just like in Java to return a value, unless it is a one-line expression in which case the return is assumed to be the result of the expression itself (such as `Sal + Comm` or `Sal > 0`).
- Use the ternary operator to implement functionality that is similar to SQL's `NVL()` function—for example, `Sal + (Comm != null ? Comm : 0)`.
- Do not use `{ }` to surround the entire script. Groovy treats `{` as a beginning of a Closure object. (See Groovy documentation for more information about Closures.)
- Any object that implements `oracle.jbo.Row`, `oracle.jbo.RowSet`, or `oracle.jbo.ExprValueSupplier` is automatically wrapped at run time into a Groovy “Expando” object to extend the properties available for those objects beyond the bean properties. This enables easy reference to ADF row properties (even if no Java class is generated) and avoids introspection for most used names.

Using Groovy Expressions for Validation

- You use the Script Expression validator or Compare validator when using Groovy for validation.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Groovy Expressions for Validation

You can use a Groovy expression to return a true/false statement. The Script Expression validator *requires* that the expression return either `true` or `false`. A common use of this feature would be to validate an attribute value, for example, to make sure that an account number is valid or check that one date is later than another, as in the example in the slide.

To use Groovy to validate a true/false expression:

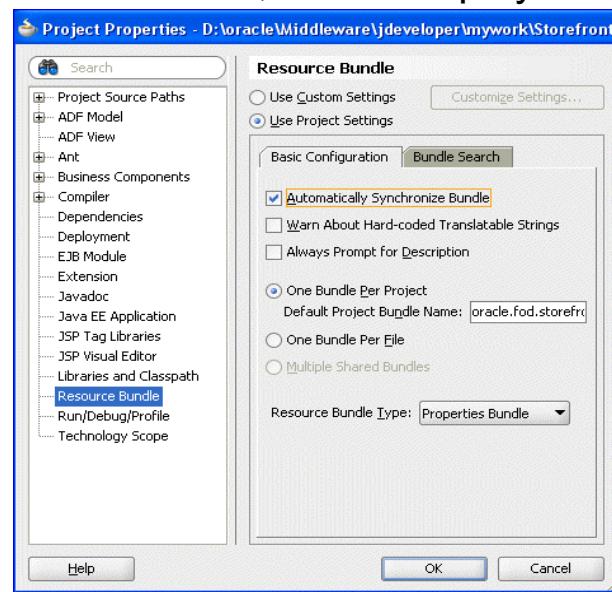
1. In the Application Navigator, double-click the desired entity object.
2. On the **Business Rules** page of the overview editor, select where you want to add the validator.
 - To add an entity-level validator, select the **Entity** folder.
 - To add an attribute-level validator, expand the **Attributes** folder and select the appropriate attribute.
3. Click the green plus sign to create a new validator.
4. In the Add Validation Rule dialog box, in the Rule Type drop-down list, select **Script Expression**.
5. Enter a validation expression in the field provided.
6. You can optionally click the Validation Execution tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

Using Groovy Expressions for Validation (continued)

7. Click the Failure Handling tab and enter or select the error message that will be shown to the user if the validation rule fails.
8. Click OK.

Internationalizing the Data Model

- When an application is internationalized, the UI displays text based on a user's browser settings.
- To use resource bundles for translatable strings, you can:
 - Create one bundle per file or project
 - Translate and append `_<locale>` to the file name
 - Configure the UI



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Internationalizing the Data Model

Many of the declarative modifications that you make to the data model involve text, such as labels or messages. ADF makes it easy to translate such text by offering the option to store it in resource bundles. You can use one resource bundle for the entire project or a separate one for each file.

Resource bundles are properties files containing translatable strings stored with key values and optionally with descriptions that are stored as comments. For example, the label for the OrderDate field of the Orders entity object is stored as:

```
#Used for the label of Orders.OrderDate
ORDER_DATE=Order Date
```

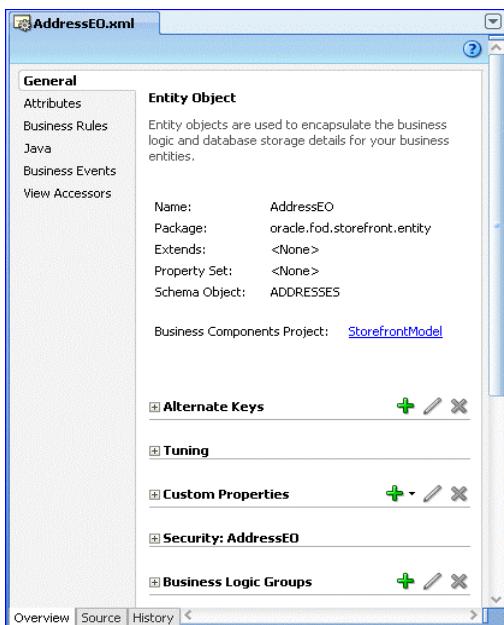
where ORDER_DATE is the key, Order Date is the translatable string, and the first line is the description. Although the description suggests the usage, the string can be used anywhere.

To provide multiple languages, a translator needs only to copy the properties file and append `_<locale>` to the name. For example, the properties file depicted in the slide may have a German translation; the name of that file would be `Project1Bundle_de.properties`. All the translations should be stored in the same directory.

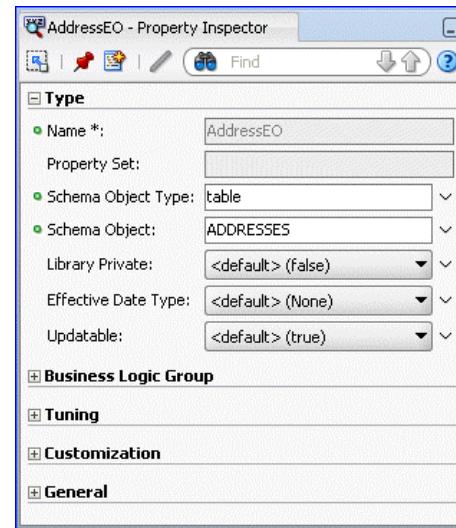
When you configure the user interface to support multiple translations as explained in the lesson titled "Planning the User Interface," which translation is used depends on the client's browser settings.

Editing Business Components

Editors provide access to business component properties:



Entity Object Editor



Property Inspector

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Editing Business Components

You can access all the properties of entity objects, associations, view objects, view links, and application modules by using the object editors and the property inspectors.

You can open the flat editor for an object by double-clicking the component in the Application Navigator, or by right-clicking it and selecting Open from the context menu. The flat editor is nonmodal, with collapsible headers.

The editor enables you to edit the .xml file for the entity object. You can edit it declaratively on the Overview tab, although in rare instances you may need to edit it directly by clicking the Source tab at the bottom of the editor. The History tab enables you to see changes that have been made and to reverse those changes if desired.

In addition to the editors, the Property Inspector enables you to edit business components. If the Property Inspector is not visible, you can display it by selecting View > Property Inspector.

The Property Inspector also has tabs to access panels that display various categories of properties for the selected object. If the property is also displayed in the editor, its value is synchronized so that a value change is reflected in both places. The Property Inspector may also display some properties of objects that are not accessible in the editor for that object.

The Property Inspector features a search bar and a toolbar; these are described thoroughly in the lesson titled “Getting Started with JDeveloper.”

Modifying the Default Behavior of Entity Objects

With declarative settings, you can:

- Define attribute control hints
- Synchronize columns with trigger-assigned values
- Use alternate key entity constraints
- Validate user input (presented in the lesson titled “Validating User Input”)



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of Entity Objects

JDeveloper 11g has the ability to define more behaviors declaratively than did earlier JDeveloper versions. This reduces the learning curve and the amount of time it takes to develop applications.

The subsequent slides provide some examples of ways to declaratively modify the default behavior of entity objects.

Modifying the Default Behavior of Entity Objects

With declarative settings, you can:

- Define attribute control hints
- Synchronize columns with trigger-assigned values
- Use alternate key entity constraints
- Validate user input (presented in the lesson titled “Validating User Input”)

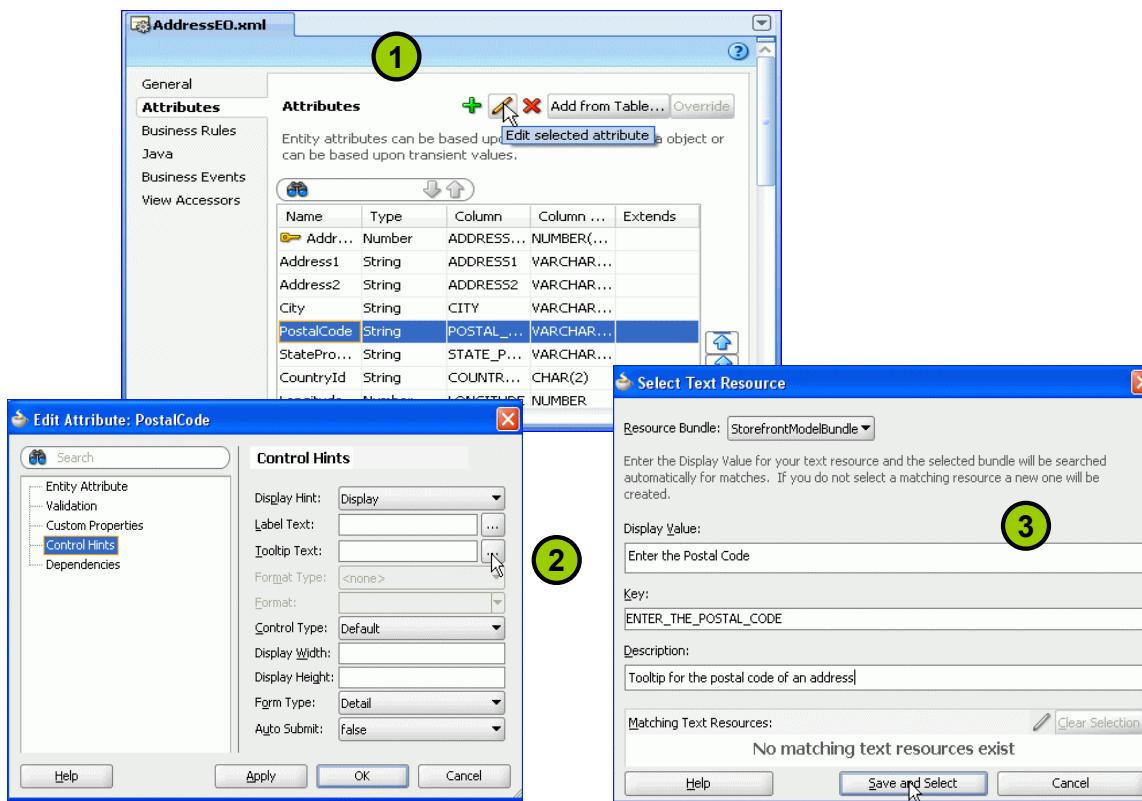


Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of Entity Objects

First you learn how to declaratively set control hints on entity object attributes.

Defining Attribute Control Hints



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Defining Attribute Control Hints

One of the many powerful, built-in features of ADF Business Components is the ability to define control hints on attributes. Control hints are additional attribute settings that the view layer can use to automatically display the queried information to the user in a consistent, locale-sensitive way. Control hints set on view objects override those set at the entity object level.

To add control hints for an entity object or a view object attribute, perform the following steps:

1. Open the object editor, select the Attributes node in the tree at the left, select the attribute, and click Edit.
2. In the attribute editor, click Control Hints, and set the desired properties. To add text to a resource bundle, click the ellipsis (...) next to the text.
3. Add the displayed value, accept the generated key or change it, and then click “Save and Select” to use the new text resource.

You can set the following control hints:

- **Display Hint:** Determines whether the attribute should be displayed
- **Label Text:** Changes the default label for the attribute. Labels are prompts or table headers that precede the value of a field.
- **Tooltip Text:** Sets text to appear in tooltip or the <ALT> attribute of HTML
- **Format Type:** Sets the formatter to use, such as currency or number for a number attribute

Defining Attribute Control Hints (continued)

- **Format:** Sets the format to use. Select a format from a list or enter any Java format that is valid for that type.
- **Control Type:** Specifies the type of control that the UI displays
- **Display Width:** Sets the character width to be used by the control that displays this attribute
- **Display Height:** Sets the number of character rows to be used by the control that displays this attribute
- **Form Type:** Specifies whether the attribute is displayed in detail or summary style layout (not valid for Web applications)
- **Auto Submit:** When enabled (`true`) , Auto Submit triggers a partial submit on value changes in the user interface. The default is `false`.

Modifying the Default Behavior of Entity Objects

With declarative settings, you can:

- Define attribute control hints
- Synchronize columns with trigger-assigned values
- Use alternate key entity constraints
- Validate user input (presented in the lesson titled “Validating User Input”)



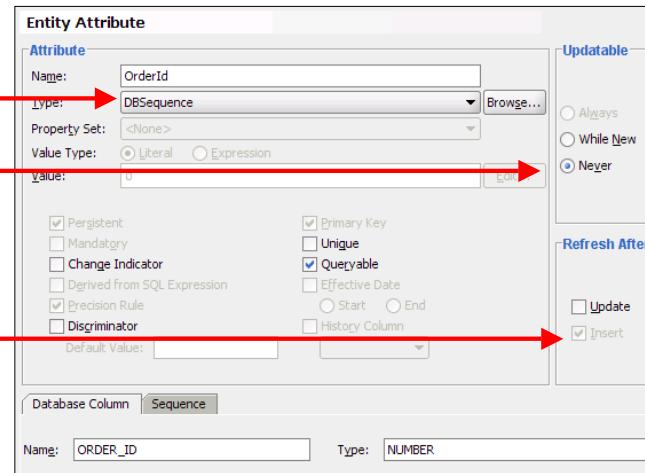
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of Entity Objects

Next you learn to synchronize entity object attribute values with database columns whose values are assigned by database triggers. The most common example of this is when a database trigger assigns a primary key value from a database sequence.

Synchronizing with Trigger-Assigned Values

- To synchronize when a database trigger updates a column, use Refresh After Update or Refresh After Insert.
- To use a database sequence to generate a primary key:
 - Set data type of attribute to DBSequence
 - Set Updatable to Never
 - Refresh After Insert is selected automatically.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Synchronizing with Trigger-Assigned Values

When a database trigger updates the underlying column value during insert or update operations, you can select the respective Refresh After Insert or Refresh After Update check boxes to ensure that the framework automatically retrieves the modified value and keeps the entity object and database row in sync. The entity object then uses the Oracle SQL RETURNING INTO feature to return the modified column back to your application in a single database round trip.

One common case for Refresh After Insert occurs when a primary key attribute value is assigned by a BEFORE INSERT FOR EACH ROW trigger. Often the trigger assigns the primary key from a database sequence. In this case, you can set the Attribute Type to the built-in data type DBSequence. Setting this data type automatically enables the Refresh After Insert property.

When you create a new entity row whose primary key is DBSequence, a unique negative number gets assigned as its temporary value. This value acts as the primary key for the duration of the transaction in which it is created. At transaction commit time, the entity object issues its INSERT operation using the RETURNING INTO clause to retrieve the actual database trigger-assigned primary key value. Any related new entities that previously used the temporary negative value as a foreign key will get that value updated to reflect the actual new primary key of the master. You can set the Updatable property of a DBSequence-valued primary key to Never, because the end user never needs to update this value.

Synchronizing with Trigger-Assigned Values (continued)

Note: The sequence name shown on the Sequence tab applies only at design time when you create a database table from an entity object. The sequence indicated on this tab is created along with the table on which the entity object is based.

Modifying the Default Behavior of Entity Objects

With declarative settings, you can:

- Define attribute control hints
- Synchronize columns with trigger-assigned values
- Use alternate key entity constraints
- Validate user input (presented in the lesson titled “Validating User Input”)



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of Entity Objects

Next you learn about alternate key entity constraints, why you would want to use them, and how to define them.

Using Alternate Key Entity Constraints

Alternate keys are:

- Used for efficient uniqueness checks
- Used for direct row lookups with `findByPrimaryKey()` methods
- Different from primary keys or unique keys



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Alternate Key Entity Constraints

Alternate keys are first class key citizens in ADF BC, and are stored in a hashmap for fast access. They are useful for:

- Efficient uniqueness checks in the middle tier via the Unique Key validator
- Direct row lookups via the `findByPrimaryKey()` class of methods

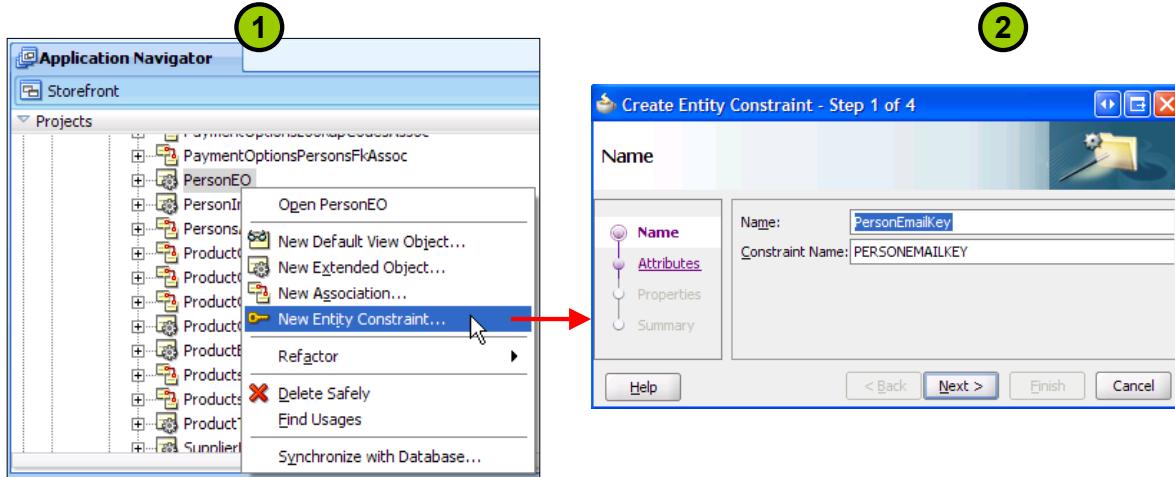
They differ from:

- Primary keys in that it is possible to define any number of multipart alternate keys
- Unique keys in that those are used primarily for forward generation of UNIQUE constraints in the database

For example, if you want to be able to locate a Department entity instance based either on the unique `DepartmentId` (primary key) or the `DepartmentName`, you can define an alternate key for the `DepartmentName` attribute. You can then use this named alternate key in the new `findByAltKey()` entity object API, or you can reference it in a Unique Key validator to automatically enforce that two departments cannot have the same `DepartmentName` value.

Creating Alternate Key Entity Constraints

Use the New Entity Constraint Wizard:



ORACLE

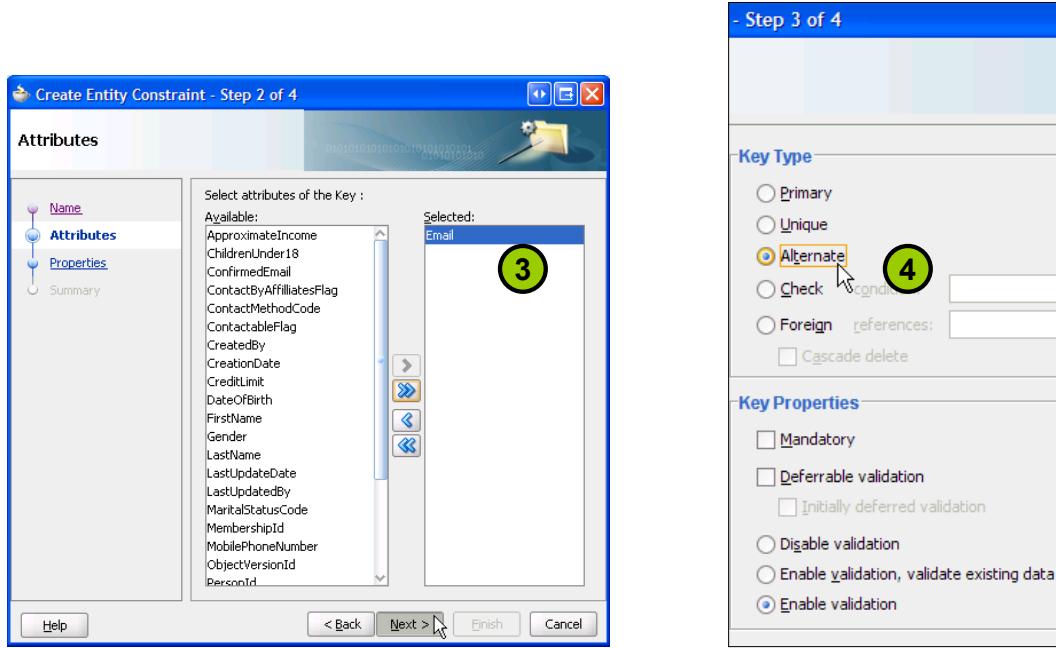
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating Alternate Key Entity Constraints

To create an alternate key entity constraint, perform the following steps:

1. Select an entity object and run the New Entity Constraint Wizard.
2. Name the constraint.

Creating Alternate Key Entity Constraints



ORACLE

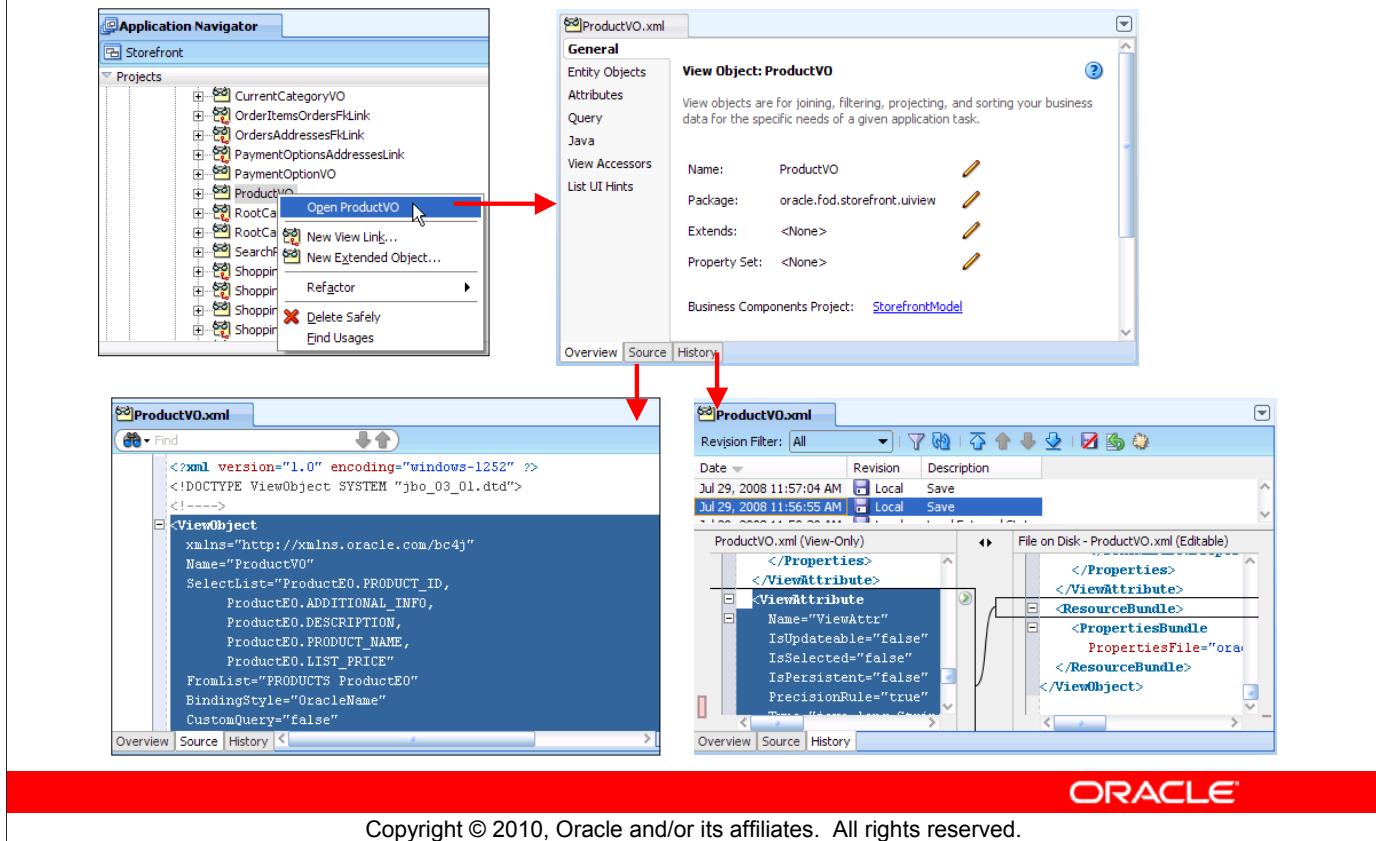
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating Alternate Key Entity Constraints (continued)

3. Select the columns to use.
4. Select the Alternate Key constraint type. You may elect to make the Alternate Key attributes mandatory, which will be automatically validated at run time by the implicit entity-level MandatoryAttributesValidator, which loops through *all* mandatory constraints on the row.

If you want to check the uniqueness of the key, you must explicitly add an entity-level Unique Key validator on the Alternate Key attributes. Adding a validator is explained in the lesson titled “Validating User Input.”

Editing View Objects



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Editing View Objects

You can invoke the view object editor by double-clicking the view object in the Application Navigator, or by right-clicking it and selecting Edit <VO Name> from the context menu.

The editor enables you to edit the .xml file for the view object. You can edit it declaratively on the Overview tab, although in rare instances you may need to edit it directly by clicking the Source tab at the bottom of the editor. The History tab enables you to see changes that have been made and to reverse those changes if desired.

When you click the Overview tab, you are presented with navigation tabs that contain the following categories:

- **General:** Set custom properties, tuning, or alternate keys for the view object.
- **Entity Objects:** Modify or add entity objects on which to base the view object.
- **Attributes:** Modify or add attributes or define custom properties or lists of values.
- **Query:** Modify the SQL query and set bind variables or view criteria.
- **Java:** Generate Java files or expose custom methods to client applications.
- **View Accessors:** Define a set of possible values for a given foreign key; used mainly for validation and for lists of values.
- **List UI Hints:** Specify default display settings for the view object when used as a list in the UI.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- Restrict the rows retrieved by a query
- Retain and reuse a row set
- Define a list of values



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of View Objects

The next few slides present several examples of the ways in which you can declaratively modify the default behavior of view objects.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- Restrict the rows retrieved by a query
- Retain and reuse a row set
- Define a list of values



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of View Objects

First you learn about defining attribute control hints and why you might want to do so at the view object level.

Defining View Object Control Hints

- Same as for entity objects
- Overrides EO control hint settings; use in cases such as:
 - The same data must be displayed differently in different views
OR
 - The VO uses the same attribute twice as in a recursive relationship
- Can also be used for transient attributes defined at the VO level



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Defining View Object Control Hints

View objects use the control hints from their associated entity objects, so you would not normally need to define them at the view object level. It is better to define them for the entity object so that displays are consistent throughout the application.

However, you can define the same control hints for view objects as you can for entity objects. If there is a control hint defined for a view object, it overrides the one defined at the entity object level.

There may be cases where you would want to override the entity object control hints for a specific view, such as the following:

- You may need to display the data differently in different views. For example, in an application's entity objects, you may define the label for all of the primary key items as ID, such as for ProductId, SupplierId, WarehouseId, and so on. But if you have a particular view that displays the IDs of multiple referenced entity objects, for that view object you may want to override the entity-level label for those IDs, as described in the slide.

Defining View Object Control Hints (continued)

- You may display the same attribute twice in a view. For example, imagine defining the Label UI hint for the Ename attribute of the Emp EO to be the string “Employee’s Name.” If there is an association that relates the Emp EO to itself via the relationship (Emp.Mgr = Emp.Empno), you may create a View Object that includes the Emp EO usage twice, once to include employee information and again to display information about that employee’s manager. In the context of this VO, you might have the Ename attribute from the base Emp EO as well as the Ename attribute to show the name of the employee’s manager. If you allow the inherited EO-level UI hints to be used, both the employee’s name and the manager’s name in this view object will use the inherited UI control hint label of “Employee’s Name.” You can define the VO-attribute-level UI hint for the ManagerEname view object attribute to have the hint “Manager’s Name” to clarify the situation, which will override the inherited “Employee’s Name” hint for that attribute.

In addition to the attributes that are based on entity objects, you may have some transient attributes that are defined only at the view object level, so you may want to define control hints for those.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- Restrict the rows retrieved by a query
- Retain and reuse a row set
- Define a list of values

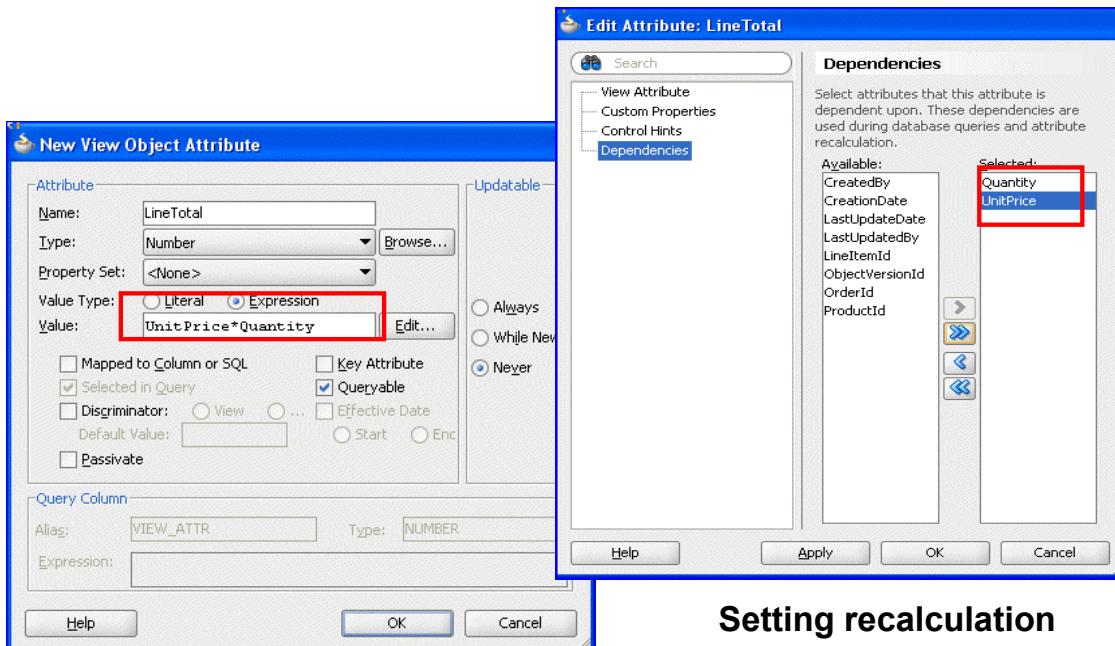


Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of View Objects

Next you learn how to use a calculation to set the value of a transient attribute.

Performing Calculations



Setting recalculation dependencies

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Performing Calculations

You can calculate the value of an attribute by entering an expression.

For example, a view object based on OrderItemEO may have a transient attribute called LineTotal. To implement a calculation for that attribute, perform the following steps:

1. Invoke the attribute editor for LineTotal.
2. In the View Attribute panel:
 - a. Select the Expression option for Value Type. Expressions can use the Groovy expression language and can reference attributes from the view object definition.
 - b. Enter the calculation for Value: UnitPrice*Quantity.
 - c. In the Dependencies panel, shuttle Quantity and UnitPrice to the Selected list. This ensures that the value for LineTotal is recalculated whenever the value for either Quantity or UnitPrice changes.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- Restrict the rows retrieved by a query
- Retain and reuse a row set
- Define a list of values



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of View Objects

Next you learn how to modify the way in which database table columns are returned by the view object query.

Restricting and Reordering the Columns Retrieved by a Query

- Use the Attributes tab of the editor to delete or reorder attributes.
- Query changes to reflect the new SELECT clause.

```
SELECT Persons.PERSON_ID,
       Persons.PRINCIPAL_NAME,
       Persons.TITLE,
       Persons.FIRST_NAME,
       Persons.LAST_NAME,
       Persons.PERSON_TYPE_CODE,
       Persons.SUPPLIER_ID,
       Persons.PRIMARY_ADDRESS_ID
  FROM PERSONS Persons
```

Original query

Name	Type	Alias Name
PersonId	Number	PERSON_ID
PrincipalName	String	PRINCIPAL_NA
Title	String	TITLE
FirstName	String	FIRST_NAME

Deleting attributes

Name	Type	Column	Info
PersonId	Number	Persons:PERSON...	
FirstName	String	Persons:FIRST_N...	
LastName	String	Persons:LAST_N...	
PersonTypeCode	String	Persons:PERSON...	
SupplierId	Number	Persons:SUPPLIE...	

Reordering attributes

Modified query

```
SELECT Persons.PERSON_ID,
       Persons.LAST_NAME,
       Persons.FIRST_NAME,
       Persons.PERSON_TYPE_CODE,
       Persons.SUPPLIER_ID,
       Persons.PRIMARY_ADDRESS_ID
  FROM PERSONS Persons
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Restricting and Reordering the Columns Retrieved by a Query

The Attributes tab of the view object editor enables you to restrict the columns retrieved by the query by deleting attributes. You can also add attributes, either new ones or, for entity-based views, attributes based on columns that exist in the entity object. You can also change the order in which the attributes appear in the query statement.

The example in the slide shows an existing query that selects PRINCIPAL_NAME, TITLE, FIRST_NAME, LAST_NAME, and some other columns. After deleting and reordering attributes, the query is changed and no longer selects the deleted PRINCIPAL_NAME and TITLE columns, and the order in which LAST_NAME and FIRST_NAME are selected is reversed.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- Restrict the rows retrieved by a query
- Retain and reuse a row set
- Define a list of values



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of View Objects

You also can change the order of the queried rows, as described next.

Changing the Order of Queried Rows

To change the order, perform the following steps:

1. Click the Query tab of the view object editor.
2. Click Edit  in the Query section of the panel.
3. Click Edit in the Edit Query dialog box.
4. In the Order By dialog box, select the columns for sorting the rows retrieved by the query.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Changing the Order of Queried Rows

You can set an ORDER BY clause on the query by clicking the Query tab of the view object editor. To invoke the Edit Query dialog box, click Edit (pencil icon) in the Query section of the panel. Enter an ORDER BY clause (without the ORDER_BY keyword). If you click Edit to the right of the Order By field, you invoke a dialog box that enables you to select the columns and construct the ORDER BY clause.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- **Restrict the rows retrieved by a query**
- Retain and reuse a row set
- Define a list of values



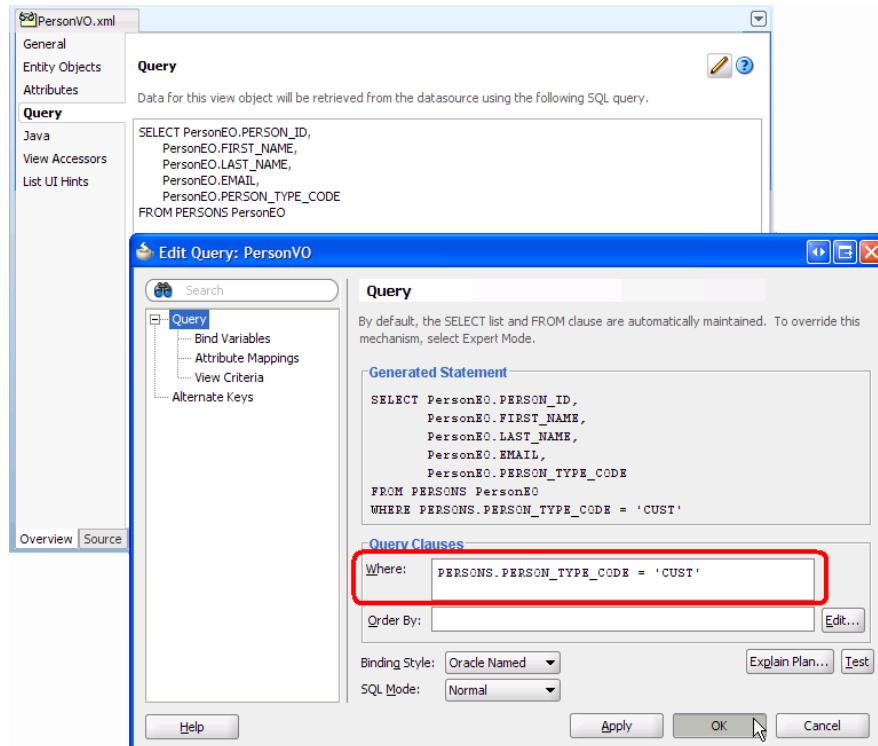
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of View Objects

There are several declarative ways to restrict the rows retrieved by a query, as explained in the following slides.

Restricting the Rows Retrieved by a Query

Add a WHERE clause.



ORACLE

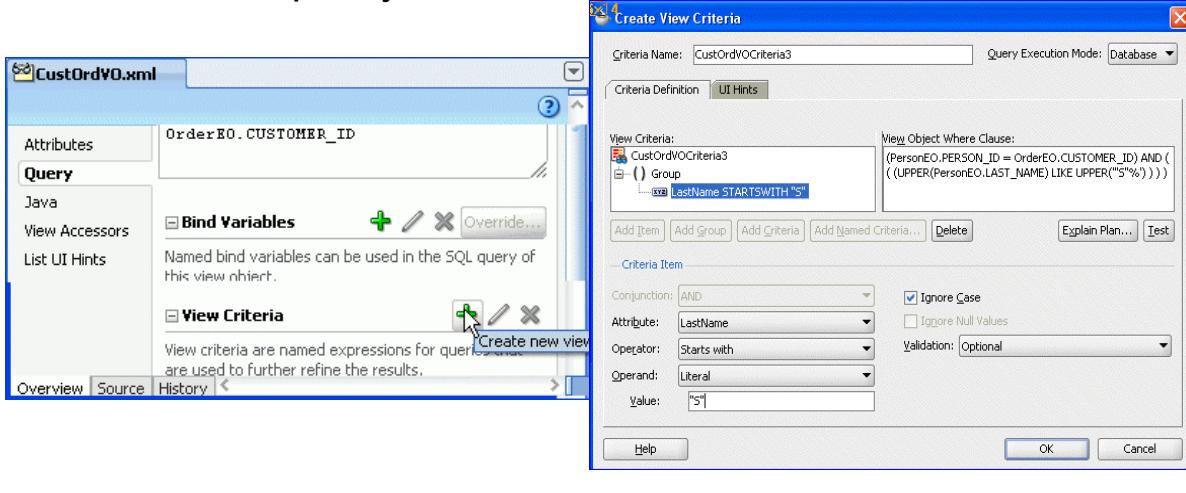
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Restricting the Rows Retrieved by a Query

One way to restrict the rows retrieved by a query is to set a WHERE clause on the query panel of the view object editor. To invoke the Edit Query dialog box, click Edit (pencil icon) in the Query section of the panel. You can hard code the value for the WHERE clause, or you can use a bind variable as described in the section titled “Using Named Bind Variables” later in this lesson.

Using View Criteria (Structured WHERE Clauses)

- You can use the view criteria to define complex query criteria at design time.
- You can specify the execution mode.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Structured WHERE Clauses (View Criteria)

Instead of changing WHERE clauses by using string manipulation, you may find it more efficient to use view criteria to assemble complex WHERE clauses. View criteria are named expressions that you can use to assemble a WHERE clause.

A view criteria row specifies query-by-example requirements for one or more view object attributes. For example, suppose the view object definition CustOrdVO includes the following attributes: LastName, OrderTotal, and DiscountId. Then a view criteria row for an instance of CustOrdView could specify requirements for each of these attributes, such as: LastName STARTSWITH "S", OrderTotal > 500, and DiscountId <> NULL.

Another view criteria row for this instance could specify different requirements for a different set of attributes, such as: OrderTotal > 1000, and CreditLimit > 5000.

For a view criteria row to be satisfied, all of its attribute requirements must be met.

View criteria are made up of collections of view criteria rows. For the view criteria to be met, at least one of the rows must be satisfied. For example, if vc1 has the requirements:

OrderTotal > 500, CreditLimit > 2500, and DiscountId <> NULL, and vc2 has the requirements: OrderTotal > 1000 and CreditLimit > 5000, then the complete view criteria are equivalent to the following WHERE clause condition:

Using Structured WHERE Clauses (View Criteria) (continued)

```
(OrderTotal > 500 AND CreditLimit > 2500 AND DiscountId <> NULL)  
OR  
(OrderTotal > 1000 AND CreditLimit > 5000)
```

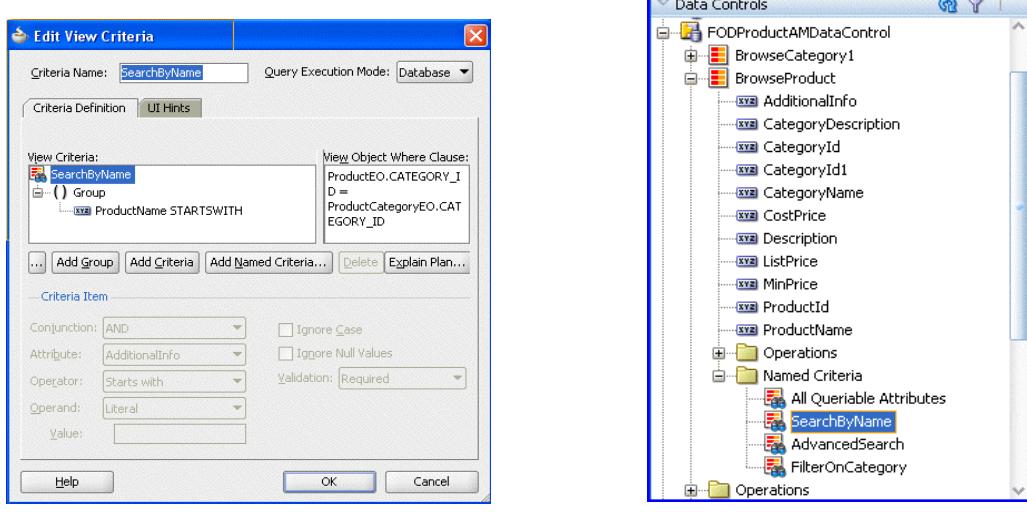
You can define view criteria at design time and apply them in the Business Components Browser to test them. Also, you can use bind variables for view criteria values.

The Query Execution Mode sets the source from which the view object retrieves rows. You can set it to:

- Database (default): Limits the results of the filtered view object to the database table specified by the query
- In Memory:
 - Limits the results to the in-memory results of the view object query
 - Uses rows already in the row set
 - Can be used to progressively refine the row set contents through in-memory filtering
 - Prevents unnecessary database access
- Both: Is useful when you have newly created, but not yet committed, results in the row set that you want to filter and combine with filtered results from the database table

Role of View Criteria in Search Forms

- Create view criteria and save them as named definitions.
- Named view criteria definitions appear under Named Criteria node in the Data Controls panel.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Role of View Criteria in Search Forms

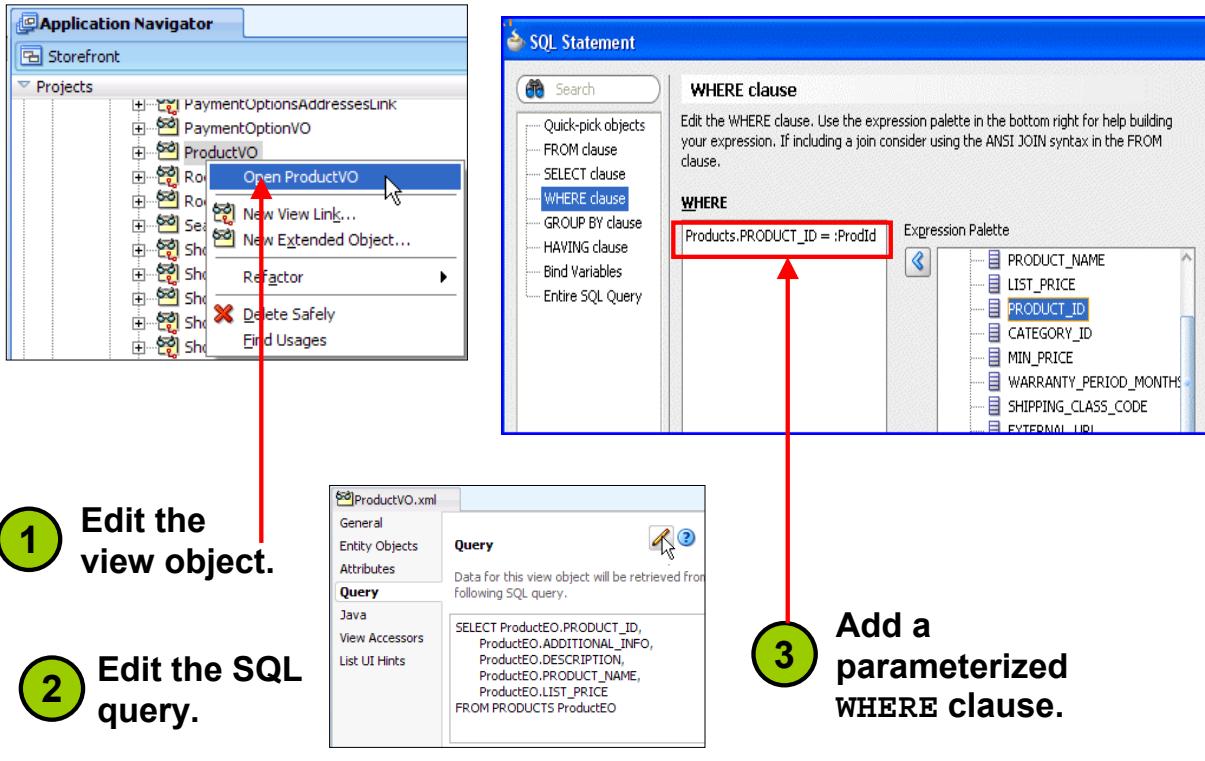
The view criteria that you define in the model facilitate the production of search forms required in the UI. The view criteria object is a row set of one or more view criteria rows, whose attributes mirror those in the view object. The view criteria definition comprises query conditions that augment the WHERE clause of the target view object. Query conditions that you specify apply to the individual attributes of the target view object.

The Edit View Criteria dialog box lets you create view criteria and save them as part of the view object's definition, where they appear as named view criteria. You use the Query page of the overview editor to define view criteria for specific view objects.

The example in the slide shows the definition of a view criteria, `SearchByName`, on the `BrowseProductVO` view object.

When the view objects are created and specified as instances in an application module, JDeveloper automatically creates a data control to encapsulate the collections (view instances) that the application module contains. JDeveloper then populates the Data Controls panel with these collections and any view criteria that you have defined, as shown on the right in the example in the slide. The `SearchByName` named criteria, part of the `BrowseProduct` collection in the `FODProductAMDataControl`, is available to be dropped onto the page for the creation of a Search form (as you learn in the lesson titled “Adding Functionality to Pages” in this course).

Using Parameterized WHERE Clauses



ORACLE

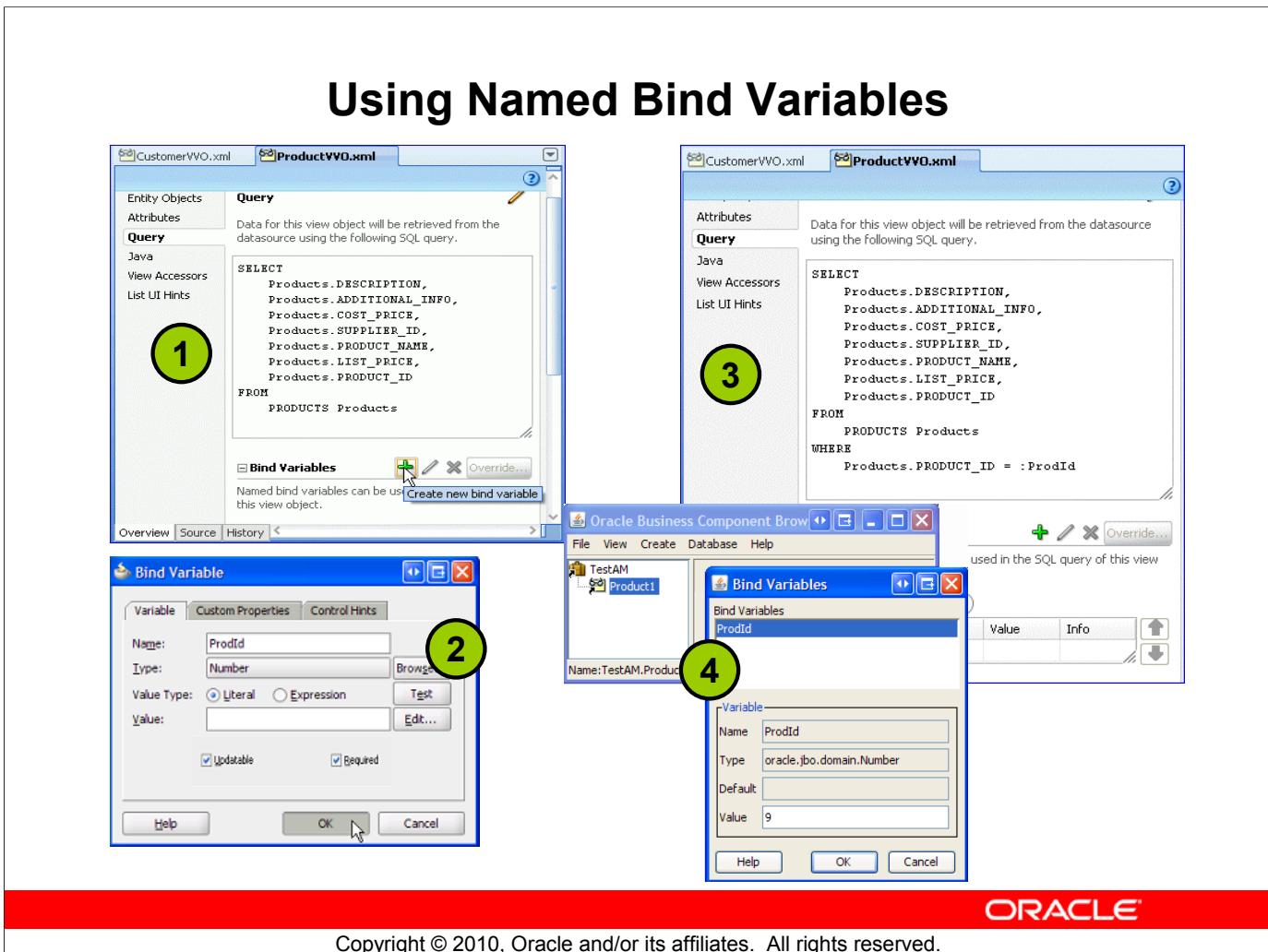
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Parameterized WHERE Clauses

If you have some, but not complete, information about the WHERE clause you want for a view object definition, you can dynamically set individual WHERE clause parameters in particular view object instances at run time.

To parameterize the WHERE clause in a view object definition, perform the following steps:

1. Open the view object editor.
2. Click the Query finger tab in the editor, and in the Query window, click Edit.
3. You can enter the complete WHERE clause at this point, or you can use Query Builder to help format the clause. Click Query Builder and select WHERE clause from the list of Quick-pick objects on the left. You can enter the text of the WHERE clause (without the WHERE keyword) or you can enter some of the text and use the Expression Palette to add object names, function names, and operators.



Using Named Bind Variables

Named bind variables enable developers to assign logical names to SQL query parameters, set minimum, maximum, and default values, and to reuse and reorder the parameters at will. Named bind variables can be assigned values in the Business Components Browser.

To use a named bind variable, perform the following steps:

1. On the Overview tab of the view object editor, click the Query finger tab. Then under Bind Variables, click Create.
2. In the Bind Variable dialog box, you specify the name, data type, and optionally a default value. You can name the variables as you like, but because they share the same namespace as view object attributes, you need to choose names that do not conflict with existing view object attribute names. The Updatable check box determines whether the value can be changed; by default, a nonupdatable named bind variable has no setter method generated for it and no methods are exposed to the client. You can click the other tabs in the dialog box to specify custom properties and UI control hints if desired.
3. After defining the bind variables, the next step is to reference them in the SQL statement in either the SELECT list or the WHERE clause.
4. When you run the Business Components Browser, a Bind Variables dialog box enables you to specify a value for the bind variables. You can also change or set this value on the toolbar.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- Restrict the rows retrieved by a query
- Retain and reuse a row set
- Define a list of values



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

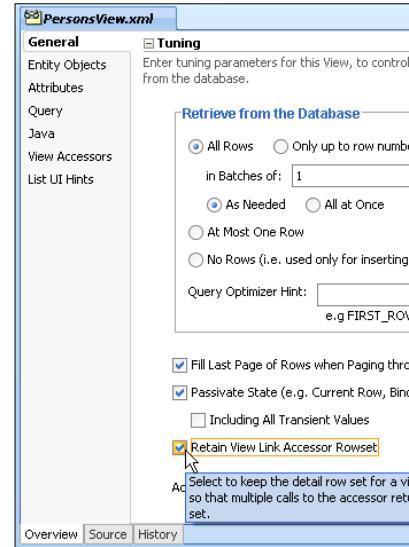
Modifying the Default Behavior of View Objects

Next you learn how to specify declaratively that a RowSet object should be reused, rather than creating a new instance of the RowSet each time the application accesses a view link accessor row set.

Retaining and Reusing a View Link Accessor Row Set

Retaining a view link accessor row set:

- Enables caching of the view link accessor RowSet object
- Avoids overhead of creating new detail RowSet objects
- May be advisable when the application makes numerous calls to the same view link accessor attributes
- Can be implemented declaratively in the Tuning section of the General tab for the view object that is the source for the view link



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Retaining and Reusing a RowSet

Each time you retrieve a view link accessor row set, by default the view object creates a new RowSet object to enable you to work with the rows. This does not imply reexecuting the query to produce the results each time, but only creating a new instance of a RowSet object with its default iterator reset to just before the first row.

Because there is a small amount of overhead associated with creating the row set, if your code makes numerous calls to the same view link accessor attributes, you can consider enabling view link accessor row set retention for the source view object in the view link. For example, because view accessor row sets remain stable as long as the master row view accessor attribute remains unchanged, it would not be necessary to re-create a new row set for UI components, such as the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows.

You can declaratively enable retention of the view link accessor row set by using the overview editor for the view object that is the source for the view link accessor. Select **Retain View Link Accessor Rowset** in the Tuning section of the General page of the overview editor for the view object.

The lesson titled “Programmatically Customizing Data Services” describes how to implement this functionality programmatically.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

- Define attribute control hints
- Perform calculations
- Restrict the columns retrieved by a query
- Change the order of queried rows
- Restrict the rows retrieved by a query
- Retain and reuse a row set
- Define a list of values



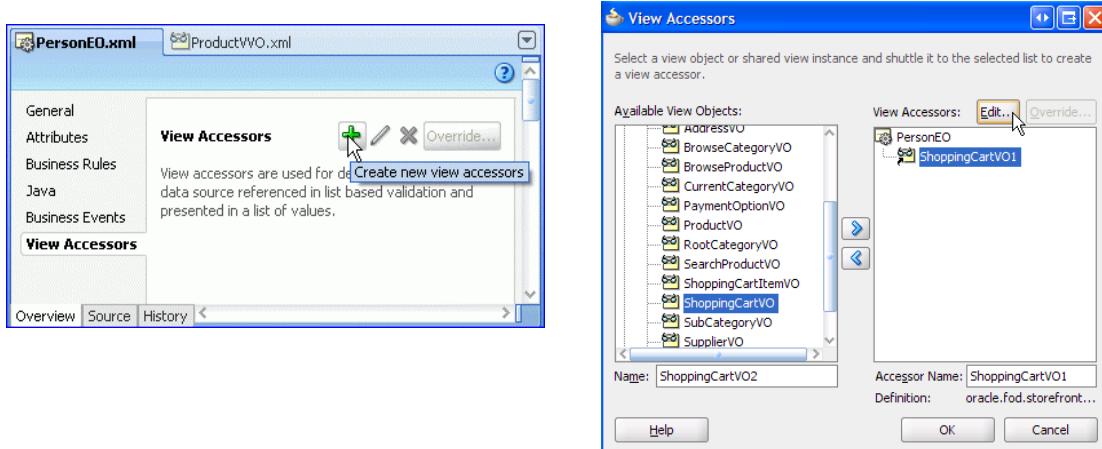
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of View Objects

The last declarative change that this lesson presents is how to declaratively define lists of values, which have view accessors as their data source.

Creating View Accessors

- View accessors are used for validation and LOVs.
- You create them on the View Accessors page of EO or VO editor.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating View Accessors

View accessors facilitate traversing from a detail record to all the associated master records. For example, a view accessor could enable you to traverse from an employee to all the departments the employee is allowed to join, rather than just to the department the employee is currently in. View accessors are commonly used both in validation, which is discussed in the lesson titled “Validating User Input,” and in lists of values, which is discussed next.

You can define view accessors for entity or view objects, but defining them at the entity level is usually preferable because it enables access by any view object that is based on that entity. You can define a view accessor to any view object in the same workspace. Simply click “Create new view accessors” on the View Accessors tab of the EO or VO editor, and then in the View Accessors dialog box, select the view object or objects to access. You can create multiple view accessors at once.

If the view accessor retrieves a view object on which view criteria or bind variables are defined, you can optionally use those for the view accessor to restrict the rows that are retrieved, and you can also add an ORDER BY clause. To choose these options, with the view accessor selected in the View Accessors dialog box, click Edit. You can also set these options after creating the view accessor by editing it.

Using a List of Values (LOV)

LOVs:

- Are defined on view object attributes
- Use view accessors as a data source:
 - For attributes mapped to an entity object attribute:
 - Use the view accessor that is used by the entity object attribute's Key Exists validator
 - Extend the view accessor at the view object level for UI hints or bind expressions if required
 - For transient attributes, you can define a new view accessor.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

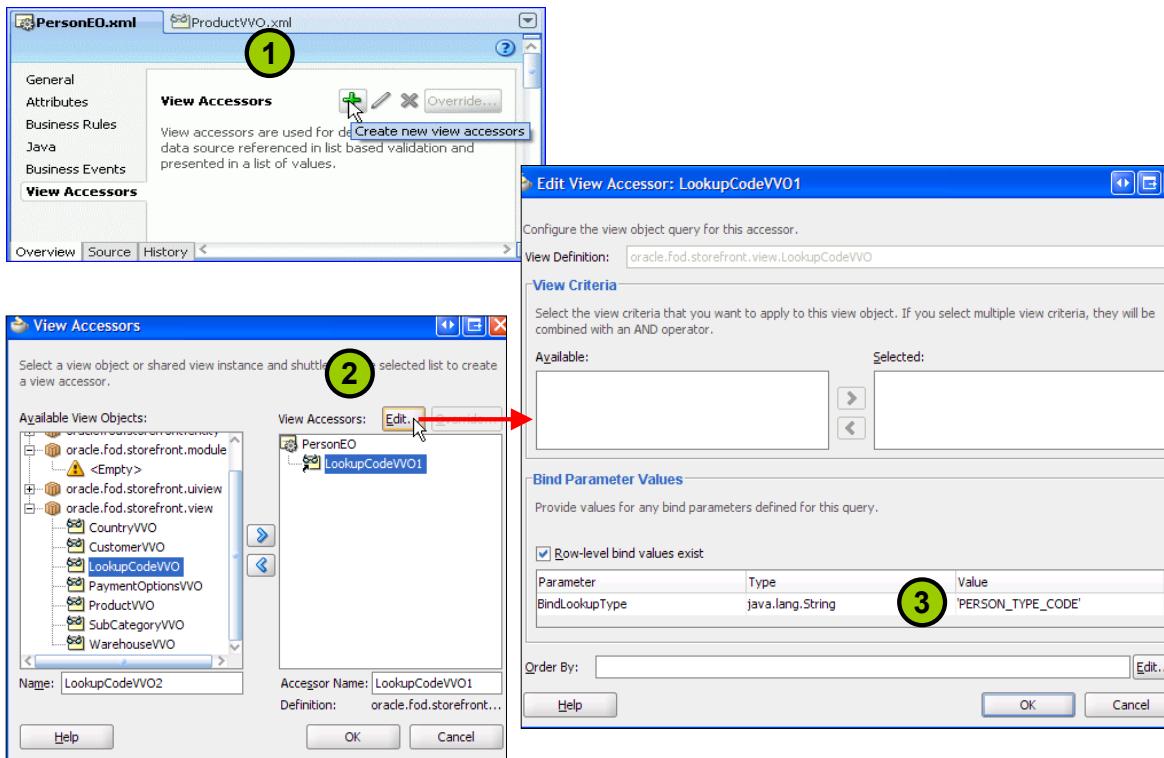
Using a List of Values (LOV)

You can define LOVs on view object attributes.

LOVs use view accessors as a data source:

- For an attribute that is mapped to an entity object, you normally use the same underlying entity object view accessor that is used by the Key Exists validator for the underlying EO attribute. The entity object view accessor can be extended at the view object level to either change the UI Hints or change bind expressions.
- You can define view object view accessors to be used for transient attributes that are not mapped to entity object attributes.

Defining the View Accessor for the List of Values



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Defining the View Accessor for the List of Values

A list of values takes its data from another view, which must have a view accessor defined on the entity object or view object where you create the LOV. To define a view accessor for a list of values, perform the following steps:

1. Create a view accessor to the view object that serves as a data source for the LOV. The example in the slide shows creating a view accessor from the PersonEO entity object to the LookupCodeVVO view object. On the View Accessors panel for PersonEO, click Add (the green plus icon.)
2. In the View Accessors dialog box, shuttle the view to the View Accessors panel.
3. In this case, while creating the view accessor, you would also need to click Edit to supply a value to the bind variable that is used in the WHERE clause, setting the LookupType to 'PERSON_TYPE_CODE'.

Defining the List of Values

The screenshot displays two windows from the Oracle Fusion Middleware 11g interface:

- Top Window (List of Values):** Shows the configuration of a List of Values named "LOV_PersonTypeCode". The "Default List Type" is set to "Choice List". Under "Display Attributes", "Meaning" is selected. A green circle labeled "1" points to the "Add" button in the "Lists of Values" section.
- Bottom Window (List of Values):** Shows the selection of a List Data Source ("PersonEO.LookupCodeVVO1") and List Attribute ("LookupCode"). A green circle labeled "2" points to the "List Attribute" dropdown. This window also includes a "List Return Values" section and a "View Attribute" mapping.
- Right Panel (Preview):** Displays a dropdown menu with the following items:
 - PersonId: 100
 - First Name: Steven
 - Last Name: King
 - Email: SKING
 - Person Type: Staff (highlighted in blue)
 - Customer
 - Supplier
 A green circle labeled "3" points to the "Selected" list, and another green circle labeled "4" points to the "Person Type" item in the dropdown list.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Defining the List of Values

To define a list of values and attach it to an attribute in a view object, perform the following steps:

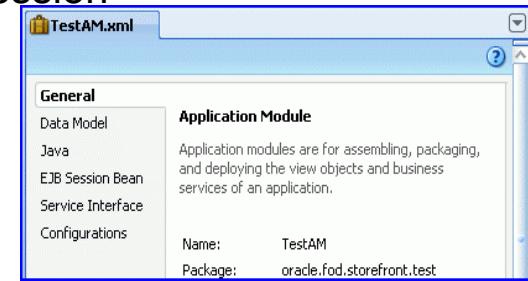
1. Create a list of values on the view object attribute that points to the view accessor. To do so, you select the attribute in the Attributes panel, expand the List of Values header, and click Add (the green plus icon). The slide shows creating an LOV on the PersonTypeCode attribute in the PersonVO view object.
2. In the “List of Values” dialog box, select the view accessor to use as the List Data Source, along with the attribute to populate the view object’s attribute. The example uses LookupCode from the PersonEO.LookupCodeVVO1 view accessor to populate PersonTypeCode in the PersonVO view object. **Note:** If you had not already created the view accessor, you could click Add (the green plus icon) in this dialog box to create one.
3. On the UI Hints tab of the List of Values dialog box, you select the list type to use as a default, the attribute or attributes to display in the list, whether selection is required, and how to display a “No Selection” item in the list if selection is not required. In the example, the Meaning attribute is selected to display in the list, and selection is required.
4. The list of values automatically renders as a drop-down list at run time.

You can optionally create a List validator on the entity attribute that points to the view accessor. Adding a validator is explained in the lesson titled “Validating User Input.”

Modifying Application Modules

The Application Module editor has the following tabs:

- General: Set tuning parameters and define custom properties.
- Data Model: Refine the data model.
- Java: Create Java classes and expose methods to the client interface.
- EJB Session Bean: Enable EJB session bean support.
- Service Interface: Enable Service Interface support.
- Configurations: Create or modify sets of configuration parameters.



ORACLE

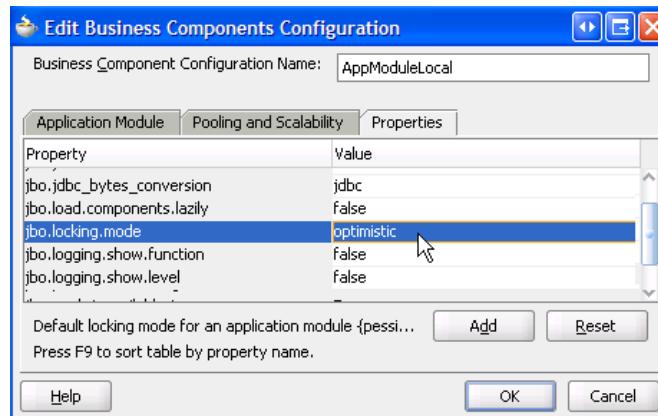
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying Application Modules

The application module editor contains several tabs that enable you to declaratively modify aspects of the application module, as listed in the slide. You learn more about modifying application modules as you complete the course application.

Changing the Locking Behavior of an Application Module

- Default locking behavior is pessimistic.
- Optimistic locking is recommended for Web applications.
- Change configuration: Set `jbo.locking.mode` to `optimistic` or `optupdate`.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Changing the Locking Behavior of an Application Module

By default, the application module uses pessimistic locking, which means that a lock is issued on a database row when an application changes data in that row.

Oracle recommends using optimistic locking for Web applications. A lock is acquired only just before posting changes to the database, and the application module instance can be immediately released when a Web page terminates. This provides the best level of performance for Web applications that expect many users to access the application simultaneously.

To change your configuration to use optimistic locking, perform the following steps:

1. Click the Configurations tab of the Application Module editor.
2. Select a configuration and click Edit.
3. On the Properties tab of the Configuration Editor, set the value of the `jbo.locking.mode` property (default value is pessimistic) to one of the following:
 - `optimistic`: Issues a `SELECT FOR UPDATE` statement to lock the row for both updates and deletes, then detects whether the row has been changed by another user by comparing the change indicator attribute if there is one, or the values of all the persistent attributes of the current entity as they existed when fetched into the cache
 - `optupdate`: Optimistic locking only for updates; the `UPDATE` statement determines whether the row was updated by another user by including a `WHERE` clause that matches the existing row; an update occurs if attributes remain unchanged since fetch.

Summary

In this lesson, you should have learned to:

- Describe how to declaratively change data behavior
- Declaratively modify view objects, entity objects, and application modules
- Create view accessors
- Create LOVs



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 6 Overview: Declaratively Modifying Business Components

This practice covers the following topics:

- Declaratively Populating a Primary Key with a Database Sequence
- Designating History Columns
- Creating and Using View Criteria
- Creating Join View Objects
- Creating LOVs



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 6 Overview: Declaratively Modifying Business Components

In this set of practices, you make declarative modifications to your business components.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Osi S.R.L. use only

Programmatically Customizing Data Services



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Generate Java classes for business components
- Override class methods
- Implement programmatic modifications
- Add service methods to an application module
- Create a test client
- Use business component client APIs



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

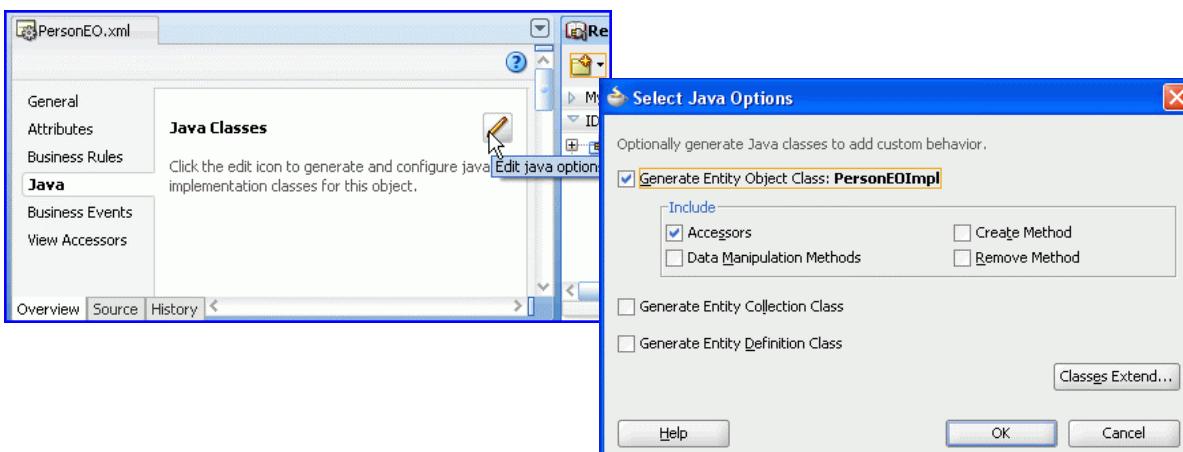
Lesson Aim

This lesson teaches you to implement business logic programmatically. You add business logic to entity and view object classes and add service methods to an application module. You learn to use ADF BC client APIs to change the behavior of business components, and then test the modified functionality by creating and running a test client.

Generating Java Classes for Adding Code

To generate a Java class:

1. On the business component editor's Java page, click Edit.
2. Select the classes to generate.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Generating Java Classes for Adding Code

ADF enables you to customize the behavior of your business components to the requirements of your application. Although an increasing number of modifications can be accomplished declaratively, there are cases where you must add code to meet the requirements.

By default, the only files generated for business components are the metadata files in XML. If you need to add Java code, you can optionally generate Java classes on the Java tab of the component's editor as follows:

- Click Edit (the pencil icon).
- In the Select Java Options dialog box, select the check boxes for the class and methods that you want to generate.
- Click OK to generate the class. Then add to the class the methods that you want to code.

Just as with the XML definition file, JDeveloper keeps the generated code in your Java classes up to date with any changes you make in the editor.

To delete a Java class that has already been generated, follow the same process, but deselect those classes that you want to delete. Any code that you may have added to those classes is lost when you delete the class.

Programmatically Modifying the Default Behavior of Entity Objects

Many modifications are possible with coding, such as:

- Traversing associations
- Overriding base class methods



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Default Behavior of Entity Objects

The following slides discuss the base Java classes of an entity object and how to add code to traverse associations and to override base class methods.

Supporting Entity Java Classes

- EntityImpl:
 - Is the entity class
 - Represents a row
 - Provides getter and setter methods
- EntityDefImpl:
 - Is the entity definition class
 - Represents the whole entity
 - Can be used to modify the entity definition
- EntityCollImpl:
 - Represents the cached set of rows from the entity
 - Is not usually necessary for the user to modify or override methods in this class

```

package oracle.fod.storefront.entity;
import ...;

// ...
// --- File generated by Oracle ADF Business Component
// --- Thu Jan 07 15:25:35 GMT 2010
// --- Custom code may be added to this class.
// --- Warning: Do not modify method signatures of generated code.

public class PersonEOImpl extends EntityImpl {
    private static EntityDefImpl mDefinitionObject;

    /**
     * AttributesEnum: generated enum for identifying attributes
     */
    public enum AttributesEnum {...}
    public static final int PERSONID = AttributesEnum.P...
}

PersonEOImpl
Source | Design | History

```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Coding: The Java Classes

There are three base Java classes that implement entity objects:

- EntityImpl: This is the entity object class. At run time, one entity object is instantiated for each row of data. This class contains methods to get and set entity attribute values. All entity object classes extend this class. The EntityImpl.java class provides methods to insert, update, delete, and lock rows. The ADF Business Components technology uses this class to manage instances of each entity. When you generate the entity object class, you can also generate:
 - Accessors: Generate typesafe accessors for the entity object's attributes.
 - Data Manipulation Methods: Generate methods to override DML methods, such as lock() and doDML().
 - Create Method: Select to override the create() method to modify or add initialization features to the create logic.
 - Remove Method: Select to override the remove() method to modify or add clean-up code to the remove logic.
- EntityDefImpl: This is the entity definition class. At run time, one entity definition is instantiated for each entity. You can add methods to this class that are used by all entity object instances. For example, you could override a method to globally change the default format mask for number attributes.

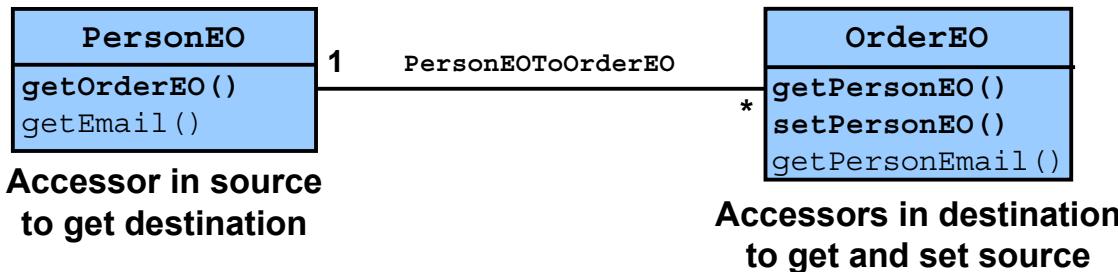
Coding: The Java Classes (continued)

- EntityCollImpl: This class represents the set of rows from the entity that is stored in the internal ADF Business Component cache. You do not usually need to generate or modify this class, but if you do, you may also choose to generate bind variable accessors and typesafe getters and setters for named bind variables. An example of when you might want to generate this class is if you need to programmatically retain and reuse an association accessor row set.

When you choose to generate Java classes for entity objects, the classes that are generated extend the base classes. The classes that you generate use the entity name, followed by `Impl`, `DefImpl`, or `CollImpl` (such as `PersonEOImpl.java`). These classes provide methods to manipulate either row-level data or the definition of the entity itself. You can use the generated classes to add custom validation or business logic.

You learn later in this lesson about the base classes for view objects and application modules.

Traversing Associations



- The destination entity's `EntityImpl.java` file contains methods to get and set the source entity. For example, `OrderEOImpl.java` contains `getPersonEO()` and `setPersonEO()`.
- You could add a method to `OrderEO.java` to get the email address of the associated person (customer):

```

public String getPersonEmail() {
    return getPersonEO().getEmail();
}
  
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Traversing Associations

By default, JDeveloper generates accessor methods for each association: a getter and a setter in the destination to retrieve the source, and a getter in the source to get the destination. These accessor methods are generated in the entity object classes on either end of the association. The client code does not access the entity object code directly, so you could use these methods in application module service methods or in programmatic view object or entity object code.

Example of Accessor Methods: The `PersonEO` and `OrderEO` entities are linked by the `PersonEOToOrderEO` association. JDeveloper generates two methods in `OrderEOImpl.java` (the destination entity object) for traversing the association from destination to source:

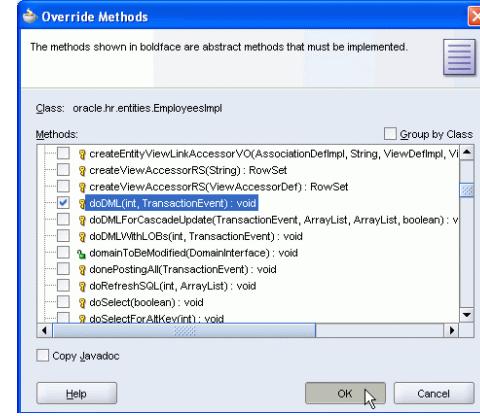
- `public PersonEOImpl getPersonEO()`: Get the department to which this employee belongs.
- `public void setPersonEO(PersonEOImpl value)`: Set the value of the department to which this employee belongs.

To retrieve an attribute's value from the source entity while in the destination entity, use the `get<Source>()` method plus the `get()` method for the attribute that you want to retrieve, as shown in the above example for retrieving the email address from the `PersonEO` entity. The `PersonEO` entity has only the `getOrderEO` accessor, with no setter.

Overriding Base Class Methods

You can override methods in the base classes for objects. For example, you can override methods in EntityImpl.java, such as:

- `doDML()`: Use it to log changes in another entity.
- `beforeCommit()`: Use it to validate multiple instances of the same entity.
- `remove()`: Use it to log a deletion in an entity.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Overriding Methods

In your object-specific `<object name>Impl.java` files, you can override methods inherited from the parent class.

For example, to override a method inherited from EntityImpl.java, select the `<entity name>Impl.java` file and select Source > Override Methods from either the main menu or the context menu, or click Override Methods on the editor toolbar. Select the methods that you want to override. You can locate a method by entering its name.

- `doRM()` method: This method fires after all the attribute and entity validation is complete. You can add code that is beyond validation logic here. For example, you could add code that records audit information in another entity.
- `beforeCommit()` method: This method is used for rules that involve more than one instance of the same entity or more than one entity. Because any of the instances could have changed in the current transaction, it is important to delay validation until all of the instances have been posted.
- `remove()` method: It is used for rules that are triggered by a delete action.

Overriding a method creates a skeleton code for that in the `<object name>Impl.java` file that performs the default functionality. You can then replace or augment that code. If you prefer, you can just enter the code instead of selecting Source > Override Methods.

Overriding Base Class Methods Example: Updating a Deleted Flag Instead of Deleting Rows

```
// In <Entity name>Impl.java

① public void remove() {
    setDeleted("Y");
    super.remove();
}

② protected void doDML(int operation,
    TransactionEvent e) {
    if (operation == DML_DELETE) {
        operation = DML_UPDATE;
    }
    super.doDML(operation, e);
}
```

Overriding the remove() and doDML() methods



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Example: Updating a Deleted Flag Instead of Deleting Rows

Your requirements may demand that rows are never physically deleted from a table, but rather only have the value of a DELETED column changed from N to Y to mark it as deleted. There are two method overrides required to alter an entity object's default behavior to achieve this effect:

1. Update a DELETED flag when a row is removed: Generate a Java class for your entity object and override the `remove()` method to set the deleted flag before calling the `super.remove()` method. The row will still be removed from the row set, but it will have the value of its DELETED flag modified to Y in the entity cache. It is important to set the attribute *before* calling `super.remove()`, because an attempt to set the attribute of a deleted row causes `DeadEntityAccessException`.
2. Force an update instead of a delete: Override the `doDML()` method and write code that conditionally changes the operation flag. When the operation flag equals `DML_DELETE`, your code will change it to `DML_UPDATE` instead.

With this overridden `doDML()` method in place to complement the overridden `remove()` method, any attempt to remove an entity row through any view object based on that entity will update the DELETED column instead of physically deleting the row. However, to prevent rows that are marked deleted from appearing in view object query results, you need to include `DELETED = 'N'` as part of the WHERE clause of each view object.

Overriding Base Class Methods Example: Eagerly Assigning Values from a Database Sequence

```
// In ProductEOImpl.java
import oracle.jbo.server.SequenceImpl;
// Default ProdId value from PRODUCTS_SEQ sequence at
protected void initDefaults() {
    super.initDefaults();
    SequenceImpl sequence = new
        SequenceImpl("PRODUCTS_SEQ",getDBTransaction());
    DBSequence dbseq = new
        DBSequence(sequence.getSequenceNumber());
    populateAttributeAsChanged(ProdId, dbseq);
}
```

Overriding the `initDefaults()` method



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Example: Eagerly Assigning Values from a Database Sequence

As you learned in the lesson titled “Declaratively Customizing Data Services,” you can declaratively assign the DBSequence type for primary key attributes whose values are populated by a database trigger at commit time. However, you can also eagerly allocate a sequence number to use as a default value for an attribute so that:

- The user can see its value
- The value does not change when the data is saved

To accomplish this, use the `SequenceImpl` helper class in the `oracle.jbo.server` package:

1. You override the `initDefaults()` method, as shown in the example in the slide. It shows code from the Java class of a `ProductEO` entity object.
2. Call the `initDefaults()` method of the parent class.
3. Create a new instance of the `SequenceImpl` object to get a handle to the database sequence.
4. Obtain the next number from the database sequence.
5. Assign the next sequence number to the `ProdId`.

The disadvantage of using a database sequence in this manner is that it results in more gaps in the numbers assigned to primary keys, because if the user does not commit the record, the sequence number is wasted.

Programmatically Modifying the Default Behavior of View Objects

You can:

- Change the WHERE or ORDER_BY clause programmatically
- Retain and reuse a row set
- Traverse view links



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Programmatically Modifying the Default Behavior of View Objects

In a similar fashion, you can modify the behavior of view objects programmatically. The next few slides discuss the Java classes for view objects and some examples of code that you can add.

Supporting View Object Java Classes

- `ViewObjectImpl`:
 - Is the view class
 - Provides methods to manage the row set
- `ViewDefImpl`:
 - Is the view definition class
 - Represents the whole view
 - Can be used to modify the view definition
- `ViewRowImpl`:
 - Is the view object row class
 - Is instantiated for each record returned by the query
 - Provides attribute accessors

```
OrdersViewImpl.java
package model.view;

import ...;
// ...
// --- File generated by Oracle ADF Business Components Design Time.
// --- Custom code may be added to this class.
// --- Warning: Do not modify method signatures of generated methods.
// ...
public class OrdersViewImpl extends ViewObjectImpl {
    /**This is the default constructor (do not remove).
     */
    public OrdersViewImpl() {
    }
}
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Supporting View Object Java Classes

There are three base Java classes that implement view objects:

- `ViewObjectImpl.java`: This is the view object class. All view object classes extend this class. The ADF Business Components technology uses this class to manage instances of each view. When you generate the view object class, you can also generate bind variable accessors (typesafe accessors for the view object's named bind variables) or custom Java data source methods.
- `ViewDefImpl`: This is the view definition class. At run time, one view definition is instantiated for each view. You can add methods to this class that are used by all view object instances.
- `ViewRowImpl.java`: This is the view row class. When you generate the view row class, you can also generate:
 - Accessors: Typesafe accessors for the view row's attributes
 - Expose Accessors to the Client: Turns all accessors into client methods

When you choose to generate Java classes for view objects, the classes that are generated extend the base classes. The classes that you generate use the entity name, followed by `ViewImpl`, `ViewRowImpl`, or `ViewDefImpl` (such as `OrdersViewImpl.java`).

Examining View Object Methods

The screenshot shows a Java Options Dialog window for the `ViewObjectImpl` class. The title bar reads "ViewObjectImpl (Oracle Fusion Middleware Java API Reference for Oracle ADF Model)". The menu bar includes "Overview Package Class Tree Deprecated Index Help". The right side of the window displays the "Oracle Fusion Middleware Java API Reference for Oracle ADF Model" header with "11g Release 1 (11.1.2.0)" and "E10653-03". Below the header, there are links for "PREV CLASS" and "NEXT CLASS", "FRAMES", "NO FRAMES", "ALL CLASSES", "SUMMARY: NESTED", "FIELD", "CONSTR", and "METHOD". The main content area shows the `ViewObjectImpl` class definition under the package `oracle.jbo.server`. It lists the inheritance path: `java.lang.Object`, `oracle.common.NamedObjectImpl`, `oracle.jbo.server.NamedObjectImpl`, `oracle.jbo.server.ComponentObjectImpl`, and finally `oracle.jbo.server.ViewObjectImpl`. It also lists "All Implemented Interfaces" which include various Oracle-specific interfaces like `ViewCriteriaClauseBuilder`, `ViewCriteriaManagerOwner`, etc. A yellow box highlights the text "Javadoc for the ViewObjectImpl class". At the bottom, it says "The implementation of the `ViewObject` interface, the middle-tier class that manages database queries and the view rows that result from executing queries." and "A View Object manages a view row set (`ViewRowSetImpl`). When the application calls a `RowSet` method

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Examining View Object Methods

Because you can access view objects directly in client code, it may be useful to review the methods that are available for view objects.

When you generate the view object class for a view object, it extends `ViewObjectImpl`, which indirectly (through the `ViewObject` interface) implements the `RowSet` interface.

`Rowset` is an interface representing a scrollable range of rows, of which one may be designated as the current row. Some of its methods include the following:

- `createRowSetIterator(java.lang.String name)`: Creates an iterator for the row set
- `executeQuery()`: Executes the view object's query
- `getApplicationModule()`: Gets the row set's Application Module
- `getEstimatedRowCount()`: Counts the number of rows in the collection defined by the view object's query
- `getMasterRowSetIterators()`: Returns all controlling masters of this row set
- `getName()`: Gets the row set's name
- `getViewObject()`: Gets the view object that contains the row set
- `getWhereClauseParams()`: Gets the bind-variable values to be used with the view object's query condition

Examining View Object Methods (continued)

- `setWhereClauseParam(int index, java.lang.Object value):` Specifies a single bind-variable value to use with the view object's query condition, and executes the query
- `setWhereClauseParams(java.lang.Object[] values):` Specifies the bind-variable values to use with the view object's query condition, and executes the query

The RowSet interface implements the RowIterator interface, which enables access to the row set. Some of its methods include the following:

- `createRow():` Creates a new Row object, but does not insert it into the row set
- `findByEntity(int eRowHandle, int maxNumOfRows):` Finds and returns view rows that use the entity row, identified by the entity row handle, eRowHandle
- `findByPrimaryKey(Key key, int maxNumOfRows):` Finds and returns view rows that match the specified key
- `first():` Designates the first row of the row set as the current row
- `getRow(Key key):` Accesses a row through a unique key
- `hasNext():` Tests for the existence of a row after the current row
- `hasPrevious():` Tests for the existence of a row before the current row
- `insertRow(Row row):` Adds a row to the row set, before the current row
- `last():` Designates the last row of the row set as the current row
- `next():` Steps forward, designating the next row as the current row
- `removeCurrentRow():` Removes the current Row object from the row set
- `previous():` Steps backward, designating the previous row as the current row
- `reset():` Clears the "current row" designation and places the iterator in the slot before the first row
- `setCurrentRow(Row row):` Designates a given row as the current row

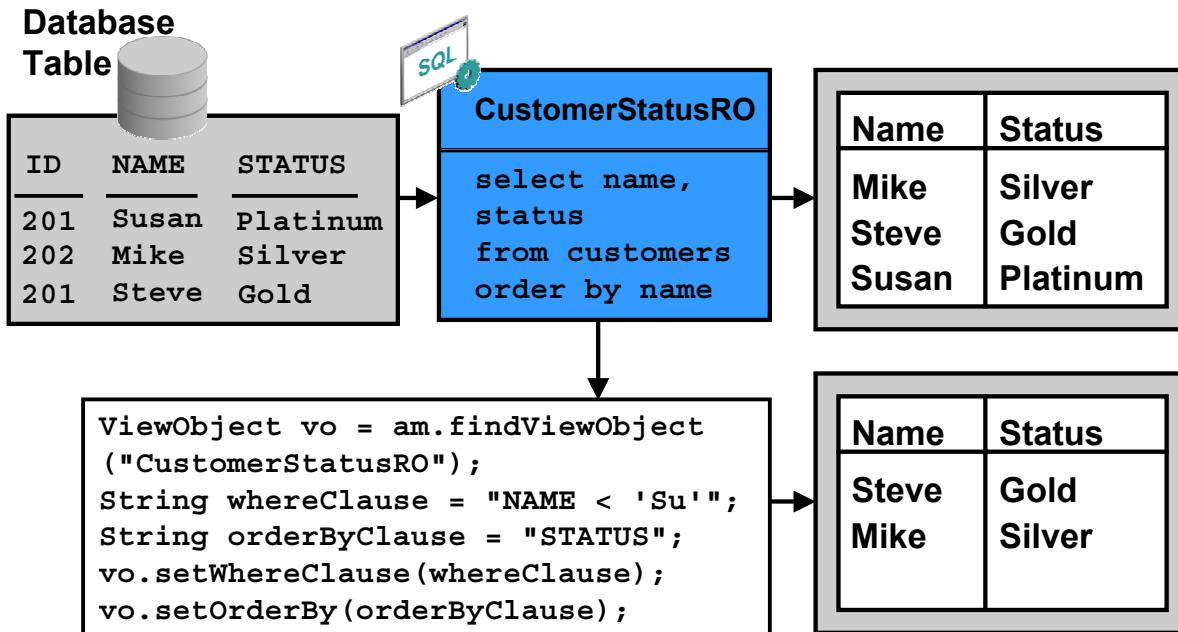
In addition to the RowSet and RowIterator interfaces, ViewObjectImpl implements the ViewObject interface, containing the following methods:

- `applyViewCriteria(ViewCriteria criteria):` Applies the view criteria to this view object
- `createRowSet(java.lang.String name):` Creates and names a row set for the view object
- `createViewCriteria():` Creates a "Query by Example" view criteria object
- `findViewLinkAccessor(ViewLink vl):` Finds the view link accessor attribute
- `getOrderByClause():` Retrieves the current ORDER BY clause of the view object's query statement
- `getQuery():` Retrieves the view object's query statement
- `getViewCriteria():` Gets this view object's view criteria
- `getViewLinkNames():` Retrieves the names of the view object's View Links
- `getWhereClause():` Retrieves the current WHERE clause of the view object's query statement
- `isReadOnly():` Tests if the view object is read-only
- `setOrderByClause(java.lang.String expr):` Sets the ORDER BY clause of the view object's query statement
- `setWhereClause(java.lang.String cond):` Sets the WHERE clause of the view object's query statement

Examining View Object Methods (continued)

The `ViewObjectImpl` class also implements some listener interfaces and contains many methods of its own, so there are several opportunities for programmatic manipulation. For a complete description of all methods that you can use for a view object, see Javadoc.

Changing View Object WHERE or ORDER BY Clause at Run Time



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Changing View Object WHERE or ORDER BY Clause at Run Time

You can change the WHERE and ORDER BY clauses of a query at run time by calling `setWhereClause()` or `setOrderByClause()` on the view object. Although `setWhereClause()` is the easiest way to change a view object's query, it often is not the most efficient. If you can, you should use parameterized WHERE clauses or structured WHERE clauses.

To change the WHERE clause of a query at run time, perform the following steps:

1. Create a String containing the new WHERE clause. Do not include the word WHERE. If you want to remove the WHERE clause entirely, use null. For example, either of the following could be appropriate definitions:
 - String whereClause = "ORDER_TOTAL > 500";
 - String whereClause = null;
2. Pass this string into `setWhereClause()`. For example, if `ordVO` is a variable containing the view object instance to be changed, you would call:
`ordVO.setWhereClause(whereClause);`
3. If there is an existing design-time WHERE clause, using `setWhereClause()` adds to it, further restricting the query results.

Changing View Object WHERE or ORDER BY Clause at Run Time (continued)

To change the ORDER BY clause of a query at run time, perform the following steps:

1. Create a String containing the new ORDER BY clause. Do not include the words ORDER BY. If you want to remove the ORDER BY clause entirely, use null. For example, either of the following could be appropriate definitions:
 - String orderByClause = "ORDER_TOTAL" ;
 - String orderByClause = null ;
2. Pass this string into `setOrderByClause()`. For example, if `OrdVO` is a variable containing the view object instance to be changed, you would call:
`ordVO.setOrderByClause(orderByClause) ;`

Using Named Bind Variables at Run Time

Assigning values to named bind variables at run time:

```
ViewObject vo = am.findViewObject("PersonView1");
vo.setNamedWhereClauseParam("TheName", "alex%");
vo.executeQuery();
```

Adding named bind variables at run time:

```
vo.setWhereClause("person_type_code = :PersonType");
vo.defineNamedWhereClauseParam("PersonType", null, null);
vo.setNamedWhereClauseParam("PersonType", "STAFF");
// execute the query, process the results, and then later...
vo.setNamedWhereClauseParam("PersonType", "CUST");
// execute the query, process the results
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Named Bind Variables at Run Time

You may have named bind variables in your view object whose value you want to assign at run time. You can use the `setNamedWhereClauseParam()` method to assign a value to any named bind variable, as shown in the first example in the slide.

If there is no design-time bind variable defined, you can add a bind variable at run time and assign its value programmatically. The second example in the slide shows adding a bind variable to a view object's WHERE clause and then defining the bind variable by using the `defineNamedWhereClauseParam()` method. In this example, the value is first set to STAFF, so records for employees are queried and processed. Next, the value for the bind variable is reset to CUST, and then customer records are queried and processed.

Programmatically Retaining and Reusing a View Link Accessor Row Set

To retain and reuse a view link accessor row set programmatically:

1. Generate a Java class for the source view object.
2. Override the `create()` method.
3. Add `setViewLinkAccessorRetained(true)`.

```
public class OrderVOImpl extends ViewObjectImpl {  
    public OrdersViewImpl() {  
    }  
    @Override  
    protected void create() {  
        super.create();  
        setViewLinkAccessorRetained(true);  
    }  
}
```

4. Call `reset()` each time you reuse the `RowSet` object.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Retaining and Reusing a Row Set

In the lesson titled “Declaratively Customizing Data Services,” you learned how to declaratively retain and reuse a view link accessor row set and why it may be advisable to do so. It is also possible to perform this caching of the row set programmatically.

To use the view link accessor retention feature, perform the following steps:

1. Generate a Java class for your view object.
2. In that Java class, override the `create()` method. (The optional `@Override` annotation tells the compiler that a superclass method is being overridden; if not done correctly, a compile error occurs.)
3. Add a line after `super.create()` that calls the `setViewLinkAccessorRetained()` method, passing `true` as the parameter

Because this is a method of the view definition class `ViewDefImpl`, this affects all view link accessor attributes for that view object. When this feature is enabled for a view object, because the view link accessor row set is not re-created each time, the current row of its default row set iterator is also retained as a side effect.

Because of this, before calling code that iterates through the row set, your code needs to explicitly call the `reset()` method on the row set that you retrieve from the view link accessor. This resets the current row in its default row set iterator back to just before the first row.

Traversing Links

- Links may be traversed in either direction.
- `ViewRowImpl.java` contains a method to get the associated row iterator.
- For example, `OrdersViewRowImpl` contains the method:

```
public oracle.jbo.RowIterator getOrderItemsView ()
```

- Use the methods of `RowIterator` to move from row to row and to get individual attribute values.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Traversing Links

Traversing a view link is similar to traversing an association. You use the accessor method in the source (master) view object to get the `RowIterator` object that contains all the associated rows in the destination (detail) view object. You can also traverse a view link from destination (detail) to source (master).

Similar to the example in the slide, the `OrdersViewRowImpl` class contains a method to return a single row containing `OrdersView`. Using this method, you can determine and manipulate detail information about the `Order` to which the current `OrderItem` belongs.

By default, a view link is a one-way relationship that enables the current row of the source (master) to access a set of related rows in the destination (detail) view object. This is accomplished because, by default, an accessor is generated only in the master view object. On the View Link Properties page of the Create View Link Wizard, you can change the default behavior by generating an accessor in the detail view object.

Traversing Links (continued)

At run time, the `getAttribute()` method on a `Row` enables you to access by name the value of any attribute of a row in the view object's result set. The view link accessor behaves like an additional attribute in the current row of the source view object, so you can use the same `getAttribute()` method to retrieve its value. The only practical difference between a regular view attribute and a view link accessor attribute is its data type. Whereas a regular view attribute typically has a scalar data type with a value such as 303 or ahunold, the value of a view link accessor attribute is a row set of zero or more correlated detail rows. Assuming that `curOrder` is a `Row` from some instance of the `Orders` view object, you can write a line of code to retrieve the detail row set of order items:

```
RowSet lineItems = (RowSet) curOrder.getAttribute("LineItems");
```

The `RowSet` is closed when you iterate and retrieve the last row from it. If you stop the iteration before retrieving the last row, you can programmatically close the `RowSet` if you want to release all the rows it contains. You can do this using the `closeRowSet()` API.

Note: If you generate the custom Java class for your view row, the type of the view link accessor will be `RowIterator`. At run time, the return value will always be a `RowSet`, so it is safe to cast the view link attribute value to a `RowSet`.

Application Module Files

- Created by default:
 - `<AppMod>.xml`: Includes detailed metadata about the view objects included
 - `bc4j.xcfg`: Contains all the configuration and connection details
- Supporting Java classes:
 - `<AppMod>Impl.java`: Contains all the methods and behaviors that affect each application module instance
 - `<AppMod>Def.java`: Contains methods to be used by all instances of the application module



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

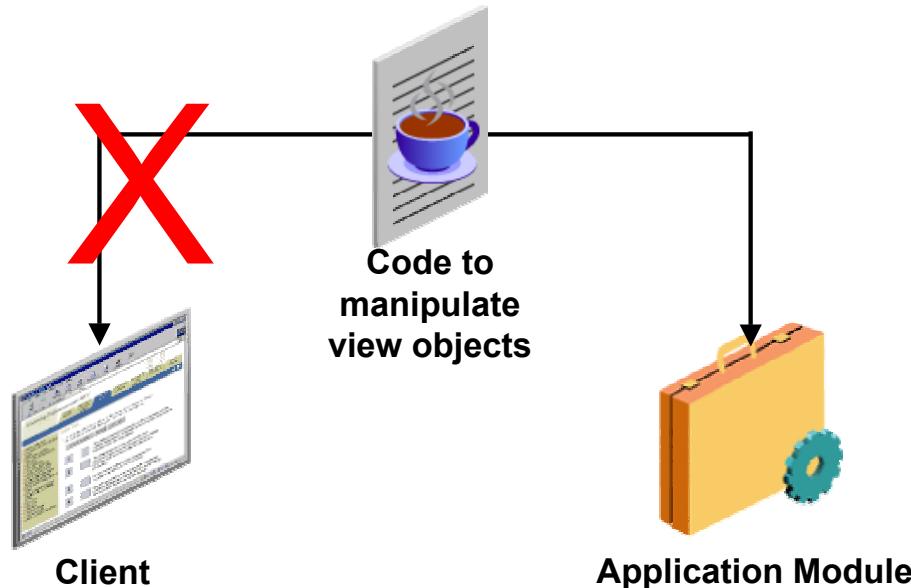
Application Module Files

Four main files control the behavior of the application module.

- `<AppMod>.xml`: The metadata file that contains all the definitions for every object included in the module. It includes all the attribute definitions needed during design time for all the view objects and view links in the module. This is the file that you edit declaratively when editing the application module.
- `bc4j.xcfg`: Contains all the configuration information including the JDeveloper connection details. This file is used by the BC Tester while running the module. You edit this file declaratively by using project properties (Business Components) to edit connection details, and by right-clicking the application module and selecting Configurations to edit configuration details.
- `<AppMod>Impl.java`: Contains all the methods and behaviors for the application module for which you have defined code. You need to generate this file only if you have to programmatically modify the behavior of the application module.
- `<AppMod>Def.java`: The application module definition class. At run time, one application module definition is instantiated for each application module. You can add methods to this class that are used by all instances, but this file is not used as often as `<AppMod>Impl.java`.

Centralizing Implementation Details

Best practice is to place code in application module service methods rather than in multiple clients.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Centralizing Implementation Details

Client code can use the `ApplicationModule`, `ViewObject`, `RowSet`, and `Row` interfaces in the `oracle.jbo` package to work directly with any view object in the data model.

However, just because you can programmatically manipulate view objects any way you want to in client code does not mean that doing so is always a best practice.

If you need to write code that is related to setting up or manipulating the data model, it is recommended that you encapsulate such code inside methods of the application module.

Whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality, you should encapsulate the details by writing a method in your application module's Java class. This includes, but is not limited to, code that:

- Configures view object properties to query the correct data to display
- Iterates over view object rows to return an aggregate calculation
- Performs any kind of multistep procedural logic with one or more view objects

Centralizing Implementation Details (continued)

By centralizing these implementation details in your application module, you gain the following benefits:

- You make the intent of your code more clear to clients.
- You allow multiple client pages to easily call the same code if needed.
- You simplify regression testing your complete business service functionality.
- You keep the option open to improve your implementation without affecting clients.
- You enable declarative invocation of logical business functionality in your pages.

Adding Service Methods to an Application Module

Service methods:

- Are useful for:
 - Code that is not dependent on a specific view
 - Performing operations across view object instances
 - Managing transactions
 - Dynamically changing the data model
- Can be called from the client, requiring very little data manipulation in the client itself
- Are implemented in the application module's class
- Are added by:
 - Adding code to the Java class
 - Publishing to the client interface



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Adding Service Methods to an Application Module

Service methods are methods that are specific to an application module. These methods may be independent of any view in the application. For example, you may need a method that determines the number of current orders for a specific item in inventory. That method can be used from any client without needing to instantiate an inventory or order view.

You can also add code that affects the behavior of a particular view object in the application module; for example, you can change the WHERE or ORDER BY clause of a view object. You would do this if you were reusing a view object in multiple application modules and wanted the view object to behave differently in different application modules.

Adding Service Methods to an Application Module (continued)

Example: Deciding When to Add Custom Code to an Application Module

Two application modules, OrderManagementApp and CustomersApp, both have CustomersView in their data model. CustomersApp requires code to perform analysis of customers' purchasing history; OrderManagementApp does not need this information. Adding this code to CustomersView would add size and complexity to the view object for the sake of functionality that will never be used by OrderManagementApp. Therefore, you should add your custom code to the CustomersApp application module.

To add a service method, you first enable a Java class, then add code to it, and finally publish the service method. You have already learned how to enable the Java class; the second and third steps are detailed in the following slides.

Coding the Service Method

Add the Java code for the method to the `<Name>AMImpl.java` file.

```
public class PersonAMImpl extends ApplicationModuleImpl {  
    /**  
     * This is the default constructor (do not remove).  
     */  
    public PersonAMImpl() {  
    }  
  
    public String retrievePersonById(Number personId) {  
        EntityDefImpl personsReqDef = PersonEOImpl.getDefinitionObject();  
        Key personReqKey = PersonEOImpl.createPrimaryKey( personId );  
        PersonEOImpl persons =  
            (PersonEOImpl)personsReqDef.findByPrimaryKey(getDBTransaction(),personReqKey);  
  
        return persons.getLastName();  
    }  
}
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Coding the Service Method

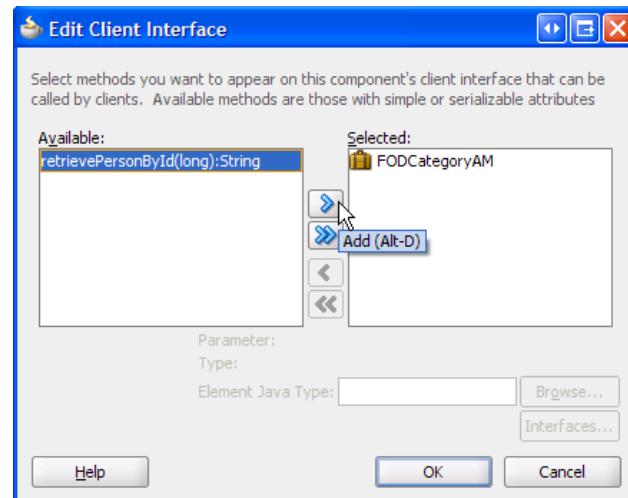
Add the Java code for the custom method to the `<name>AMImpl.java` file.

The example in the slide shows a `retrievePersonById()` method in the `PersonAMImpl.java` class for the `PersonAM` application module. It uses the `getDefinitionObject()` method of the `PersonEOImpl` entity object class to access its related entity definition, uses the `createPrimaryKey()` method on the entity object class to create an appropriate Key object to look up the person, and then uses the `findByPrimaryKey()` method on the entity definition to find the entity row in the entity cache. It returns the `Lastname` attribute of the `PersonEOImpl` object.

Publishing the Service Method

To publish the service method to clients:

1. Open the Application Module Editor and click the Java finger tab.
2. Click the Pencil icon to edit the Client Interface.
3. Shuttle the method you want to publish to the Selected list.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Publishing the Service Method

When you add a public custom method to your application module class, if you want clients to be able to invoke it, you need to include the method on the application module's client interface. To do this, perform the following steps:

1. Click the Java finger tab in the Application Module Editor.
2. Click the Edit application module client interface icon (the Pencil icon).
3. Shuttle the method you want to publish from the Available pane to the Selected pane, and then click OK.

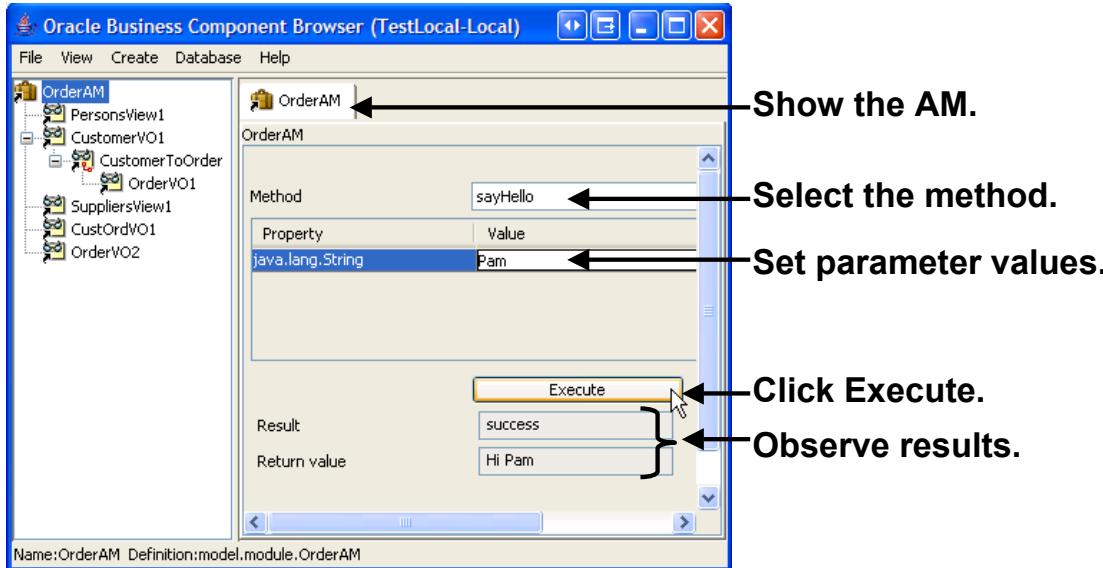
JDeveloper creates a Java interface with the same name as the application module in the common subpackage of the package where your application module resides. The interface extends the base `ApplicationModule` interface in the `oracle.jbo` package, reflecting that a client can access all the functionality that your application module inherits from the `ApplicationModuleImpl` class. The interface created by the example in the slide includes the `retrievePersonById()` method that was added to the client interface of the application module. Each time a method is added to or removed from the Selected list in the Edit Client Interface dialog box, the corresponding service interface page is updated automatically.

Publishing the Service Method (continued)

JDeveloper also generates a client proxy class that is used when you deploy your application module for access by a remote client. All the generated files are displayed in the Navigator under the application module's node.

Note: If your method does not appear in the Available list in the Edit Client Interface dialog box, it may be that it contravenes one or more of the rules for inclusion. Only public methods with parameter and return types that are primitives or that implement the `Serializable` interface are included in the list.

Testing Service Methods in the Business Components Browser



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Testing Service Methods in the Business Components Browser

You can use the Business Components Browser to test service methods. Double-click the application module in the tree at the left to show it. Select the method to test from the Method drop-down list, and supply values for any required parameters. When you click Execute, the results, including a return value if any, are displayed in the lower part of the window.

Accessing a Transaction

- Transaction and DBTransaction are interfaces that define database transactions.
- Use the methods in these interfaces to access an application module's transaction. For example:

```
ApplicationModuleImpl am; ...
// Commit the transaction
am.getTransaction().commit();
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Accessing a Transaction

You access an application module's transaction context through the Transaction and DBTransaction interfaces. You can access the methods in these interfaces through the getTransaction() and getDBTransaction() methods in the ApplicationModuleImpl class.

Some useful methods of the Transaction interface with regard to application modules are:

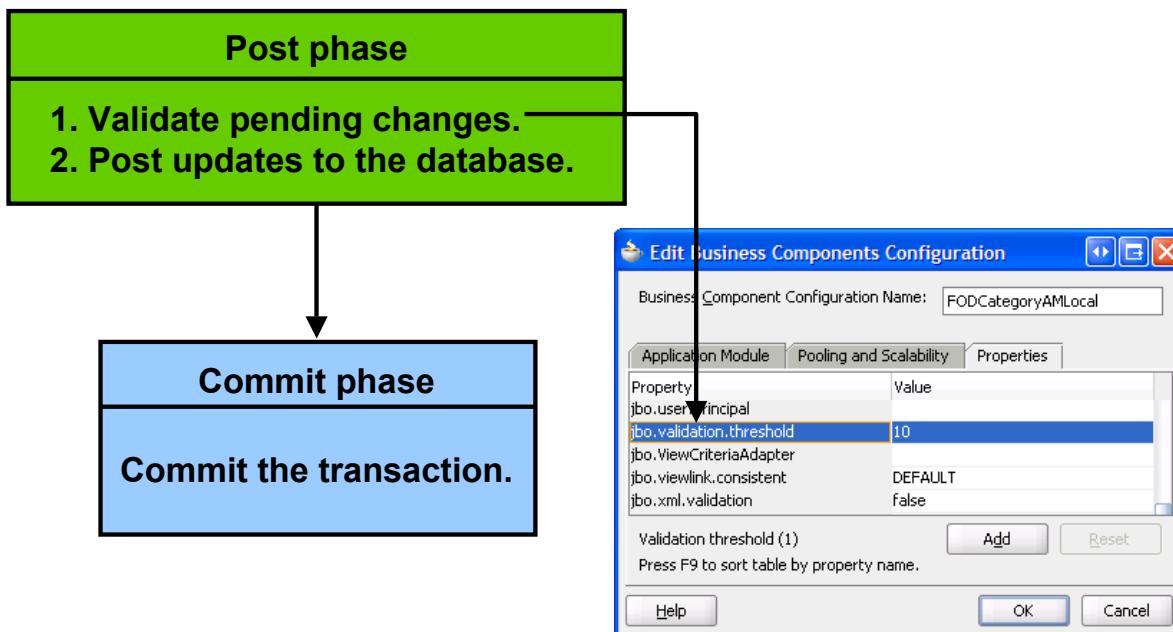
- **commit**: Commits the transaction, and saves all the changes to the database. If the database connection is established, the transaction is implicitly started.
- **connect**: Attempts to establish a connection to the given database URL
- **disconnect**: Disconnects the server from the database
- **getLockingMode**: Gets the preferred locking mode for this transaction. In ADF BC, the locking mode defaults to LOCK_PESSIMISTIC; however, Oracle Fusion applications use optimistic locking by default.
- **rollback**: Rolls back the transaction, and discards all the changes
- **setLockingMode**: Sets the preferred locking mode for this transaction. Changing the locking mode affects only the subsequent locks that are placed.

Accessing a Transaction (continued)

The Transaction interface contains a standard set of methods that include `commit()`, `setLockingMode()`, and `getLockingMode()`. The DBTransaction interface extends Transaction. It contains some more specialized methods that deal with entity objects. Some of the methods are `findByPrimaryKey()`, `getSession()`, and `getEnvironment()`. For the most part, the Transaction interface provides the methods you will use most often.

JDeveloper's online Help contains the Javadoc for `oracle.jbo.Transaction` and `oracle.jbo.server.DBTransaction`. Refer to the Javadoc for a complete list of available methods.

Committing Transactions



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Committing Transactions

Post Phase: The framework validates all entity objects that are marked as invalid. (Any new or updated entity object is automatically marked as invalid.) This process includes applying all validation rules that you have added in the Entity Object Wizard, as well as executing any custom code that you have added to the entity's `validateEntity()` method.

Commit Phase: The transaction is committed, and the updated data is available to other application modules.

Avoiding Infinite Validation Cycles

Because custom validation code itself can change an entity object's data, the validation process in the post phase is repeated up to a limit specified by the transaction-level Validation Threshold setting, until no entity objects are marked as invalid. Then the updates are posted to the database. If invalid entities remain after the threshold is reached, an exception is generated. To set the Validation Threshold property, you can edit the application module by performing the following steps:

1. Right-click the application module and select the Configurations panel.
2. Select a configuration and click Edit.
3. Click the Properties tab.
4. If the `jbo.validation.threshold` property exists, you can edit its value; otherwise, click Add to manually add the `jbo.validation.threshold` string and set its value.

Customizing the Post Phase

- Override the entity object's `postChanges()` method.
- Example: For a deleted entity, mark the database row instead of deleting the data.

```
public void postChanges(TransactionEvent e) {  
    if (getPostState() == Entity.STATUS_DELETED) {  
        // Custom code to mark database rows, such as  
        // by updating a Deleted flag in the record  
    }  
    else {  
        super.postChanges(e);  
    }  
}
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Customizing the Post Phase

You can customize the behavior of the post operation by overriding `EntityImpl.postChanges()`, which is the method that is used, by default, to perform the post operation. The slide shows an example of this, with code that intercepts a delete action and only marks the deleted rows as deleted, without actually removing them from the table. You could use this technique if you want to preserve history of deleted rows.

The code in the example calls `getPostState()` to determine the entity's *post state*. The post state is the state of the entity object's data relative to the transaction's corresponding database data. The `Entity` interface contains a number of constants that define the possible values for an entity's post state. The post state is set to `STATUS_UNMODIFIED` when the data is posted to the database.

Customizing the Commit Phase

- Implement a TransactionListener.
- Implement `beforeCommit()` and `afterCommit()`.
- Add your listener to the transaction's list of event subscribers.
- Alternatively, override the entity object's `beforeCommit()` or `afterCommit()` methods.

Example: In `PersonEOImpl`, print notification that record is committed:

```
@Override  
public void afterCommit(TransactionEvent transactionEvent) {  
    System.out.println("Record committed for " +  
        getFirstName() + " " + getLastName());  
    super.afterCommit(transactionEvent);  
}
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Customizing the Commit Phase

The `TransactionListener` interface defines `beforeCommit()` and `afterCommit()` methods. You can implement a transaction listener and add it to the transactions list of event listeners by calling the transaction's `addTransactionListener()` method. For example, in an entity object's code, call:

```
getTransaction().addTransactionListener(myTransListener);
```

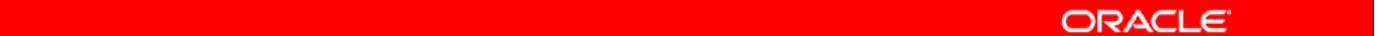
Your entity objects are also transaction listeners, and are already on your transaction's list of event listeners. Because of this, you can also override an entity object's `beforeCommit()` and `afterCommit()` methods. The simple example in the slide shows adding a printed notification that a record has been committed; however, you could perform any action, such as notifying an employee's manager by email that information about an employee has been updated.

Customizing Rollback

The procedure for customizing rollback behavior is similar to the procedure for customizing commit behavior. The `TransactionListener` interface also defines `beforeRollback()` and `afterRollback()` methods that are called before and after a rollback operation.

Using Entity Objects and Associations Programmatically

- Code to manipulate an entity object is typically placed in an application module class or the class of another entity object.
- Typical tasks that you may need to code:
 - Finding an entity object by primary key
 - Updating or removing an existing entity row
 - Creating a new entity row

The red bar spans most of the slide width, centered horizontally.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Entity Objects and Associations Programmatically

While external client programs can access an application module and work with any view object in its data model, by design neither UI-based nor programmatic clients work directly with entity objects. Any code you write that works programmatically with entity objects is typically placed in an application module class or in the class of another entity object.

There are many ways to use business components APIs to manipulate entity objects programmatically. The next few slides present some examples.

Finding an Entity Object by Primary Key

```
public String findOrderID(long orderId) {  
    String entityName = "oracle.fod.storefront.model.entity.OrderEO";  
    EntityDefImpl orderDef = EntityDefImpl.findDefObject(entityName);  
  
    1   Key orderKey = new Key(new Object[]{orderId});  
  
    2   EntityImpl order = orderDef.findByPrimaryKey(getDBTransaction(),  
        orderKey);  
    3   if (order != null) {  
        return (String)order.getAttribute("ShipToName");    }  
    4   else {      return null;    } }
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Finding an Entity Object by Primary Key

To find an entity object by primary key, follow these basic steps:

1. **Find the entity definition:** Obtain the entity definition object for the OrderEO entity by passing its fully qualified name to the static `findDefObject()` method on the `oracle.jbo.server.EntityDefImpl` class, a Java class that implements the entity definition for each entity object.
2. **Construct a key:** Build a `Key` object containing the primary key attribute that you want to look up. In this case, you are creating a key containing the single `orderId` value passed into the method as an argument.
3. **Find the entity object using the key:** Use the entity definition's `findByPrimaryKey()` method to find the entity object by key, passing in the current transaction object, which you can obtain from the application module by using its `getDBTransaction()` method. The concrete class that represents an entity object row is the `oracle.jbo.server.EntityImpl` class.
4. **Return some of its data to the caller:** You use the `getAttribute()` method of `EntityImpl` to return the value of the `ShipToName` attribute to the caller.

Updating or Removing an Existing Entity Row

```
public void updateEmpEmail(long empId, String newEmail) {  
    EntityImpl emp = retrieveEmployeeById(empId);  
    if (emp != null) {  
        emp.setAttribute("Email",newEmail);  
        try {  
            getDBTransaction().commit();  
        } catch (JboException ex) {  
            getDBTransaction().rollback();  
            throw ex;        }    } }
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Updating or Removing an Existing Entity Row

You can add a method to update or remove a row. For example, you can add an application module method to update an employee row by performing the following steps:

1. Find the Employee by ID: This example assumes that you have defined a helper method named `retrieveEmployeeById()` that retrieves the `EmployeeEO` entity object by its primary key.
2. Set one or more attributes to new values: Use the `setAttribute()` method of the `EntityImpl` class to update the value of the `Email` attribute to the new value passed in.
3. Commit the transaction: Use the application module's `getDBTransaction()` method to access the current transaction object and call its `commit()` method to commit the transaction.

The example for removing an entity row would be the same as this, except that after finding the existing entity, you would use the following line instead to remove the entity before committing the transaction:

```
emp.remove();
```

Creating a New Entity Row

```
public long createProduct(String name, String description) {  
    String entityName = "oracle.fod.storefront.model.entity.ProductEO";  
    1 EntityDefImpl productDef = EntityDefImpl.findDefObject(entityName);  
  
    2 EntityImpl newProduct =  
        productDef.createInstance2(getDBTransaction(), null);  
    3 newProduct.setAttribute("Name", name);  
    4 newProduct.setAttribute("Description", description);  
    try {  
        getDBTransaction().commit();  
    catch (JboException ex) {  
        throw ex; }  
    5 DBSequence newIdAssigned =  
        (DBSequence) newProduct.getAttribute("ProdId");  
    return newIdAssigned.getSequenceNumber().longValue();  
}
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating a New Entity Row

In addition to using the entity definition for finding existing entity rows, you can also use it to create new ones. For example, you could write a `createProduct()` method to accept the name and description of a new product, and return the new product ID assigned to it. Assume that the `ProdId` attribute of the `Product` entity object has been updated to have the `DBSequence` type, so that its value is automatically refreshed to reflect the value that the `ASSIGN_PRODUCT_ID` trigger on the `PRODUCTS` table assigns to it from the `PRODUCTS_SEQ` sequence in the database schema.

The example follows these steps:

1. **Find the entity definition:** Use `EntityDefImpl.findDefObject()` to find the entity definition for the `ProductEO` entity.
2. **Create a new instance:** Use the `createInstance2()` method on the entity definition to create a new instance of the entity object. (The method name really has a 2 at the end. The regular `createInstance()` method has protected access and is designed to be customized by developers. The second argument of `AttributeList` type is used to supply attribute values that must be supplied at create time; it is not used to initialize the values of all attributes found in the list. For example, when creating a new instance of a

Creating a New Entity Row (continued)

composed child entity row using this API, you must supply the value of a composing parent entity's foreign key attribute in the `AttributeList` object passed as the second argument. Failure to do so results in an `InvalidOwnerException`.

3. **Set attribute values:** Use the `setAttribute()` method on the entity object to assign values for the `Name` and `Description` attributes in the new entity row.
4. **Commit the transaction:** Call `commit()` on the current transaction object to commit the transaction.
5. **Return the trigger-assigned product ID to the caller:** Use `getAttribute()` to retrieve the `ProdId` attribute as a `DBSequence`, and then call `getSequenceNumber().longValue()` to return the sequence number as a `long` value to the caller.

Using Client APIs

Client interfaces in the `oracle.jbo` package include:

- `ApplicationModule`
- `ViewObject`
- `Row`

(But NOT Entity)

The screenshot shows a Java API documentation page for the `oracle.jbo` package. The top navigation bar includes links for Overview, Package (which is selected), Class, Tree, Deprecated, Index, and Help. Below the navigation is a toolbar with PREV PACKAGE, NEXT PACKAGE, FRAMES, NO FRAMES, and All Classes. The main content area starts with a section for the `oracle.jbo` package, which contains a brief description: "Contains interfaces for client-side applications." It also lists "See:" and a link to "Description". A "Interface Summary" section follows, containing two entries:

Interface	Description
ApplicationModule	The interface for application modules.
ApplicationModuleHandle	A handle to an ApplicationModule instance.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

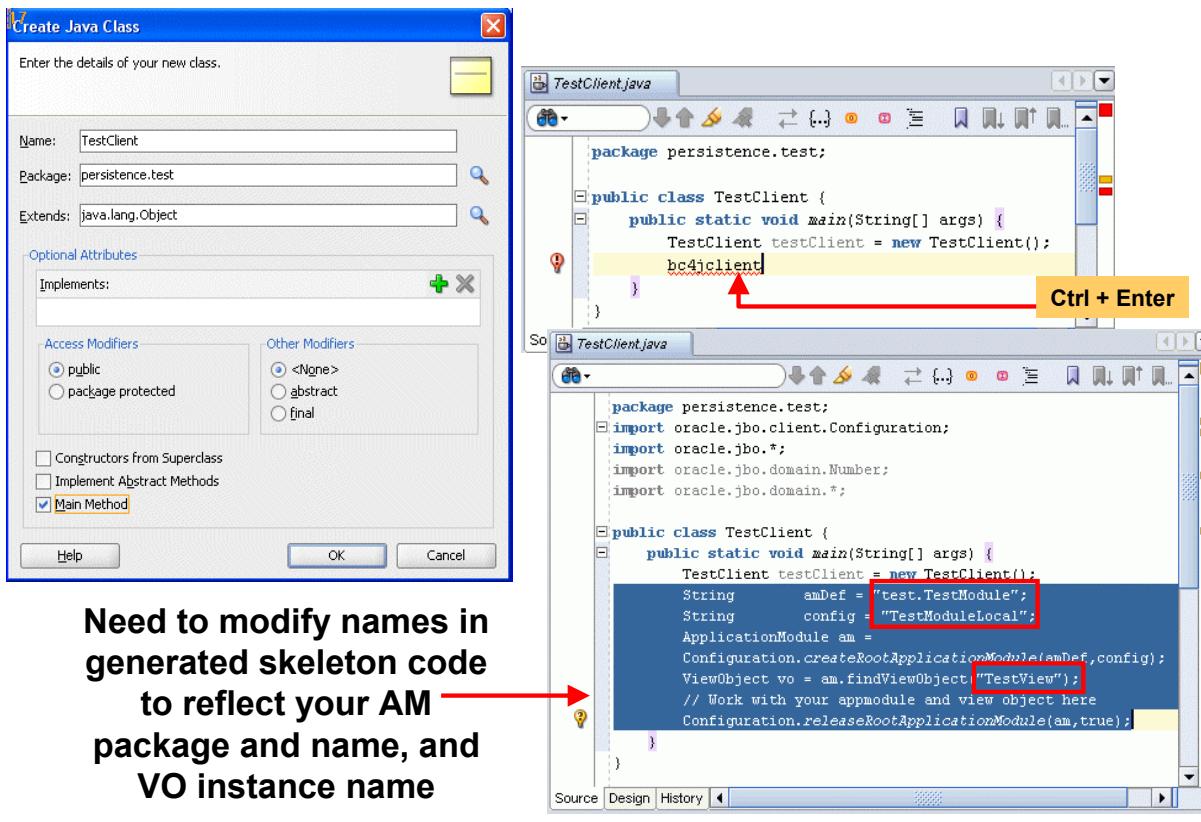
Using Client APIs

The Java interfaces of the `oracle.jbo` package provide a client-accessible API for your business service. This package intentionally does not contain an `Entity` interface, or any methods that would allow clients to directly work with entity objects. Instead, client code works with interfaces such as:

- `ApplicationModule`: To work with the application module
- `ViewObject`: To work with the view object
- `Row`: To work with the view rows

For a list of the most commonly used methods in the `oracle.jbo` interfaces, see Appendix E in the *Fusion Developer's Guide for ADF*.

Creating a Test Client



Need to modify names in generated skeleton code to reflect your AM package and name, and VO instance name

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating a Test Client

If you want to test custom code, you can create a test client program. Create a Java class by using the Create Java Class Wizard that you invoke from the New Gallery under the General category.

1. Enter a class name and a package.
2. In Optional Attributes, deselect the “Constructors from Superclass” and Implement Abstract Methods check boxes.
3. Select the Main Method check box.
4. Click OK to create the class.

The file opens in the editor to show you the skeleton code:

- Place the cursor on a blank line inside the body of the `main()` method and use the `bc4jclient` code template to create the few lines of necessary code. To use this predefined code template, enter the characters `bc4jclient`, and then press `Ctrl + Enter` to expand the code template.
- Adjust the values of the `amDef` and `config` variables to reflect the names of the application module definition and the configuration that you want to use, respectively, such as `module.FODCategoryAM` and `FODCategoryAMLocal`.
- Finally, change the view object instance name in the call to `findViewObject()` to be the one you want to work with. Specify the name exactly as it appears in the Data Model tree on the Data Model panel of the Application Module editor, such as `CategoryVO1`.

Using View Objects in Client Code

Examples of using view objects programmatically:

- Using query results programmatically
- Using view criteria to alter view object queries at run time
- Iterating master-detail hierarchy
- Finding a row and updating a foreign key value
- Creating a new row

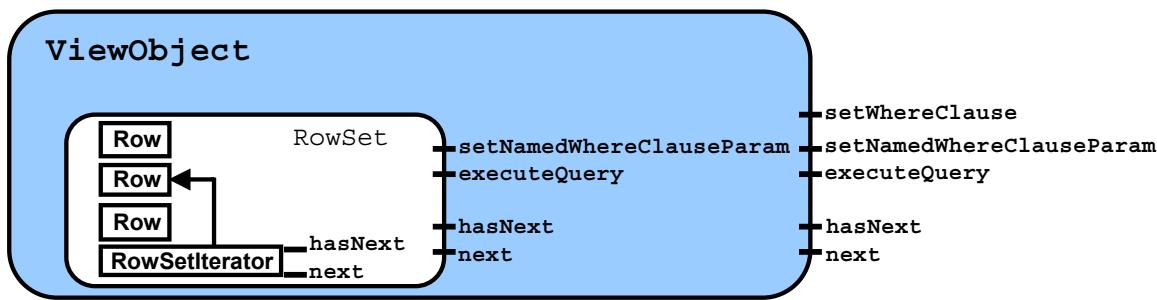


Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using View Objects in Client Code

From the point of view of a client accessing your application module's data model, the APIs to work with a read-only view object and an entity-based view object are identical. The key functional difference is that entity-based view objects enable the data in a view object to be fully updatable. The application module that contains the entity-based view objects defines the unit of work and manages the transaction.

Using Query Results Programmatically



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Query Results Programmatically

The `ViewObject` interface in the `oracle.jbo` package provides the methods to make quick work of any data-retrieval task.

Most of the time you work with only a single row set of results at a time for a view object, with a single iterator. The view object contains a default `RowSet`, which in turn contains a default `RowSetIterator`. The default `RowSetIterator` enables you to call the following methods directly on the `ViewObject` component itself, knowing that they apply automatically to its default row set:

- `executeQuery()`: Executes the view object's query and populates its row set of results
- `setWhereClause()`: Adds a dynamic predicate at run time to narrow a search
- `setNamedWhereClauseParam()`: Sets the value of a named bind variable
- `hasNext()`: Tests whether the row set iterator has reached the last row of results
- `next()`: Advances the row set iterator to the next row in the row set
- `getEstimatedRowCount()`: Counts the number of rows a view object's query would return

Using View Criteria Programmatically

```
ViewCriteria vc = custOrdVO.createViewCriteria();  
  
ViewCriteriaRow promotionRow = vc.createViewCriteriaRow();  
ViewCriteriaRow noPromRow = vc.createViewCriteriaRow();  
promotionRow.setAttribute("OrderTotal", "> 500");  
promotionRow.setAttribute("CreditLimit", "> 2500");  
promotionRow.setAttribute("DiscountId", "<> NULL");  
noPromRow.setAttribute("OrderTotal", "> 1000");  
noPromRow.setAttribute("CreditLimit", "> 5000");  
  
vc.addElement(promotionRow);  
vc.addElement(noPromRow);  
  
custOrdVO.applyViewCriteria(vc);  
  
custOrdVO.executeQuery();
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using View Criteria Programmatically

You also can create and use structured WHERE clauses programmatically. The example in the slide programmatically constructs view criteria that are similar to the view criteria constructed declaratively in the lesson titled “Declaratively Customizing Data Services.” The sample code does the following:

1. Instantiates the ViewCriteria class by calling `createViewCriteria()` on the view object instance.
2. For each set of criteria that you need, create a view criteria row by calling `createViewCriteriaRow()` on the ViewCriteria.
3. Set the requirements for each view criteria row by calling `setAttribute()` on ViewCriteriaRow. Pass to `setAttribute()` the name of the view object attribute and the SQL requirement to be applied to the attribute.
4. Add the rows to the view criteria by passing them to `addElement()` on ViewCriteria.
5. Apply the view criteria to the view object instance by passing ViewCriteria to `applyViewCriteria()`.
6. Execute the query on the view object to return only the rows that satisfy all the view criteria that you applied.

Using View Criteria Programmatically (continued)

At any time, you can do any of the following:

- Change the requirements in any row using `setAttribute()`.
- Add further rows to the criteria using `addElement()`.
- Remove rows from the criteria using `removeElement()`.
- Construct and apply entirely different view criteria.
- Nullify view criteria by passing `null` to `applyViewCriteria()`.

Iterating Master-Detail Hierarchy

```

public class TestClient {
    public static void main(String[] args) {
        TestClient testClient = new TestClient();
        String amDef = "oracle.hr.test.TestAM";
        String config = "TestAMLocal";
        ApplicationModule am =
        Configuration.createRootApplicationModule(amDef,config);
        ViewObject dept = am.findViewObject("DepartmentsView1");
        dept.executeQuery();
        while (dept.hasNext()) {
            Row department = dept.next();
            System.out.println("Department " +
                department.getAttribute("DepartmentId"));
            RowSet emps = (RowSet)department.getAttribute("EmployeesView");
            while (emps.hasNext()) {
                Row emp = emps.next();
                System.out.println(" Employee Name: " +
                    emp.getAttribute("LastName")); }}
```

Configuration.releaseRootApplicationModule(am,true);}}

1
2
3
4
5
6
7

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Iterating Master-Detail Hierarchy

The sample code, placed in a test client, iterates over all departments and their associated employees in a master-detail hierarchy. The beginning of the code sets up standard test client code, which you learn to do shortly. The remaining code performs the following actions:

1. Finds the DepartmentsView1 view object
2. Executes the view object query
3. Iterates over the resulting rows
4. Prints out the department name
5. Gets the related RowSet of employees by using the view link accessor
6. Iterates over the employees rows
7. Prints out the last name of each employee

The beginning of the resulting output looks like this:

```

Department 10
Employee Name: Whalen
Department 20
Employee Name: Hartstein
Employee Name: Fay
...
```

Finding a Row and Updating a Foreign Key Value

```

public class TestClient {
    public static void main(String[] args) {
        TestClient testClient = new TestClient();
        String amDef = "oracle.hr.test.TestAM";
        String config = "TestAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef,config);
        ViewObject vo = am.findViewObject("EmployeesView1");
        Key empKey = new Key(new Object[]{101});
        Row[] empsFound = vo.findByKey(empKey,1);
        if (empsFound != null && empsFound.length > 0) {
            Row emp = empsFound[0];
            System.out.println("Employee Name: " + emp.getAttribute("LastName"));
            System.out.println("Employee is in department " +
                emp.getAttribute("DepartmentId"));
            emp.setAttribute("DepartmentId",60);
            System.out.println("Employee reassigned to department " +
                emp.getAttribute("DepartmentId"));
            am.getTransaction().rollback();
            System.out.println("Transaction cancelled");
        }
        Configuration.releaseRootApplicationModule(am,true); }}
```

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Finding a Row and Updating a Foreign Key Value

This sample code looks up an employee and reassigns the employee to a different department. It uses the following steps:

1. Finds the employees view object
2. Constructs a key to find employee number 101
3. Finds the row that matches this key
4. Prints the employee's name
5. Prints the current value of the employee's department
6. Assigns the employee to a different department
7. Prints the new value of the employee's department
8. Rolls back the transaction

Here is the output of this code:

```

Employee Name: Kochhar
Employee is in department 90
Employee reassigned to department 60
Transaction cancelled
```

Creating a New Row

The screenshot shows the Oracle JDeveloper interface. On the left, the code editor displays `TestClient.java` with the following code:

```

public class TestClient {
    public static void main(String[] args) {
        TestClient testClient = new TestClient();
        String amDef = "oracle.hr.test.TestAM";
        String config = "TestAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        ViewObject vo = am.findViewObject("EmployeesView1");
        Row newEmp = vo.createRow();
        vo.insertRow(newEmp);
        newEmp.setAttribute("EmployeeId", 999);
        newEmp.setAttribute("FirstName", "Pam");
        newEmp.setAttribute("LastName", "Gamer");
        Date now = new Date();
        newEmp.setAttribute("HireDate", now);
        newEmp.setAttribute("JobId", "IT_PROG");
        newEmp.setAttribute("Email", "pgamer@mymail.com");
        newEmp.setAttribute("DepartmentId", 60);
        newEmp.setAttribute("ManagerId", 103);
        am.getTransaction().commit();
        System.out.println("Added employee " +
            newEmp.getAttribute("EmployeeId") + " to Department " +
            newEmp.getAttribute("DepartmentId"));
        Configuration.releaseRootApplicationModule(am, true);
    }
}

```

On the right, the Navigator pane shows the `EMPLOYEES` view object with the following data:

EMPLOYEE ID	FIRST NAME	LAST NAME	EMAIL	PHONE NUMBER	HIRE DATE	JOB ID	SALARY	C
107	206	William	Gietz	WGI...	515.123.8181	07-JUN-94	AC_ACCOUNT	8300
108	999	Pam	Gamer	pga...	(null)	01-JAN-70	IT_PROG	(null)

Step numbers 1 through 5 are overlaid on the code editor area, corresponding to the numbered steps in the process.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating a New Row

To create a new employee, you perform the following steps:

1. Find the employees view object.
2. Create a new row and insert it into the row set.
3. Set the value of some required attributes in the row.
4. Commit the transaction.
5. Display a message.

The output is:

Added employee 999 to Department 60

You can view the table by browsing it from the Navigator to see that the record was added.

Summary

In this lesson, you should have learned how to:

- Generate Java classes for business components
- Override class methods
- Implement programmatic modifications
- Add service methods to an application module
- Create a test client
- Use business component client APIs



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 7 Overview: Programmatically Modifying Business Components

This practice covers the following topics:

- Adding Code to Entity Objects
- Programmatically Assigning a Database Sequence
- Populating History Columns When There Is No Logged-in User
- Creating and Running a Test Client
- Creating an Application Module Base Class
- Editing an Application Module to Extend the New Base Class
- Adding and Exposing Service Methods
- Restricting the Shopping Cart Query



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 7 Overview: Programmatically Modifying Business Components

In this set of practices, you add programmatic functionality to your business components.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Osi S.R.L. use only

8

Validating User Input

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Describe the types of validation available to ADF BC applications
- Decide which validation options are appropriate for different validation requirements
- Use the declarative validation options provided in JDeveloper
- Create programmatic validation
- Use domains in validation



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson covers data validation options for ADF BC applications. You learn when each option is most suitable. You create and use validation methods and declarative validation, and you create and use a domain for validation.

Validation Options for ADF BC Applications

- Validation options
 - Business services
 - ADF BC
 - Metadata (declarative validation)
 - Java (programmatic validation)
 - Database (PL/SQL)
 - User interface
- Managing the various validation options



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Validation Options for ADF BC Applications

There are various options for handling validation in ADF BC applications:

- **Business services:** You should build in as much validation as you can when creating the business services in order to protect the integrity of the data in the database and to prevent malicious attacks. There are a multitude of opportunities to add validation at the business services level:
 - Oracle ADF Business Components provides a number of predefined validation rule classes that enable you to add business logic without writing a single line of code, or you can also add programmatic validation.
- **Database (PL/SQL):** You can implement validation in the database; however, such validation cannot be exposed in the UI. For example, you cannot use database validation to expose a list of valid values in the UI.
- **User interface:** The ADF Faces input components that you use to build the pages of your application also have built-in validation capabilities. You use these prebuilt ADF Faces validators to ensure that when a user edits or enters data in a field and submits the form, the data is validated against the rules and conditions you have specified. However, UI validation should always have equivalent validation at the business services layer, so that the same validation is applied when the model is exposed in other ways.

Validation Options for ADF BC Applications (continued)

Managing the Various Validation Options

There are various opportunities for adding validation in ADF BC applications (as shown in the slide), but be aware that they can create an overhead in terms of managing them; keeping them in synch with each other is a manual process.

Take the example of a data structure, Person, which represents the concept of a Person entity known to the business. This Person must have a gender and that gender's value can be either Male or Female. You define a simple business rule with a name, optionality, and domain. This rule can easily end up being implemented in different places (as described below).

You create a relational table in the database and create a column called gender and make it a mandatory column. The table's DDL is one implementation of the part of the rule. If you add a SQL check constraint (gender = male or gender = female), you have completely implemented the rule in the table and have a single location to maintain.

However, in most object-oriented (OO) systems today, you need a software object to represent this rule through Object-Relational Mapping (ORM). So now you need a class named Person and an attribute named gender. You could code this in Java and use JEE/JPA or ADF BC to provide the support for ORM. This is not the problem. The problem is that you want to implement that rule closer to the client. You can thus implement the rule in the class itself and have the class test the gender value before it gets back to the database. So, the rule is now implemented in two languages in two places.

But if this is a Web application, you might want to have the rule enforced in the client, before getting back to the Java class in the server. So, now you implement the rule in JavaScript. The rule is now implemented in three languages and platforms.

And it does not stop there. If you wanted to expose the Person functionality via a Web server, you might code the rule there also, or in different languages for different applications. It would not be unusual, over time to see this same gender rule implemented in DDL, SQL, PL/SQL, Java, JavaScript, an EJB, a Web Service , C++, .Net, BPEL, a rules engine for SOA systems, and so on.

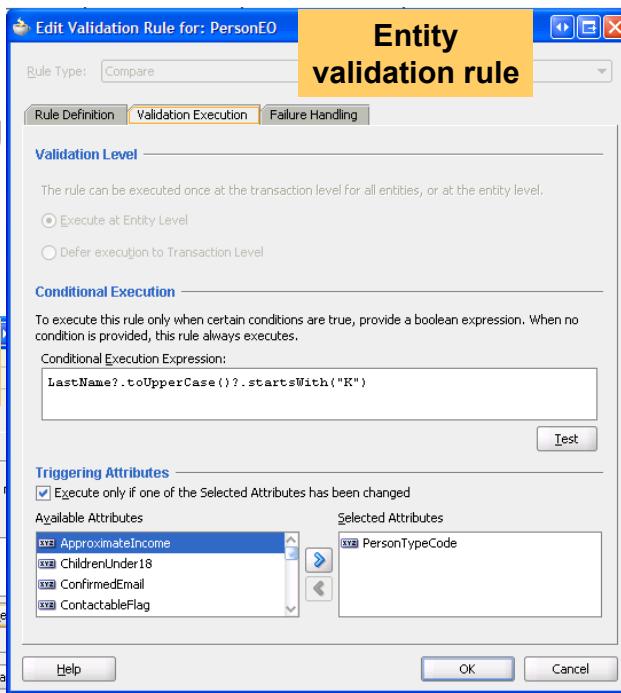
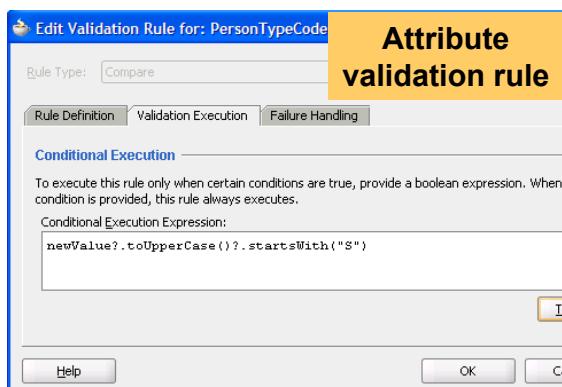
Eventually the rule may need to be changed, or extended. How do you know all the places to change it? How do you know if you have found and changed them all? And, how do you know it was changed or implemented correctly and consistently in all the different places?

Thus, you see that adding validation in the necessary places to protect your data is only part of the task; managing the maintenance of it is equally important.

Triggering Validation Execution

The Validation Execution tab enables you to specify:

- When validation should occur
- The conditions governing validation execution
- Which attributes trigger validation



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Triggering Validation Execution

Each entity row tracks whether or not its data is valid. When a user modifies a persistent attribute of an existing entity row or creates a new entity row, the entity is marked as invalid, and any validation that you have implemented is evaluated again before the entity is again considered valid. In validation rules, you can specify when this validation is executed as follows:

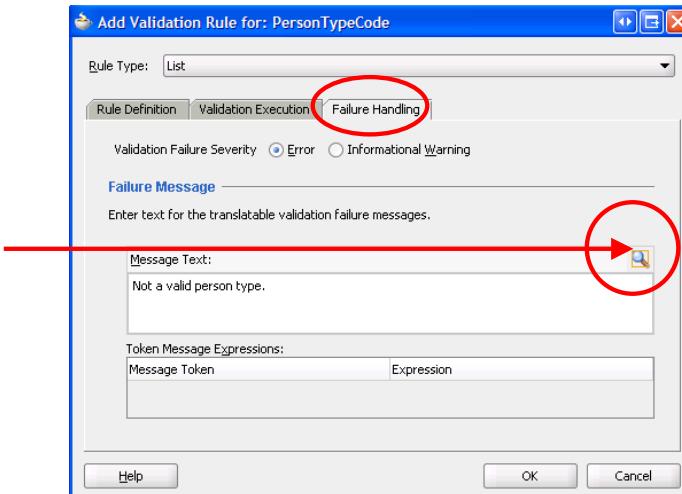
- **Validation Level:** In an entity validation rule, you can specify whether validation should be performed at the entity-level (the default) or at the transaction level. If you specify that it should be performed at the transaction level, it will be carried out after all entity-level validation. For this reason, you should use this option where you want to ensure that a specific piece of validation is performed at the end of the process.
- **Conditional Execution:** You can enter a Boolean condition using a Groovy expression to define conditional validation execution. When the condition evaluates to `true`, the rule is executed. You have the option to test your expression to see if it is valid.
- **Triggering Attributes:** By default, a validator fires on an attribute whenever the entity as a whole is dirty. However, in an entity validation rule, you can choose to trigger validation only when certain attributes change (when one of the triggering attributes is dirty).

Validation always occurs before changes are posted to the database. If you must code validation that depends on seeing posted changes, override the `beforeCommit()` method in the object's Java file.

Handling Validation Errors

Use the Failure Handling tab on the Validation Rule page of the entity object editor or attribute editor to:

- Specify message severity
- Insert Groovy variables into error messages
- Click Select Text Resource to store error messages as translatable strings



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Handling Validation Errors

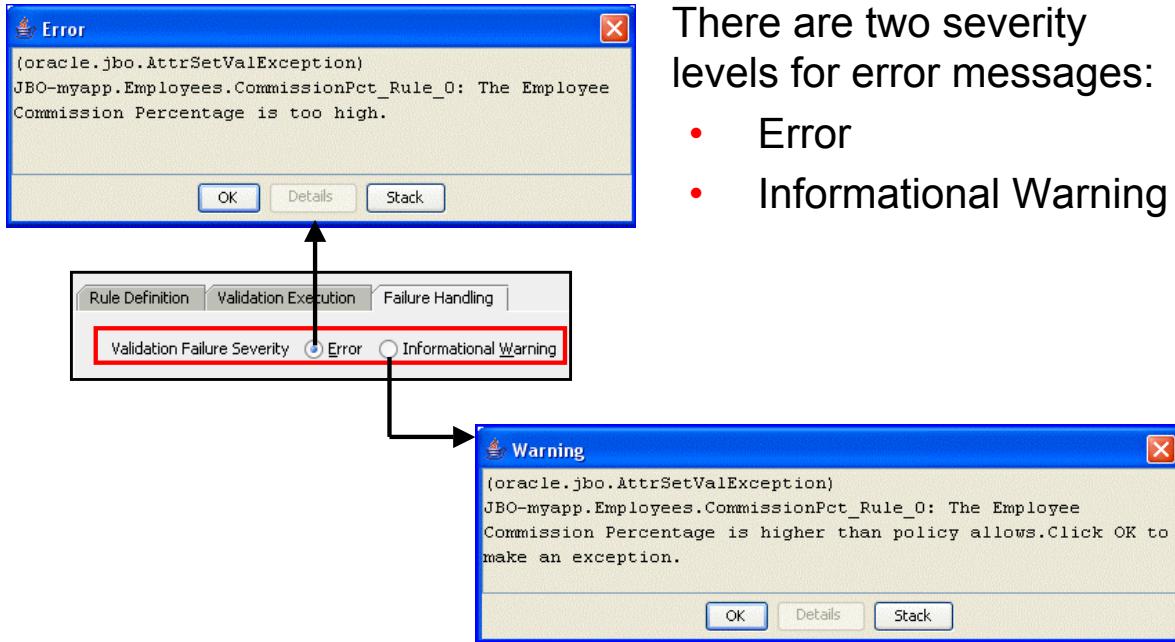
When a validation rule throws exceptions, the exceptions are bundled and returned to the client, and any database changes pertaining to the transaction are rolled back.

The Failure Handling tab on the Validation page of the Validation Rule page of the entity object editor (or the attribute editor) is where you specify the error message that you want to display to the end user when the validation fails.

You use the Validation Failure Severity option buttons to specify whether the message is an Error message (default) or Informational Warning.

You can enter Groovy expressions to populate variables in error message text to make the message more meaningful. And you can store messages in a resource bundle or property file to make them translatable.

Specifying the Severity of an Error Message



There are two severity levels for error messages:

- Error
- Informational Warning

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

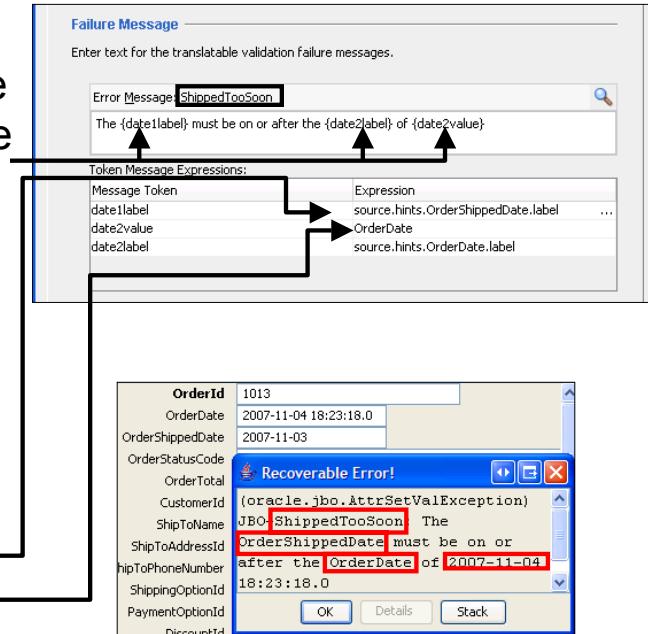
Specifying the Severity of an Error Message

If you define the message as an Error message, the end user is not able to navigate out of the current field or record and must revisit the field where the data has failed the validation test. If the message is classified as Informational Warning, the user can continue after the message has been acknowledged.

Using Groovy Variables in Error Messages

In error messages:

- You indicate a variable by surrounding a name with braces {}
- Message tokens are added automatically
- You provide message token expressions:
 - source.hints.<attr>.label for labels
 - <attr> for values



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Using Groovy Variables in Error Messages

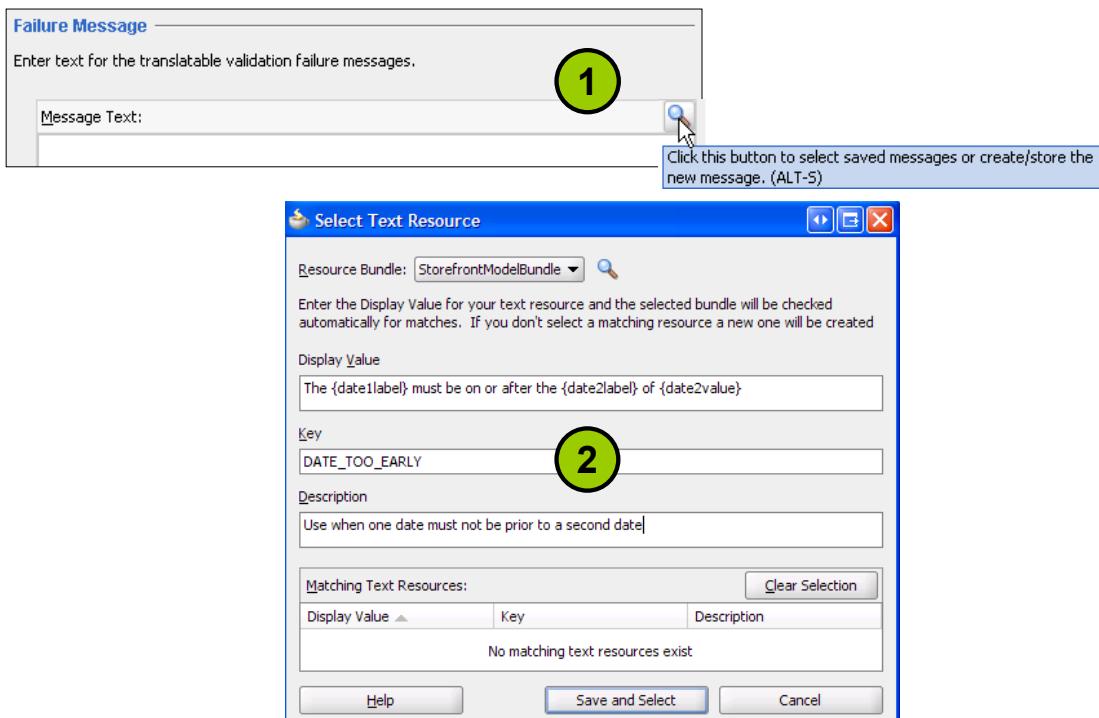
Using Groovy variables in error messages enables you to display dynamic labels and values in your messages. This also makes your messages more reusable when adding to a resource, as described in the next slide.

When you create the message, use any variable name, surrounded by braces. JDeveloper automatically adds the variable as a message token in the lower part of the Failure Handling page.

You must supply the values for the message tokens. You can use expressions such as source.hints.OrderDate.label to reference the value of the user-friendly display label of the OrderDate attribute of the source entity object being evaluated. To reference the value, simply use the name of the attribute.

At run time, the error message is displayed with the specified labels and values.

Storing Error Messages as Translatable Strings



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Storing Messages as Translatable Strings

Typically you store error messages in a resource bundle or other property file to facilitate translation and reuse of messages. A resource bundle contains key-value pairs for any messages and data strings in your application that you want to translate or reuse.

To use an existing message or create a new stored message, perform the following steps:

1. Click the Select Text Resource button to open the Select Text Resource dialog box.
2. In the Select Text Resource dialog box:
 - To use an existing message, select it and click Select
 - To create a new message, enter the message in the Value field. This automatically populates the Key field, but you can change the key to a more meaningful name if desired. You also enter a description, which can assist the translator in knowing how the message is to be used.
 - Click "Save and Select" to store and use the message

Defining Validation in the Business Services

- Implementing validation in entity objects ensures that validation is:
 - Consistent
 - Easily maintained
- You can define validation by using:
 - Declarative validation:
 - Built-in validation rules
 - Custom validation rules
 - Programmatic validation
 - Domains: Prevalidated custom data types



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Defining Validation in the Business Services

When you incorporate validation into the ADF BC business model, the most logical place to implement such validation is in the entity objects. Encapsulating business logic in these shared reusable components provides the following benefits:

- Ensures that your business information is validated consistently on every page where end users are allowed to make changes
- Simplifies maintenance by centralizing validation

There are several possibilities for implementing validation, as described in the following slides.

Using Declarative Validation: Built-in Rules

- Greatly reduce the need for coding
- Include the following types of rules:
 - Defined only at the entity level:
 - Collection validator
 - Unique key validator
 - Can be defined at the entity or attribute level, but pertain to an EO attribute:
 - Compare validator
 - Key Exists validator
 - Length validator
 - List validator
 - Range validator
 - Regular Expression validator
 - Can be defined at the entity or attribute level to validate the EO or the attribute:
 - Method validator
 - Script Expression validator



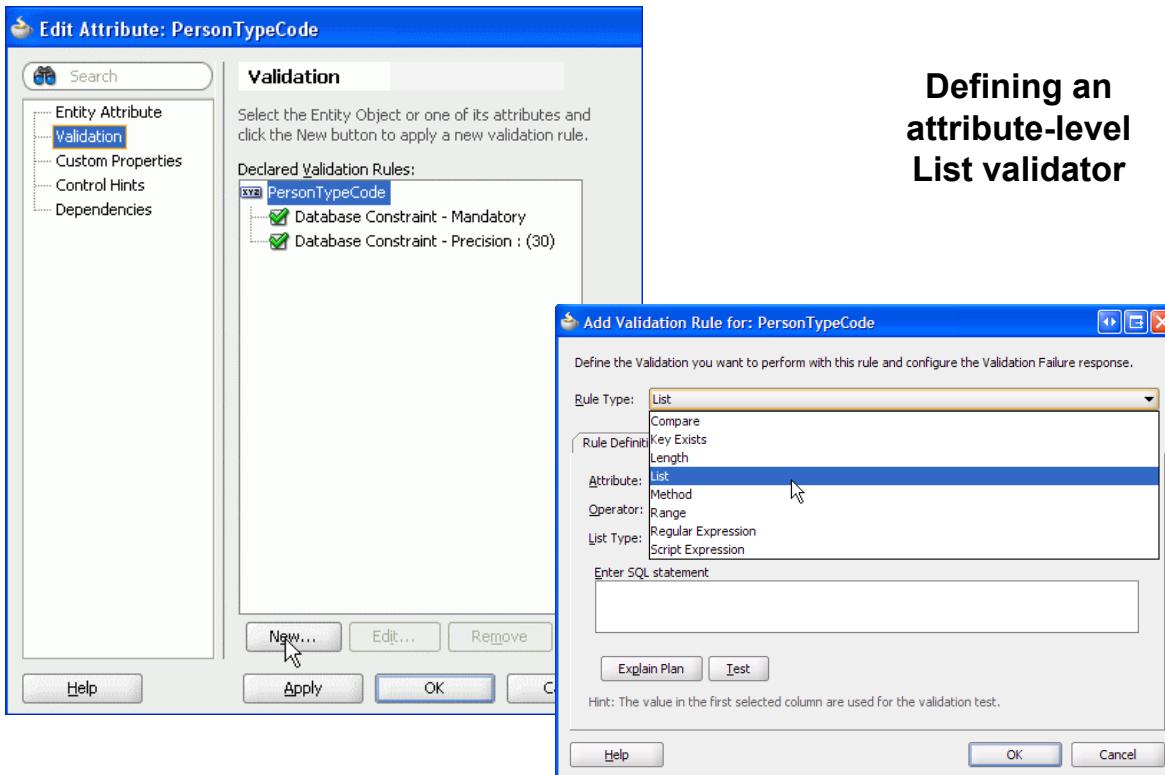
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Prebuilt Declarative Validation Rules

The easiest way to create and manage validation rules is through the declarative validation framework. Oracle ADF is shipped with a number of built-in declarative validation rules that will satisfy many of your business needs. You implement these rules through the entity object editor and they are stored in the entity object's XML file.

You can implement these rules either on the entity object itself, or on its attributes. The next few slides show how to use the built-in declarative validation rules.

Defining Declarative Validation



Defining an attribute-level List validator

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Defining Declarative Validation

To include declarative validation on an entity object, perform the following steps:

1. Double-click the entity object in the Application Navigator to invoke the editor.
2. Invoke the Validation Rule editor:
 - For entity validation, in the Entity Validation Rule section of the General tab, click Add Validation Rule.
 - For attribute-level validation, select the Attributes page and double-click the attribute for which you would like to define a validation rule. In the Edit Attribute dialog box, click Validation in the list on the left of the page. In the Validation pane (with the attribute selected), click New. You can also define the attribute.
3. In the Add Validation Rule dialog box, select the type of validation required from the Rule Type list and add details of the rule.
4. Define validation execution rules and failure handling procedures by clicking the relevant tabs and entering the preferences.
5. Click OK. The validation rule is displayed in the list of validation rules for the entity or attribute that owns it.

Using Declarative Validation: Built-in Rules

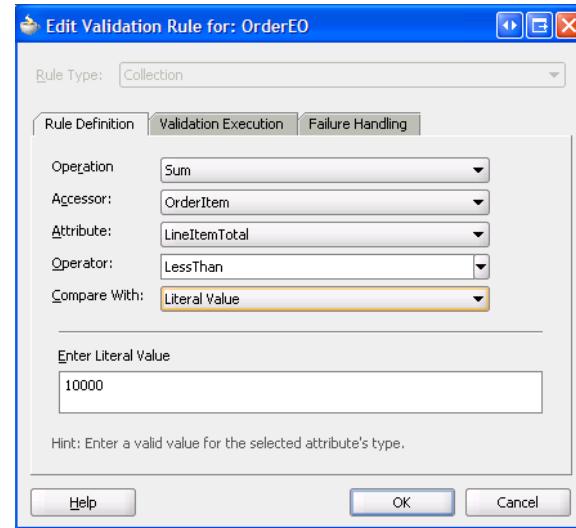
- Built-in rules greatly reduce the need for coding
- Include the following types of rules:
 - Defined only at the entity level:
 - Collection validator
 - Unique Key validator
 - Can be defined at the entity or attribute level, but pertain to an EO attribute:
 - Compare validator
 - Key Exists validator
 - Length validator
 - List validator
 - Range validator
 - Regular Expression validator
 - Can be defined at the entity or attribute level to validate the EO or the attribute:
 - Method validator
 - Script Expression validator

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Declarative Built-in Rules: Collection Validator

- Collection validators are defined at the entity level.
- To define a Collection validator, specify:
 - Operation
 - Accessor
 - Attribute
 - Operator
 - Compare With
 - Compare with value (if using a literal)



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Collection Validator

You can use collection validation to:

1. Compute the average, count, sum, min, or max of an attribute in a child of the current entity object
2. Compare the computed value with a literal, a view object attribute, an entity attribute, or the value returned by a SQL query, a view accessor, or an expression

For example, you could verify that the total value of an order (the sum of its line items) does not exceed \$10,000, as depicted in the example in the slide.

When you use a Collection validator, a `<CollectionValidationBean>` tag is added to the XML file.

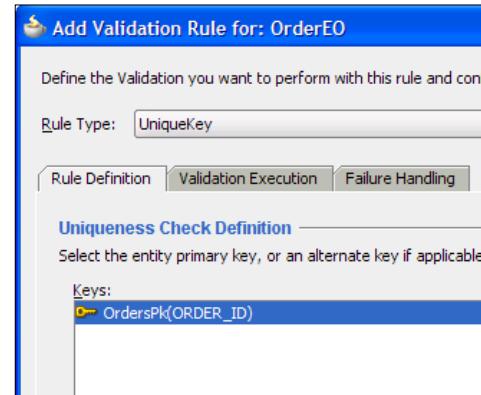
You set the following options to define a Collection validator:

- **Operation:** Select sum, average, count, min, or max.
- **Accessor:** Select an association that has been defined for the entity.
- **Attribute:** Select the child attribute on which the operation is to be performed.
- **Operator:** Select equals, not equals, less than, greater than, less than or equal to, or greater than or equal to.
- **Compare With:** Select literal value, query result, view object attribute, view accessor attribute, expression, or entity attribute.
- **Enter Literal Value:** Select or define the value with which to compare the calculation.

Using Declarative Built-in Rules: Unique Key Validator

The Unique Key validator:

- Is defined at the entity level
- Ensures that primary key values are always unique
- Can also be used for alternate keys
- Upon failure, throws `TooManyObjectsException`



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Unique Key Validator

A Unique Key validator can be defined only at the entity level. However, it behaves internally like an attribute-level validator; users see the validation error when they attempt to tab out of the key attribute that the validator is validating.

When you create a Unique Key validator, a `<UniqueKeyValidationBean>` tag is added to the entity object's XML file.

Using Declarative Validation: Built-in Rules

- Built-in rules greatly reduce the need for coding
- Include the following types of rules:
 - Defined only at the entity level:
 - Collection validator
 - Unique Key validator
 - Can be defined at the entity or attribute level, but pertain to an EO attribute:
 - Compare validator
 - Key Exists validator
 - Length validator
 - List validator
 - Range validator
 - Regular Expression validator
 - Can be defined at the entity or attribute level to validate the EO or the attribute:
 - Method validator
 - Script Expression validator



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Declarative Built-in Rules: Compare Validator

For a Compare validator, you specify:

1. Attribute (if entity level)

2. Operator

3. Value to compare with

The screenshot shows the 'Add Validation Rule for: PersonEO' dialog. The 'Rule Type' is 'Compare'. The 'Attribute' is 'PersonId', 'Operator' is 'Equals', and 'Compare With' is 'Literal Value'. A callout points to each step: 1 points to the 'Attribute' dropdown, 2 points to the 'Operator' dropdown, and 3 points to the 'Compare With' dropdown.

Attribute-level Compare validator

The screenshot shows the 'Add Validation Rule for: OrderShippedDate' dialog. The 'Rule Type' is 'Compare'. The 'Attribute' is 'OrderShippedDate', 'Operator' is 'GreaterOrEqualTo', and 'Compare With' is 'Expression' with 'OrderDate' selected. A callout points to the 'Compare With' dropdown.

Entity-level Compare validator

The screenshot shows the 'Edit Validation Rule for: PersonEO' dialog. The 'Rule Type' is 'Compare'. The 'Attribute' is 'CreditLimit', 'Operator' is 'LessThan', and 'Compare With' is 'Literal Value' with '10000' entered. A callout points to the 'Enter Literal Value' field.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Compare Validator

The Compare validator performs a logical comparison between an entity attribute and another value. You specify the operator and the value to compare. The Compare With list includes:

Compare With choice	Attribute is compared with
Literal Value	A literal value that must conform to the format of the attribute's data type
Query Result	The value of the first column of the first row of the SQL query
VO Attribute	The value of an attribute in the first row of the selected VO
View Accessor Attribute	The value of an attribute in the first row of the view accessor's row set
Expression	The value of the defined expression
Entity Attribute	The value of another attribute in the EO of the same data type; valid only for entity-level compare validators

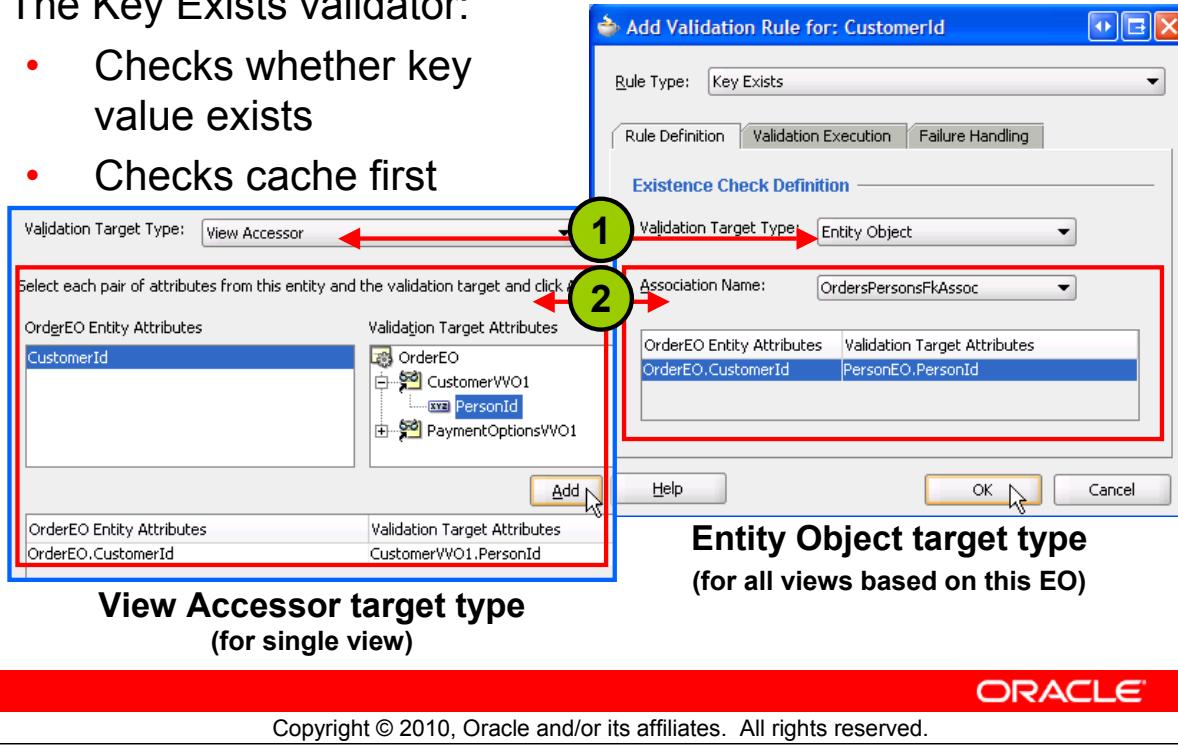
Using a Compare Validator (continued)

Depending on your Compare With choice, additional options must be specified, such as the SQL query or the view accessor and attribute. When you define a Compare validator, a `<CompareValidationBean>` tag is added to the XML file.

Using Declarative Built-in Rules: Key Exists Validator

The Key Exists validator:

- Checks whether key value exists
- Checks cache first



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Key Exists Validator

The Key Exists validator checks whether a key exists based on a primary, an alternate, or a foreign key. It first checks the cache, thus finding rows not yet committed to the database, and then checks the database if the key is not found in the cache.

When you define a Key Exists validator, after you have selected the rule type in the Add Validation Rule dialog box, you also specify the following:

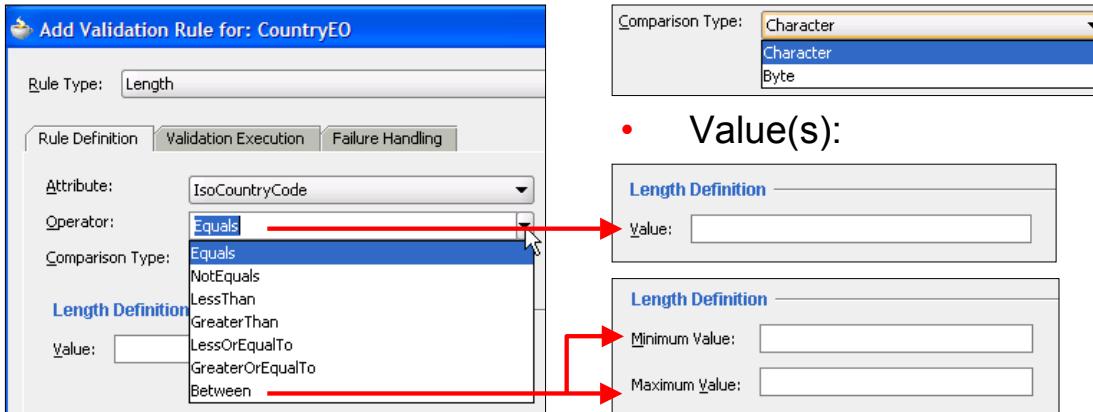
1. Validation Target Type: You can select Entity Object, View Object, or View Accessor. If you want the Key Exists validator to be used for all view objects that use this entity attribute, select Entity Object.
2. If you have chosen the Entity Object target type, you select the Association Name from a drop-down list. If you have chosen either View Object or View Accessor, you select an entity attribute and the attribute of the target whose key you are searching, and then click Add to add the attribute/target pair.

When you use a Key Exists validator, an <ExistsValidationBean> tag is added to the XML file.

Using Declarative Built-in Rules: Length Validator

For a Length validator, you specify:

- Attribute (if defining at the EO level)
- Operator:
- Comparison Type:



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Using a Length Validator

The Length validator compares the number of characters or bytes in the attribute value against the specified length. You select the operator and comparison type, and then you set the length value.

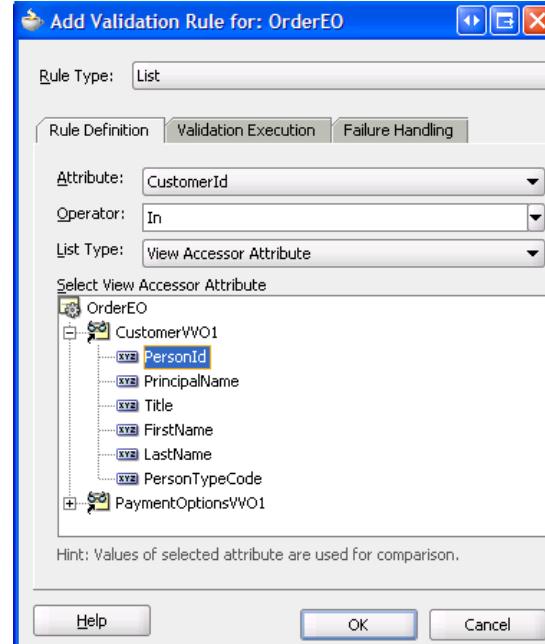
When you use a Length validator, a <LengthValidationBean> tag is added to the XML file.

Using Declarative Built-in Rules: List Validator

Ensures that value is in (or not in) a list defined by:

- Literal values
- Query result
- VO attribute
- View accessor

Although you can define at the entity or attribute level, the List validator pertains to an entity attribute.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a List Validator

The List validator compares an attribute against a list that is of one of the following types:

- **Literal value:** Ensures that the value is in, or not in, the list of literal values that you define
- **Query result:** Ensures that the value is in, or not in, the first column of the query's result set
- **View attribute:** Ensures that the value is in, or not in, the attribute of the specified view object; all rows are retrieved
- **View accessor:** Ensures that the value is in, or not in, the specified attribute in all rows of the view object retrieved by the view accessor. This is similar to a view attribute, except that a view accessor is required to be defined when you want to use an LOV in the user interface.

The example in the slide shows a List validator that ensures that a valid customer ID is entered for an order. It is based on a view accessor, defined on the OrderEO, that accesses a view object that queries customers.

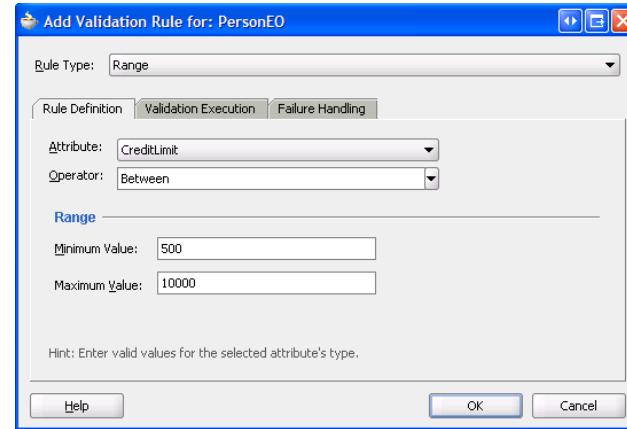
When you use a List validator, a <ListValidationBean> tag is added to the XML file.

Note: List validators based on SQL queries or VO attributes retrieve all rows of a query each time validation is performed. It is recommended that you use these types of List validators only for a relatively small set of values.

Using Declarative Built-in Rules: Range Validator

For a Range validator, you specify:

- Attribute: Although you can define at the entity or attribute level, the Range validator pertains to an attribute.
- Operator: Between or Not Between
- Range: Minimum and maximum values



ORACLE

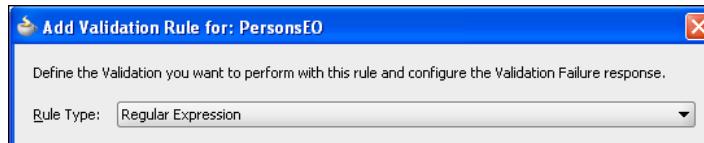
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Range Validator

You can use a Range validator to ensure that an attribute falls within, or outside of, specified minimum and maximum values. This validator performs an inclusive range comparison.

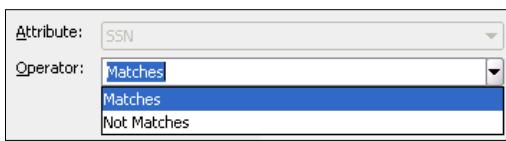
When you use a Range validator, a <RangeValidationBean> tag is added to the XML file.

Using Declarative Built-in Rules: Regular Expression Validator

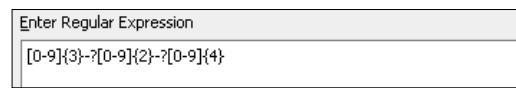


For a Regular Expression validator, you specify:

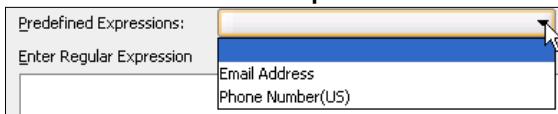
- Operator:



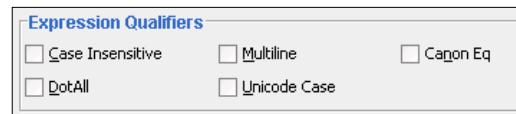
- Regular Expression:



- Predefined Expressions



- Qualifier(s):



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Regular Expression Validator

Regular expressions provide a powerful syntax that enables matching strings with particular characters, words, or patterns of characters. This syntax can be quite complex and is outside the scope of this course. To learn about regular expressions, you can visit <http://www.regular-expressions.info>, or explore the Java tutorial at <http://java.sun.com/docs/books/tutorial/essential/regex/index.html>.

The Regular Expression validator compares attribute values against a mask specified by a Java regular expression. This type of validator enables you to provide a dynamic script for more complex validation rules.

Instead of defining an expression in this validator, you can select from a list of predefined expressions. The built-in predefined expressions include telephone number and email address. You can add your own expressions to this list by modifying the `PredefinedRegExp.properties` file. This file is located in the `o.BC4J` subdirectory of the JDeveloper system directory (`system11.1.1.0.xx.xx.xx`), which is a subdirectory of JDeveloper's user directory.

When you use a Regular Expression validator, a `<RegExpValidationBean>` tag is added to the XML file.

Using Declarative Validation: Built-in Rules

- Built-in rules greatly reduce the need for coding
- Include the following types of rules:
 - Defined only at the entity level:
 - Collection validator
 - Unique Key validator
 - Can be defined at the entity or attribute level, but pertain to an EO attribute:
 - Compare validator
 - Key Exists validator
 - Length validator
 - List validator
 - Range validator
 - Regular expression validator
 - Can be defined at the entity or attribute level to validate the EO or the attribute:
 - Script Expression validator
 - Method validator

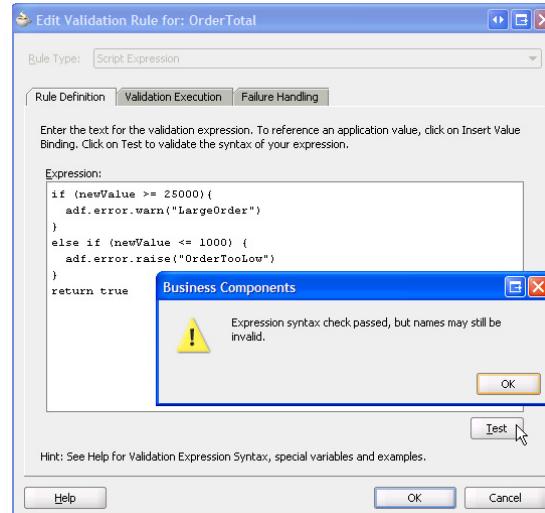


Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Declarative Built-in Rules: Script Expression Validator

To define a Groovy expression, you:

- Omit braces { }
- Use newValue for new value of current attribute
- Write code to return true or false and/or call adf.error.raise or adf.error.warn
- Can use aggregate functions on RowSet
- Can use ternary operator
- Click Test to check syntax



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Script Expression Validator

A Script Expression validator enables you to use a Groovy expression that returns either true or false for validation, or to use an entire Groovy script. When you define a Script Expression validator, an <ExpressionValidationBean> tag is added to the XML file.

The following are some usage notes about the Script Expression validator:

- All Java methods, language constructs, and Groovy constructs are available in the script.
- Do not use { } to surround the entire script.
- Use newValue to refer to the new value of the attribute being validated.
- Use the return keyword just like in Java to return true or false, unless it is a one-line expression, in which case the return is assumed to be the result of the expression itself—for example, `Sal > 0`.
- You must either return true or false, or call `adf.error.raise` or `adf.error.warn` to display a message that is defined in your message bundle.
- You can use built-in aggregate functions on RowSet objects by referencing the functions: `sum(String attrName)`—for example: `sum("Salary")`, `count(String attrName)`, `avg(String attrName)`
- Use the ternary operator to implement functionality that is similar to SQL's `NVL()` function—for example, `Sal + (Comm != null ? Comm : 0)`.

Using Declarative Custom Rules: Entity-Specific Rules (Method Validators)

Method validators:

- Extend declarative rules for entities or attributes
- Call Java methods in your entity object (not in a separate class as a global rule is)
- Are called automatically during validation cycle
- Must:
 - Be defined as public
 - Return a Boolean value
 - Be named like validateXXX()

```
public boolean validateOrder(){  
    if( // add your validation code )  
        return true;  
    else  
        return false;  
}
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Entity-Specific Validation Rules

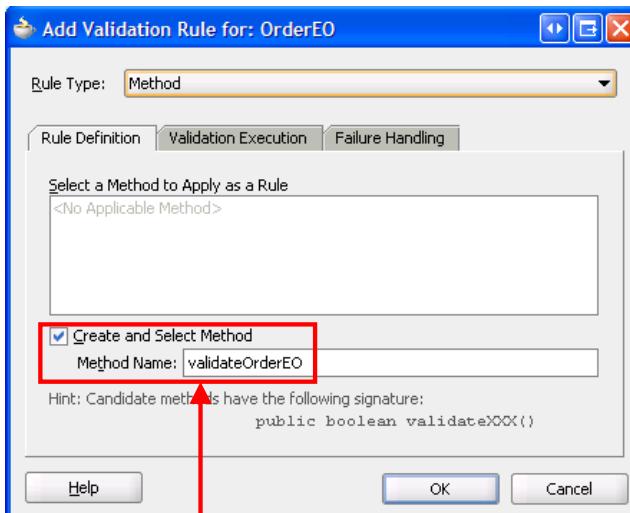
You can implement more complex validation rules for your business domain layer by using your own Java code in Method validators. You code the rule in Java, and it is triggered by the Method validator at the appropriate point in the entity object validation cycle.

You can create entity-level or attribute-level Method validators. You can add any number of attribute-level or entity-level Method validators, provided they each trigger a distinct method name in your code. There are many types of validation you can code with a Method validator, either on an attribute or on an entity.

The method must be defined as public and return a Boolean value. The name of the method must begin with the keyword validate. The code can be as complex or simple as the validation rules require.

Using Declarative Custom Rules: Creating an Entity-Specific Method Validator

Create at either the entity or attribute level:



Adds a method to the
<EO>Impl.java file

```

public RowSet getPaymentOptionsVVO1
    return (RowSet)getAttributeInt(
)
}

/**Validation method for OrderEO.
 */
public boolean validateOrderEO() {
    return true;
}

/**Creates a Key object based on g1
 */

```

Add your own code
to the validateXXX() method.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating a Method Validator

To create a Method validator for an entity, perform the following steps:

1. Either for the entity object or for one of its attributes, invoke the Add Validation Rule editor as you do for any type of validation rule.
2. Select Method from the Rule Type drop-down list.
3. Select an existing method, or to create a new one, leave the “Create and Select Method” check box selected and enter a name for your method in the Method Name field. Remember that your method name must begin with the word validate, or use the default name.
4. Click OK to add the method to your entity object’s Java class. You are then given the option to create the <EO>Impl.java file if it does not exist.
5. In the <EO>Impl.java file, edit the method to add your own validation code.

When you add a new Method validator, JDeveloper updates the XML component definition to reflect the new validation rule, adding a <MethodValidationBean> tag to the XML file.

Using Declarative Global Validation Rules

Custom global validation rules are available to assign declaratively to any business components by using the following steps:

1. Create a Java class that implements the `JboValidatorInterface` interface in the `oracle.jbo.rules` package.
2. Modify the code for the validation rule.
3. Assign the rule to a business component object.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Declarative Global Validation Rules

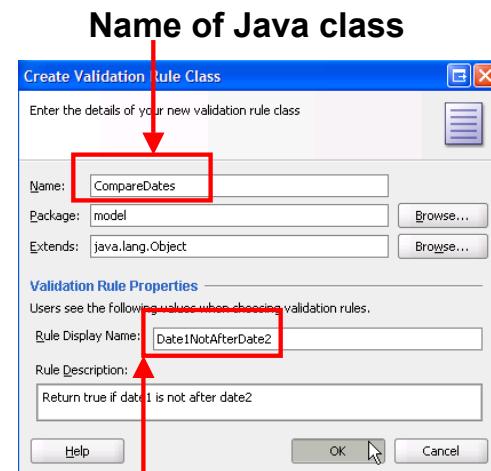
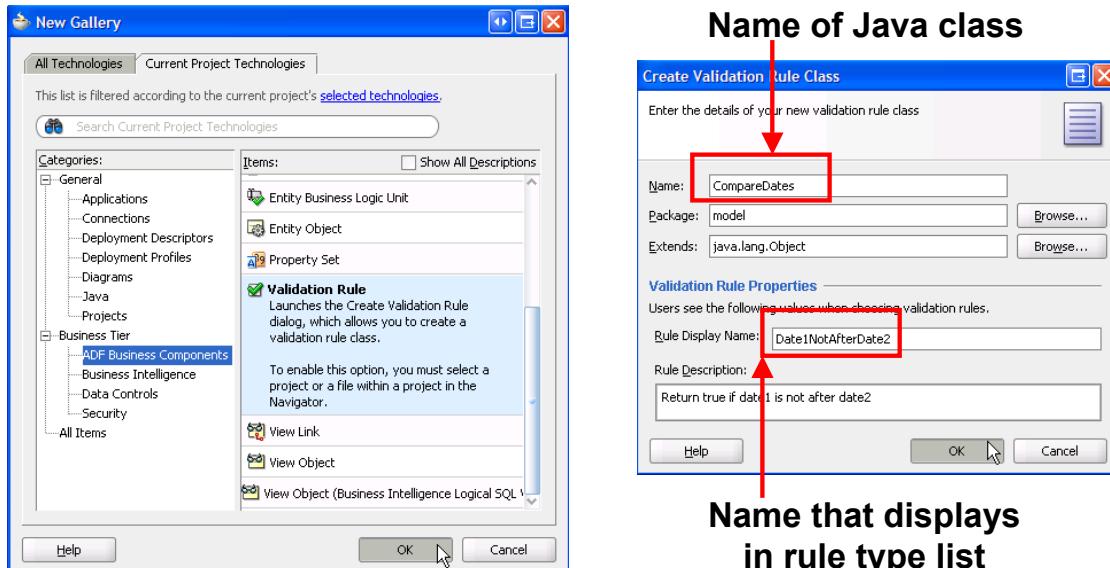
As you saw in the preceding slides, ADF Business Components comes with a base set of built-in declarative validation rules. However, one of the most powerful features of the validator architecture is that you can create your own programmatic validation rules. When you realize that you are writing the same kind of validation code over and over again, you can build a validation rule class that captures this validation “pattern.” After you have defined a validation rule class, it is as simple to use as any of the built-in rules.

For easier reuse of your custom validation rules, you can package them into a Java Archive (JAR) file for reference by applications that make use of the rules.

The slides that follow describe each of the steps involved in the process of creating and using your own declarative validation rules.

Creating the Java Class for a Global Rule

Purpose of rule: Compare any two dates in any entity object and raise an exception if the first date is after the second date.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Creating the Java Class for a Global Rule

The Java class needs to implement the `JboValidatorInterface` interface in the `oracle.jbo.rules` package. To create the class, invoke the New Gallery and in the Categories list, select Business Tier > ADF Business Components. Select Validation Rule in the Items list to create a class where you can add the validation rule code.

In the Create Validation Rule Class dialog box, enter a name and package for the rule, along with a display name and description. The display name should be the name you want to appear in the list of validators to choose from when adding validation to an attribute, whereas the description becomes a comment in the code to describe the validation that is implemented. When you click OK, the Java class is created.

The example in the slide shows creating a validation rule that compares two dates, validating that one date is not later than another. The intention is that this rule can be used to compare any two date attributes in any entity object.

Examining the Generated Skeleton Code

```
public class CompareDates implements JboValidatorInterface {  
    private String description = "Returns true if date1 is not after date2";  
    public CompareDates() {}  
    public boolean validateValue(Object value) {  
        return true;  
    }  
    public void validate(JboValidatorContext ctx) {  
        if (!validateValue(ctx.getNewValue())) {  
            throw new ValidationException("model.CompareDates validation failed");  
        }  
    }  
    public String getDescription() {  
        return description;  
    }  
    public void setDescription(String str) {  
        description = str;  
    }  
}
```

This method is executed when validation occurs.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Examining the Generated Skeleton Code

The slide shows the skeleton code for the example (without the package name, import statements, or comments). The Java class that is created implements the JboValidatorInterface interface with its three methods:

- `get Description()`: Gets the text description of this validator
- `setDescription(java.lang.String description)`: Sets the text description of this validator
- `validate(JboValidatorContext ctx)`: Tests the validity of the object or value as described in the validator context; this is the method that is executed when the validation occurs

The newly created class also sets the description that you entered when you defined the rule, and adds its own `validateValue()` method where you can add the validation code. You can replace the return statement with your own code.

After you have created a rule in this way, it is automatically registered in the project and is available for use in your business components.

Modifying the Code for the Global Rule

```

1   public class CompareDates implements JboValidatorInterface {
2       private String description = "Return true if date1 is not after date2";
3       private String earlierDateAttrName = "";
4       private String laterDateAttrName = "";
5
6       public void validate(JboValidatorContext ctx) {
7           if (validatorAttachedAtEntityLevel(ctx)) {
8               EntityImpl eo = (EntityImpl)ctx.getSource();
9               Date earlierDate = (Date)eo.getAttribute(getEarlierDateAttrName());
10              Date laterDate = (Date)eo.getAttribute(getLaterDateAttrName());
11              if (!validateValue(earlierDate, laterDate)) {
12                  throw new ValidationException("model.CompareDates validation failed");
13              }
14          } else {
15              throw new RuntimeException("Rule must be at entity level");
16          }
17      }
18
19      private boolean validateValue(Date earlierDate, Date laterDate) {
20          return (earlierDate == null) || (laterDate == null) ||
21                 (earlierDate.compareTo(laterDate) <= 0);
22      }
23
24      private boolean validatorAttachedAtEntityLevel(JboValidatorContext ctx) {
25          return ctx.getOldValue() instanceof EntityImpl;
26      }

```

This method is executed when validation occurs.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Modifying the Code for the Global Rule

To achieve the goal of comparing any two date attributes in an entity, you modify the skeleton code as follows:

1. You want to use this validation rule to compare any two dates in any entity object, so you first add some properties that will specify which entity attributes should be compared. Add two more private strings and initialize them to null:

```
private String earlierDateAttrName = "";
private String laterDateAttrName = "";
```

You also generate public accessors for these new properties by right-clicking the editor and selecting Generate Accessors. Accessors are not shown in the example in the slide.

2. Because you intend this to be an entity-level validation rule, you add a method that returns true if the rule is applied at the entity level:

```
private boolean
validatorAttachedAtEntityLevel (JboValidatorContext ctx) {
    return ctx.getOldValue() instanceof EntityImpl;
}
```

Modifying the Code for the Global Rule (continued)

3. Next you perform the comparison in the `validate()` method. You first call the method that you just defined to ensure that the validator is defined at the entity level, and then you call `validateValue()` to compare the dates, raising an exception if the validation fails:

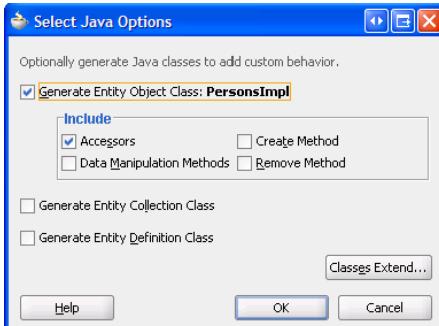
```
public void validate(JboValidatorContext ctx) {  
    if (validatorAttachedAtEntityLevel(ctx)) {  
        EntityImpl eo = (EntityImpl)ctx.getSource();  
        Date earlierDate =  
            (Date)eo.getAttribute(getEarlierDateAttrName());  
        Date laterDate =  
            (Date)eo.getAttribute(getLaterDateAttrName());  
        if (!validateValue(earlierDate, laterDate)) {  
            throw new ValidationException("model.CompareDates  
validation failed");  
        }  
    }  
    else {  
        throw new RuntimeException("Rule must be at entity  
level");  
    }  
}
```

4. Finally, you modify `validateValue()` to perform the comparison:

```
private boolean validateValue(Date earlierDate, Date  
laterDate) {  
    return (earlierDate == null) || (laterDate == null) ||  
        (earlierDate.compareTo(laterDate) <= 0);  
}
```

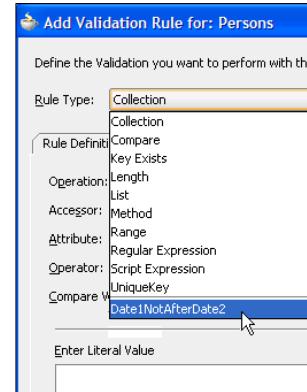
Assigning the Global Rule to an Object

Generate the entity Java class.



Create a new validator for an attribute or for the entity.

Select the global rule from the Rule Type list.



Provide values for any properties.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Assigning the Global Rule to an Object

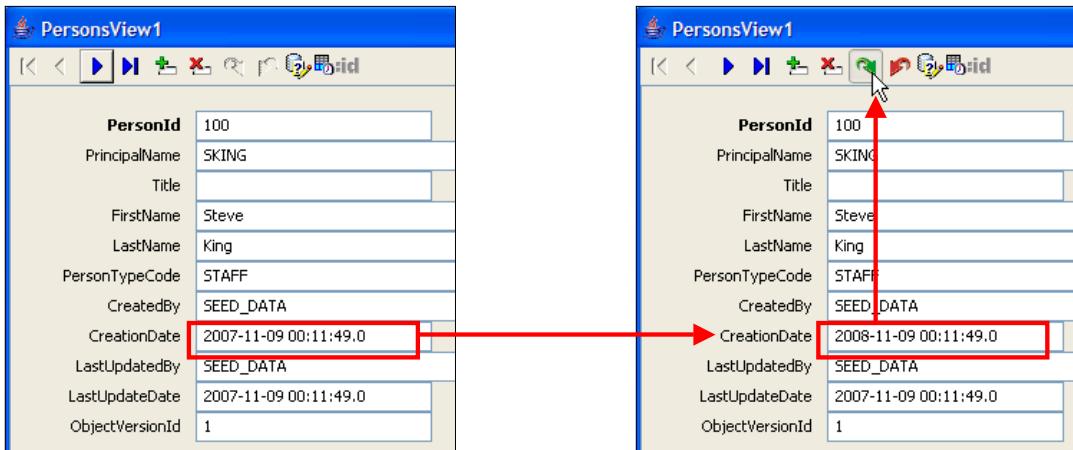
After the business rule has been registered with the project (which is automatic when you create a new validation rule from the New Gallery), you can assign it to an entity or attribute. This is completed in the properties for any specific entity object. Select the entity or attribute to use the business rule, and invoke the Add Validation Rule dialog box. The display name that you specified appears in the Rule Type list of the Add Validation Rule dialog box.

When using the rule, you are required to specify values for any properties defined in the rule. In this way, the programmatic validation rule can be reused in another entity object with different date values.

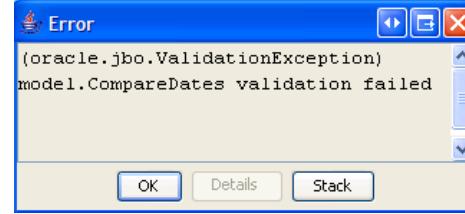
The example shows assigning the rule to the Persons entity. The Rule Type comes from the Rule Display Name that you specified when you created the new validation rule. After selecting the rule type as the global validation rule that you defined, you are presented with a rule definition panel where you can provide values for the properties that you defined in the validation rule. In this case, you want `CreationDate` of the Person entity to be before, or the same, as `LastUpdateDate`, so you specify those attribute names for the `earlierDateAttrName` and `laterDateAttrName`, respectively.

You also can modify the Validation Execution tab so that validation fires only when one of the specified attributes has changed. In this example, you need to also generate the Java class for the entity, because the code accesses `EntityImpl` to get the names of the attributes to compare.

Testing the Global Validation Rule



When you change the date to violate the validation rule and attempt to commit the change, an error message displays the run-time exception.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Testing the Global Validation Rule

At run time, an instance of the validator class is instantiated for each usage in each entity object. Its metadata property values are set during metadata load time, and might have different values for each instance of the validator that is used in entity objects.

The validators may be used by multiple threads at the same time. They must be thread-safe and they should not maintain private state; that is, they should not store data between calls to the validate() method.

To test the rule, invoke the Business Components Browser for the application module containing a view based on the entity where the validation is defined. Change a value so that the validation rule is violated, then tab out of the field for attribute-level validation or commit the record for entity-level validation. This should generate the exception that you defined in the validation rule's Java class.

Using Programmatic Validation

Create custom methods in the EntityImpl.java file:

```
public boolean checkOrderMode(){
    if ( ("ONLINE".equals(getOrderMode()) &&
          (getCustomerEmail() == null))
    {
        return false;
    }
    else {
        return true;
    }
}
```

Call the overridden validateEntity() method:

```
protected void validateEntity(){
    if (!checkOrderMode()) {
        throw new JboException("Online order must have email");
    }
    super.validateEntity();
}
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Programmatic Validation

You can also implement validation logic in a nondeclarative manner adding a method to an entity object and then explicitly calling it as part of the validation cycle. You can accomplish this by adding a method in the EntityImpl.java file. Because the method is not called automatically by the framework, call it from the validateEntity() method. The validateEntity() method is called at commit time when any attribute has been updated.

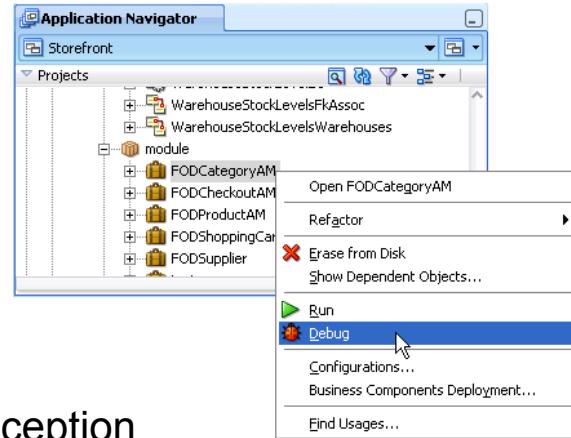
To create the validateEntity() method if it does not exist, perform the following steps:

1. Select Source > Override Methods from the source code editor of the <EO_Name>.Impl.java class.
2. Select validateEntity() to override.
3. Add code to call the custom method:

```
protected void validateEntity() {
    if (!checkOrderMode()) {
        throw new JboException("Online order must have email");
    }
    super.validateEntity();
}
```

Debugging Custom Validation Code with the JDeveloper Debugger

- JDeveloper Debugger is useful for pinpointing problems in your custom validation code.
- Set source breakpoints to pinpoint problems.
- Set exception breakpoints to stop when a particular exception is thrown.
- At breakpoints, you can execute code one line at a time and view variable values.
- To run a file in debug mode, right-click and select Debug.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Debugging Custom Validation Code with the JDeveloper Debugger

JDeveloper's integrated debugger is very useful for finding and fixing problems in your code. It offers both local and remote debugging of Java code. A local debugging session is started by setting breakpoints in source files, and then starting the debugger. When debugging a program in JDeveloper, you have complete control over the execution of the flow, and can view and modify values of variables.

More information about the debugger is included in the lesson titled "Troubleshooting ADF BC Applications" in this course.

Using a Domain to Create Custom-Validated Data Types

- Domains are Java classes that extend the basic data types (String, Number, Date, and so on).
- You can create your own data types that encapsulate specific validation rules.
- Use domains for more complex validation:
 - Format of a phone number
 - Validity of a URL
 - Validity of an email address
 - Checksum of digits of a credit card number
- Validation is done by the domain constructor.
- A domain is not bound to a particular entity or attribute.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using a Domain to Create Custom-Validated Data Types

When you find yourself repeating the same sanity-checking validations on the values of similar attributes across multiple entity objects, you can save yourself time and effort by creating your own data types that encapsulate this validation.

For example, in your application, you may have numerous entity object attributes that store strings that represent URLs. One technique you could use to ensure that end users always enter a valid URL is to create your own data type (for example, `URLDomain`) that represents the structure and format of a URL. You can then use `URLDomain` as the data type of every attribute in your application that represents a URL.

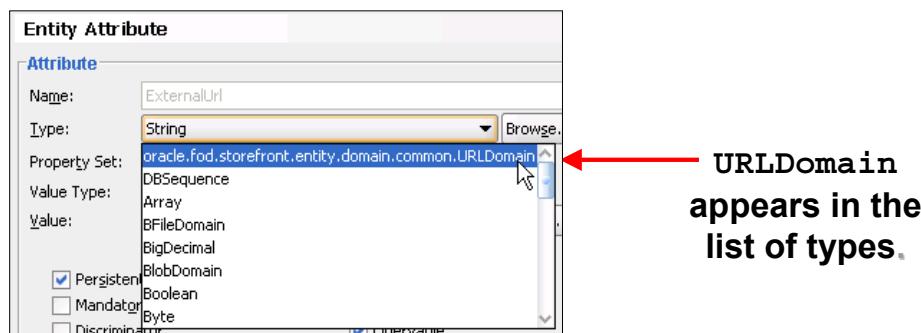
When you create a new domain object, JDeveloper generates code in the domain's constructor to call the `validate()` method of the domain class. When you add your validation code to this method, it is automatically executed when the domain object is instantiated.

The clear benefit of using a domain is that a domain can be used by multiple attributes, whereas a validation rule (other than a global rule) applies to one attribute or entity object only.

Note: Domain validation code is executed whenever you query existing data. If existing data fails the check, you need to either correct the existing data or code the check so that existing data is determined to be valid.

Creating and Using a Domain

1. Create the domain.
2. Add validation code to the validate() method in the domain's Java file.
3. Edit an entity and change the type of an attribute to the domain.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating and Using a Domain

The slide shows the steps to create and use a domain.

1. You create the domain by right-clicking in the Application Navigator and selecting New Domain, or by invoking the New Gallery (Business Tier > ADF Business Components > Domain).
2. When you create a new domain object, JDeveloper generates code in the domain's constructor to call the validate() method, as shown in the next slide. Therefore, if you add your validation code to the validate() method, it is automatically executed when the domain object is instantiated.
3. In the final step, you change the type of an entity's attribute to the domain, such as URLDomain, as shown in the example in the slide. This causes the validation code to be called when the entity object is instantiated.

Coding Validation in a Domain

- The validate() method is called by the domain's constructor: (mData is a private String variable containing the string to be validated.)
 - Example: URLDomain verifies that an attribute is a valid URL.

```
protected void validate()
{
    try {
        // Make sure the user entered a valid URL
        java.net.URL u = new java.net.URL(mData);
    }
    catch (java.net.MalformedURLException e)
    {throw new
        oracle.jbo.domain.DomainValidationException
        ("Invalid URL"); }
}
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Domains: Example

When you create a domain object, you are essentially creating a new data type; the example creates the type called URLDomain. You can then change an entity attribute's type to your new type; in this example, you can change the type of an entity's attribute to URLDomain. When the entity is instantiated, or when the entity's attribute is populated or changed, the URLDomain() constructor is called. The constructor executes the validation code that you have written to verify that this attribute is valid.

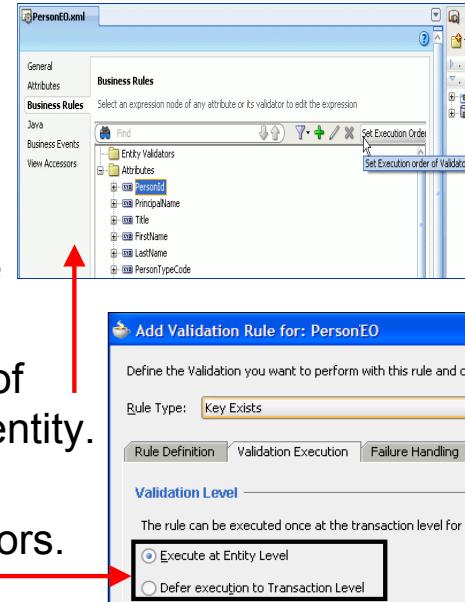
Because String is a base JDK type, a domain based on a String variable aggregates a private mData String member field to hold the value that the domain represents.

Note: Domain validation code is executed whenever you query existing data. If existing data fails the check, you need to either correct the existing data or code the check so that existing data is determined to be valid.

Specifying Validation Order

General validation order:

- Changed attributes are validated when they lose focus.
- Changed entities are validated at commit time.
- In compositions, child entities are validated before parent entities.
- You can control execution order of validation within attributes or an entity.
- You can specify validation at the transaction level for some validators.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Specifying Validation Order

Attribute validation occurs when focus moves from the attribute, whereas entity validation occurs when the record is committed. With compositions, changing a child entity causes the parent entity to be validated as well, with child entities being validated before parent entities.

You can specify the validation order of rules within a certain group (entity or an attribute) by using the up and down arrows to rearrange the validation rules on the Validators page of the editor. For example, although you cannot control the order in which attributes are validated (they are validated in the order in which they appear in the entity definition), you can order validations for a given attribute.

If you want to ensure that a Key Exists or an entity-level Method validator is performed at the end of the validation process, you can set it to perform at the transaction level, rather than at the entity level. It is then fired after all entity-level validation is performed. You set the level of validation on the Validation execution tab of a validation rule editor.

Summary

In this lesson, you should have learned how to:

- Describe the types of validation available to ADF BC applications
- Decide which validation options are appropriate for different validation requirements
- Use the declarative validation options provided in JDeveloper
- Create programmatic validation
- Use domains in validation



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Summary

The ADF BC developer has opportunities for adding validation into the application at the business services level as well as in the UI. This lesson focused on adding validation at the business services level.

The easiest way to create and manage validation rules is through the declarative validation framework. Oracle ADF is shipped with a number of built-in declarative validation rules that will satisfy many of your business needs. In addition to the prebuilt validation rules, you can also create your own XML validator beans and add them to the declarative framework.

You can implement more complex validation rules for your business domain layer by using your own Java code in Method validators. There are many types of validation you can code with a Method validator, either on an attribute or on an entity. You also can add programmatic validation that you explicitly call, and you can define domains to use as prevalidated data types for multiple attributes.

JDeveloper's integrated debugger is a useful tool for pinpointing problems in your custom validation code.

Practice 8 Overview: Implementing Validation

This practice covers the following topics:

- Adding Declarative Validation: List Validator
- Adding Declarative Validation: Unique Key Validator
- Adding Programmatic Validation: Method Validator
- Creating and Using a Domain for Validation
- Testing the Validation



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 8 Overview: Implementing Validation

In this practice, you add several types of both declarative and programmatic validation to entity objects, including a domain, and you test the validation.

Troubleshooting ADF BC Applications

9

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Tools for logging and diagnostics
- Design-time code validation
- FileMon and JUnit
- JDeveloper Profiler
- JDeveloper Debugger
- Sources of help



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

As an application is being built, it is advisable to test it whenever a functionality is added, so that errors are caught as early as possible. This makes problems easier to diagnose and correct. This lesson discusses ways to track down application problems and create test cases to improve software quality. It discusses tools to help you troubleshoot Fusion applications, including FileMon, JUnit, ADF Logger, ADF Diagnostics, and the JDeveloper Debugger. You also learn how to obtain assistance with JDeveloper and application problems.

Troubleshooting the Business Service

- Test a business service in isolation from views or controllers:
 - BC Browser
 - Java test clients
- Use JDeveloper and ADF tools for logging and diagnostics:
 - Java logging
 - ADF Logger
 - ADF Diagnostics
- Validate code at design time.
- Build unit tests with JUnit.
- Diagnose CLASSPATH and other file access problems using FileMon.
- Diagnose performance problems using JDeveloper Profiler.
- Step through code using JDeveloper Debugger.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Troubleshooting the Business Service

You have already learned about two ways to test business services without the need to define a user interface:

- Business Components Browser (see the lessons titled “Building a Business Model with ADF Business Components” and “Querying and Persisting Data”). If you want to troubleshoot problems with Java code in your model, you can set breakpoints in the code, and then run the BC Browser in debug mode as follows: In the Structure window, right-click the application module’s `<AppModuleName>Impl.java` file (for example, `EmpModuleImpl.java`), and select Debug from the context menu.
- Java test client (see the lesson titled “Programmatically Customizing Data Services”)

This lesson discusses various other methods that you can use to diagnose problems with the business service model.

Troubleshooting the UI

- Java EE Logging (Java Logger, ADF Logger)
- Design-time troubleshooting
- Using the JDeveloper Debugger
- Debugging through the source



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Troubleshooting the UI

Some of the methods that you learn in this lesson are also useful for troubleshooting problems with the user interface.

Using Logging and Diagnostics

- Displaying debug messages to the console
- Java logging
- ADF Logging/Oracle Diagnostic Logging (ODL)



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

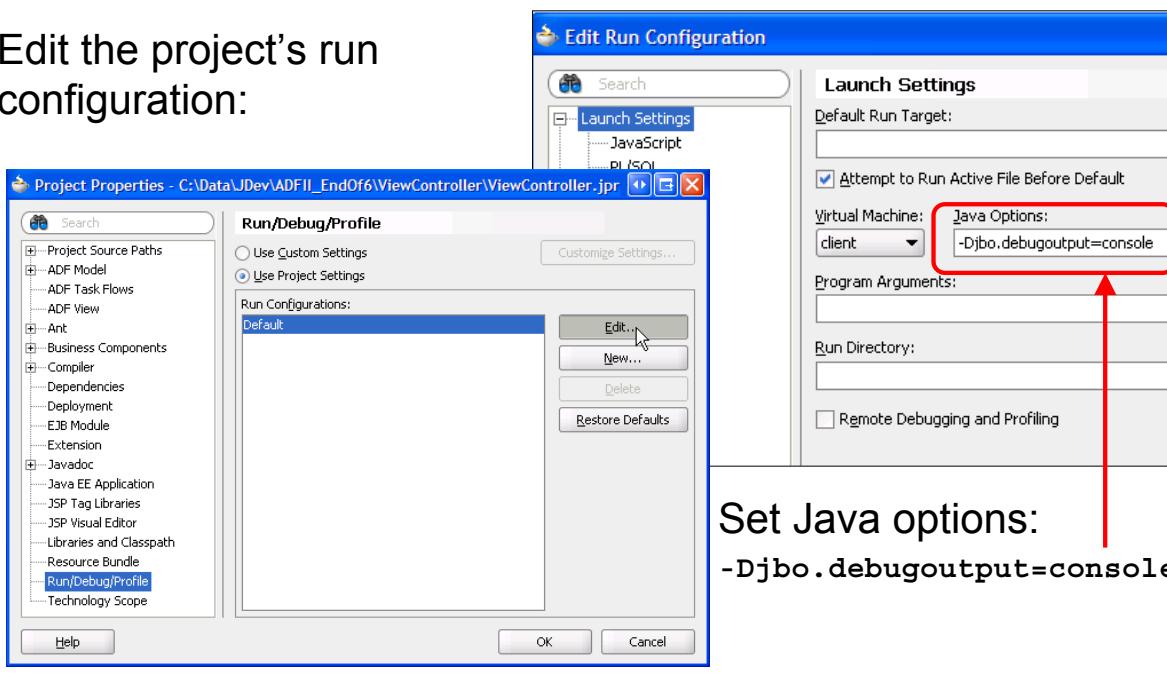
Logging and Diagnostics

Java and Oracle provide logging frameworks:

- The Java logging framework, introduced in JDK 1.4, provides extensive logging APIs through the `java.util.logging` package. You can find an overview of the `java.util.logging` package at <http://java.sun.com/j2se/1.4.2/docs/api/overview-summary.html>. For an overview of the Java logging framework, visit <http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html>.
- Oracle's logging framework complements the standard Java framework to automatically integrate log data with Oracle log analysis tools. In the ODL framework, log files are formatted in XML, enabling them to be more easily parsed and reused. The ODL framework provides support for managing log files, including log file rotation. The maximum log file size and the maximum size of log directories can also be defined.

Displaying Debug Messages to the Console

Edit the project's run configuration:



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Displaying Debug Messages to the Console

Even before starting to use the actual debugger, running with the framework's diagnostic logging turned on can be helpful to see what is happening when a problem occurs. To turn on diagnostic logging, set the Java system property named `jbo.debugoutput` to the value `console`. Legal values for this flag are `silent`, `console`, `file`, and `ADF Logger`. (`ADF Logger` is discussed shortly.)

To set this system property while running your application inside JDeveloper, edit project properties and select the Run/Debug/Profile node in the tree at the left. You can either edit an existing run configuration, such as Default, or create a new one. Then add the following string to the Java Options field: `-Djbo.debugoutput=console`.

Java Logging

- It is useful in environments where you cannot run the debugger or where running the debugger might hide the problem.
- Many Java programs use the Java logging API to produce useful messages.
- You can configure Java logging to produce different levels of logging output:
 - SEVERE
 - WARNING
 - INFO
 - CONFIG
 - FINE
 - FINER
 - FINEST
 - ALL
 - NONE



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Java Logging

The logging API is part of the Java Platform, Standard Edition (Java SE) in JDK 1.4 and later versions, and the `java.util.logging` package is shipped with the JDK. It is designed to enable a Java program to produce messages of interest to end users, system administrators, and software developers. It is useful in production environments where you cannot run the debugger, or where running the debugger might hide the problem. For example, timing-related problems often cannot be reproduced when running in the debugger.

The primary class in the logging facility is `Logger`. A `Logger` represents a sort of channel through which logging messages can be sent. Usually each class has its own `Logger`.

A `Logger` is configured with a `Level`. This is a class that indicates the severity of the problem being reported or the level of detail required for that individual class, or both. The predefined levels are as follows:

- SEVERE: Extremely important messages such as fatal errors
- WARNING: Warning messages
- INFO: Informational run-time messages
- CONFIG: Informational messages about configuration settings
- FINE: Used for greater detail when diagnosing problems
- FINER: Even greater detail

Java Logging (continued)

- FINEST: Greatest detail

There are two other values: ALL (log all messages) and NONE (no messages).

A LogRecord is an object that represents a message that should be written to a log. It contains a variety of information, including the message to be printed, the name of the Logger to which it was originally sent, the Level and creation time and date of the message being sent, and the thread ID of the caller.

In ADF, Java logging is particularly useful for tracing Expression Language (EL) evaluation in Faces, although the JDeveloper Debugger has been enhanced to enable you to evaluate EL expressions.

Core Java Logging

- Edit <Java_Home>/jre/lib/logging.properties:
 - Set
java.util.logging.ConsoleHandler.level=FINE.
 - Add the line: com.sun.faces.level=FINE.
- Now the log contains JSF Reference Implementation debug messages.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Core Java Logging

You can use Java logging to display to the console the JSF phases of the life cycle and other exceptions occurring in the JSF code. You configure this in the <Java_Home>/jre/lib/logging.properties file. You set the logging level by changing the following line:

java.util.logging.ConsoleHandler.level = INFO

Change the level from INFO to FINE.

Then, at the end of the file, add the following line:

com.sun.faces.level=FINE

The log now includes JSF Reference Implementation messages, such as:

FINEST: End execute (phaseId=RESTORE_VIEW 1)

com.sun.faces.lifecycle.LifecycleImpl

haspostDataOrqueryParams

FINEST: Request Method: POST/PUT

com.sun.faces.lifecycle.LifecycleImpl execute

Using ADF Logging

- ADF Logging is built on Oracle Diagnostic Logging (ODL), which:
 - Enables the capture of context information
 - Provides control over which messages are logged by setting the:
 - Logging level
 - Module filter
- ODL uses a common format and repository for all components in the WebLogic Server.

The red bar spans most of the width of the slide, centered horizontally.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using ADF Logging

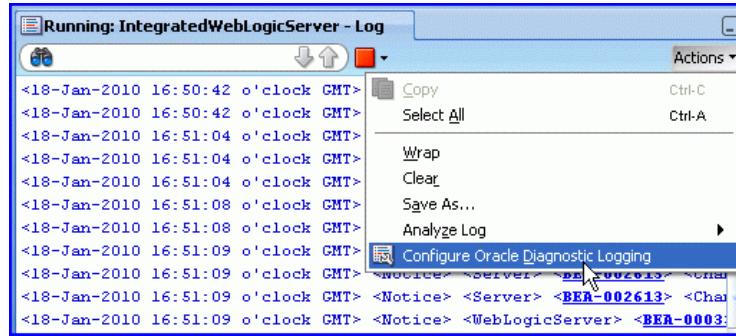
Oracle Diagnostic Logging (ODL) enables the automatic capture of context information for meaningful analysis. You have control over which messages are logged, through logging level and module filter.

By using ODL, ADF Logging messages are handled in the same way as all logging messages in the application server—they are output in a common format, and loaded into a common repository where they can be viewed as a single stream or analyzed in different ways.

ADF Faces leverages the Java Logging API (`java.util.logging.Logger`) to provide logging functionality.

Configuring ADF Logging

- The Oracle Diagnostic Logging (ODL) configuration file (`logging.xml`) controls the loggers under the Oracle tree.
- You use the overview editor for Oracle Diagnostic Logging Configuration to configure the logging levels specified in the `logging.xml` file.
- You access the overview editor for Oracle Diagnostic Logging Configuration from the Application Server Navigator or from the Log window.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Configuring ADF Logging

Oracle WebLogic Server (WLS) uses `logging.xml` to configure the ODL loggers globally. You can access the overview editor for Oracle Diagnostic Logging Configuration from the Application Server Navigator or from the Log window. The slide shows the Actions menu in the Log window.

There are several ADF loggers that you can configure in the ODL configuration file.

A logging level is a threshold set by the system administrator to control the logging of messages. The logging can be set to any one of the following seven severities for each logger individually:

- Failure reporting:
 - SEVERE: Highest level of severity to catch any unexpected errors during normal execution
 - WARNING: For internal errors or exceptions for which the user was not notified
- Progress reporting:
 - INFO: Key flow steps
 - CONFIG: Configuration properties and environment settings
 - FINE: High-level logging message (This is the level that Oracle recommends.)
 - FINER: Entry or exit from a routine
 - FINEST: Low-level logging messages giving maximum detail

Configuring ADF Logging (continued)

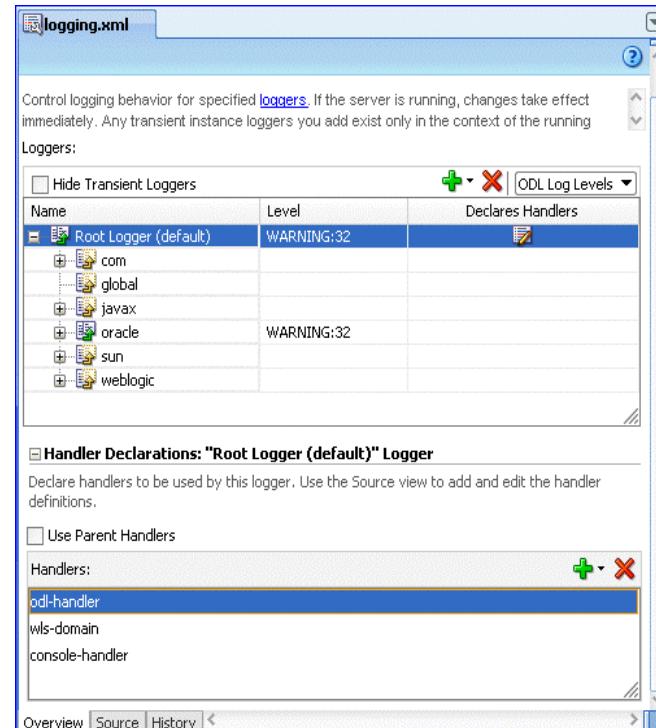
After a level is set, only messages that have a severity greater than or equal to the defined level will be logged. For example, setting the level to WARNING means that logging occurs for WARNING and SEVERE messages. Setting the level to the lowest severity, FINEST, means that messages of all seven severities are logged.

A module filter is an optional comma-delimited list of non-case-sensitive strings. You can define a filter, so only messages that match the filter criteria get logged. Use of a wildcard (%) is supported—for example, MODULE=store%, jtf%.

ODL Configuration Overview Editor

You can:

- Choose to view ODL log levels or Java log levels for the logger types
- Add persistent or transient loggers
- Specify a handler for a logger



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ODL Configuration Overview Editor

You configure logging levels or view existing log settings in the ODL Configuration Overview Editor.

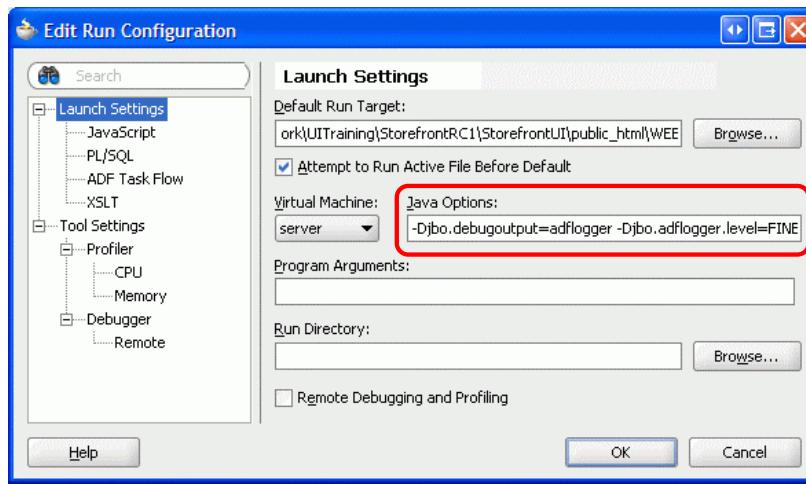
To configure log levels, perform the following steps:

1. In the overview editor, click the Overview tab, and then select ODL Log Levels or Java Log Levels for the logger types that you want to view.
2. If you want to see only persistent loggers, select Hide Transient Loggers.
3. To add a logger:
 - a. If the server is running, click the Add icon drop-down menu, and select Add Persistent Logger or Add Transient Logger. If the server is not running, click Add to add a persistent logger. You cannot add a transient logger.
 - b. In the Add Logger dialog box, enter a logger name.
 - c. Select the logging level.
 - d. Click OK.
4. For any logger, including a newly created logger, you can specify its handlers by selecting from a list of available handlers by clicking the Add icon in the Handler Declarations section.

Creating Logging Configurations in JDeveloper

ADF Model debugging configuration:

- Edit the project's run configuration.
- Set Java options: **-Djbo.debugoutput=adflogger
-Djbo.adflogger.level=FINE**



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating Logging Configurations in JDeveloper

To create an Oracle ADF Model debugging configuration, perform the following steps:

1. In the Application Navigator, double-click the user interface project.
2. In the Project Properties dialog box, click **Run/Debug/Profile** and create a new run configuration, for example, named ADF Debugging.
3. Double-click the new run configuration to edit the properties.
4. In the Edit Run Configuration dialog box, for Launch Settings, enter the following Java options for the default virtual machine:
-Djbo.debugoutput=adflogger -Djbo.adflogger.level=FINE
Oracle recommends **level=FINE** for detailed diagnostic messages.
5. On the **Run** menu, select the new configuration as the active run configuration (**Run > Choose Active Run Configuration**).

Viewing ODL Logs

- You use the Oracle Diagnostic Log Analyzer to view the log entries of a log file.
- The log analyzer enables you to filter the entries by log level, entry type, log time, and entry content.

Time	Message Id	Type	Message	Module	Application	Related
Jun 9, 2009 ...		Notification	ADFApplicati...	oracle.adf.s...	wsm-pm	
Jun 9, 2009 ...		Notification	ADFApplicati...	oracle.adf.s...	wsm-pm	
Jun 9, 2009 ...		Notification	Cleaning up ...	oracle.adf.s...	wsm-pm	
Jun 9, 2009 ...		Notification	Calling ADF ...	oracle.adf.s...	wsm-pm	
Jun 9, 2009 ...		Notification	Releasing M...	oracle.adf.s...	wsm-pm	
Jun 9, 2009 ...		Notification	MBean: oracl...	oracle.mds	wsm-pm	
Jun 9, 2009 ...		Notification	Remove appl...	oracle.jps.d...	StoreFrontM...	
Jun 9, 2009 ...		Notification	ADFApplicati...	oracle.adf.s...	StoreFrontM...	

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Viewing Logs with the Log Analyzer

1. From the main menu, select **Tools > Oracle Diagnostic Log Analyzer**.
2. In the editor for Oracle Diagnostic Log Analyzer, navigate to the log file or enter the path and name of the log file.
3. From the drop-down list, select **ODL Log Level** or **Java Log Level**.
4. Select the corresponding check box for each type of log entry you want to view. You must select at least one type.
The available **ODL** log level types are Incident Error, Error, Warning, Notification, Trace, and Unknown.
The available **Java** log level types are Severe, Warning, Info, Config, Fine, Finer, Finest, and Unknown.
5. Specify a time period for the entries you want to view. You can select the most recent period or a range.
6. To filter the results, use the query panel to search on a text pattern. For additional query panels, click **Add**.
7. To initiate the filters and display the log messages, click **Search**.
8. To order the results by the message ID, select the **Group by Id** check box.
9. To group the messages by time period or by request, in the Related column, select either **Related by Time** or **Related by Request**.
10. To show or hide columns for display, select from the **Column** drop-down list.

Using Design-Time Code Validation

JDeveloper's editors provide error cues and correction suggestions for files such as:

- Java
- XML
- JSPX



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Design-Time Code Validation

In addition to providing features such as code insight, JDeveloper's source editor provides code validation, error cues, and correction suggestions. The following slides discuss how this works with Java, XML, and JSPX files.

Auditing Java Code

The screenshot shows the Oracle JDeveloper IDE interface. A Java file named `FDDShoppingCartAMClient.java` is open in the editor. The code is annotated with various inspection markers: a red exclamation mark icon above the `addItemToCart` method indicates a warning or error. A yellow question mark icon is placed over the `InvokeExportedMethod` call. A tooltip box appears over the yellow question mark, stating "Method 'InvokeExportedMethod' not found". The code itself is as follows:

```
import oracle.jbo.client.remote.ApplicationModuleImpl;
import oracle.jbo.domain.Number;
//
// --- File generated by Oracle ADF Business Components Design Time
// --- Mon Sep 08 17:16:42 MDT 2008
// --- Custom code may be added to this class.
// --- Warning: Do not modify method signatures of generated methods.
//
public class FDDShoppingCartAMClient extends ApplicationModuleImpl implements ...
{
    /**
     * This is the default constructor (do not remove).
     */
    public FDDShoppingCartAMClient() {
    }

    public void init() {
        Object _ret = this.riInvokeExportedMethod(this,"init",null,null);
        return;
    }

    public void addItemToCart(Number productId) {
        Object _ret =
            this.InvokeExportedMethod(this,"addItemToCart",new String[]
        {
            productId.toString()
        });
        return;
    }
}
```

ORACLE

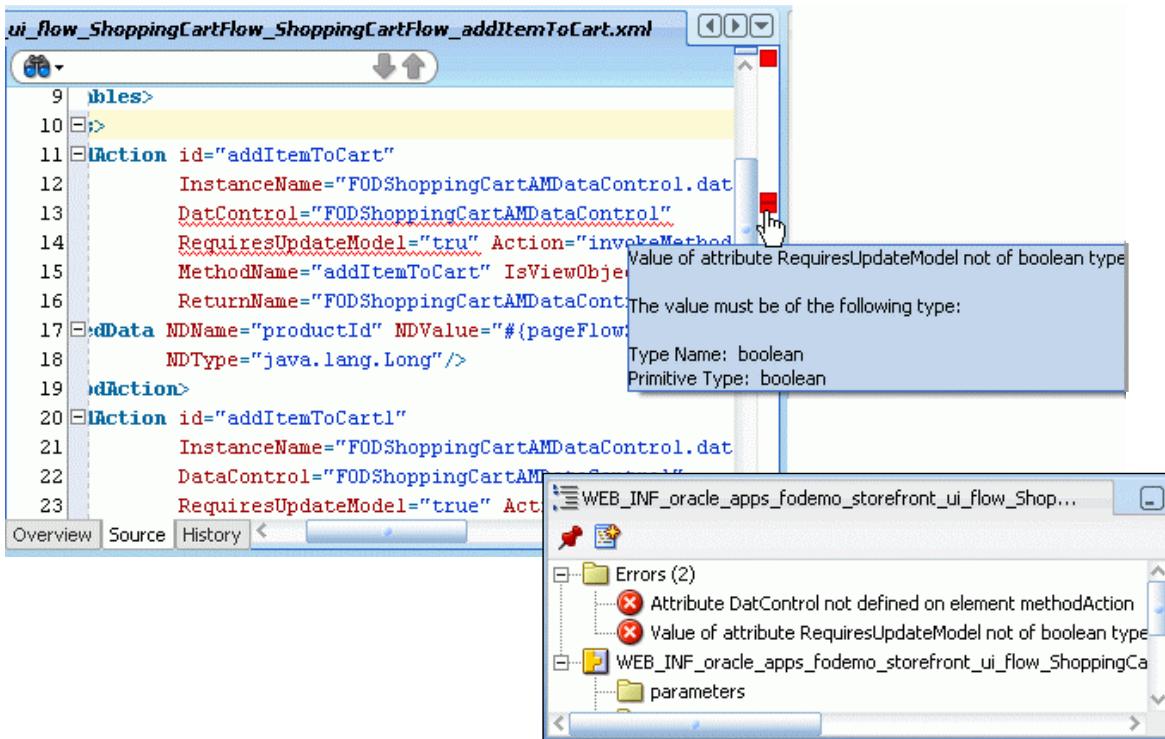
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Auditing Java Code

Auditing is the static analysis of code for adherence to rules and metrics that define programming standards. Auditing finds defects that make code difficult to improve and maintain. JDeveloper's auditing tools help you find and fix such defects. Code can be audited even when it is not compilable or executable.

Auditing is concerned with programming standards, rather than syntactic correctness. The focus of an audit is defined by a profile, which is essentially a set of audit rules and metrics. You can create and customize profiles, choose the rules to be used, and set parameters for individual rules. To do this, select **Tools > Preferences**, and then select **Audit** in the left pane.

Design-Time XML Validation



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

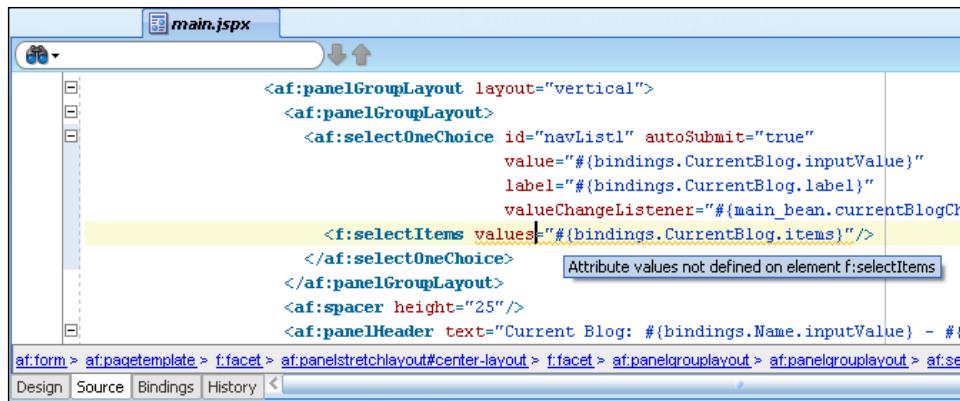
Design-Time XML Validation

All XML documents that have a schema or a document type definition (DTD) registered with JDeveloper (such as page definition files, .cpx files, page definition files, faces configuration files, web.xml, and so on) provide validation, as well as code insight, and adherence to audit rules.

When you create Web pages and work with the ADF data controls to create the ADF binding definitions in JDeveloper, the Oracle ADF declarative files that you edit must conform to the XML schema defined by Oracle ADF. When an XML syntax error occurs, the JDeveloper XML compiler immediately displays the error in the Structure window. You can double-click the error to open the file in the XML editor.

Design-Time JSPX Validation

- As you edit a page, the XML syntax is checked and errors and warnings are flagged.
- Errors are visible in Design, Source, and Bindings editors.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JSPX Validation

JSPX source and design time both flag errors as well as deprecated components, and perform other validations specific to JSPX and JSF.

Using Tools and Utilities

- JUnit
- JDeveloper Profiler
- Audit Profiles
- FileMon
- JDeveloper Debugger



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Testing Java Code with JUnit

- JUnit is an open source regression-testing framework.
- It is useful for creating tests to verify Java code.
- JDeveloper's JUnit extension provides wizards for creating test components.
- More information and examples:
 - SRDemo sample application
 - Toystore sample application
 - JDeveloper online documentation
 - <http://www.junit.org>



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using JUnit

JUnit is an open source regression-testing framework for Java (see <http://www.junit.org>). JUnit is more of a testing tool than a troubleshooting tool, but if you are developing an application of any complexity, it could be useful to create a suite of JUnit tests to quickly verify your Java code. For example, you could write a JUnit test for each ADF BC view object to instantiate the view object and execute its query.

JDeveloper provides two JUnit extensions: JUnit integration and JUnit integration for business components. You can install these extensions by selecting Tools > Preferences > Extensions > “Check for Updates.”

Unit Testing with JUnit

JUnit makes it easy for developers to write and run repeatable tests. Its features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test suites for organizing and running tests
- Graphical and textual test runners



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Unit Testing

Testing code as it is being developed, before it goes to QA, can improve code quality and drastically reduce maintenance and debug time.

A unit test:

- Is highly localized
- Is designed to test code only within a single package
- Does not test interactions between packages (functional testing does that)

JUnit is the most popular framework for unit testing, enabling developers to write and run repeatable tests. Its features include:

- Assertions to test expected results
- Fixtures for sharing test data
- Test suites that organize and run tests
- Graphical and textual test runners

JUnit works by performing many small tests, reporting success or failure of each; it is not intended to report multiple failures per test. Improving the testability of code usually results in better design.

JUnit enables you to automate tests for repeatability and to collect tests into suites for regression testing.

Using JDeveloper's Profiler

With the Profiler, you can:

- Monitor programs as they run
- Find bottlenecks and memory leaks

Two types of profiling:

- CPU: Method processing times
- Memory: Allocation and freeing of data objects



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Profiling a Project

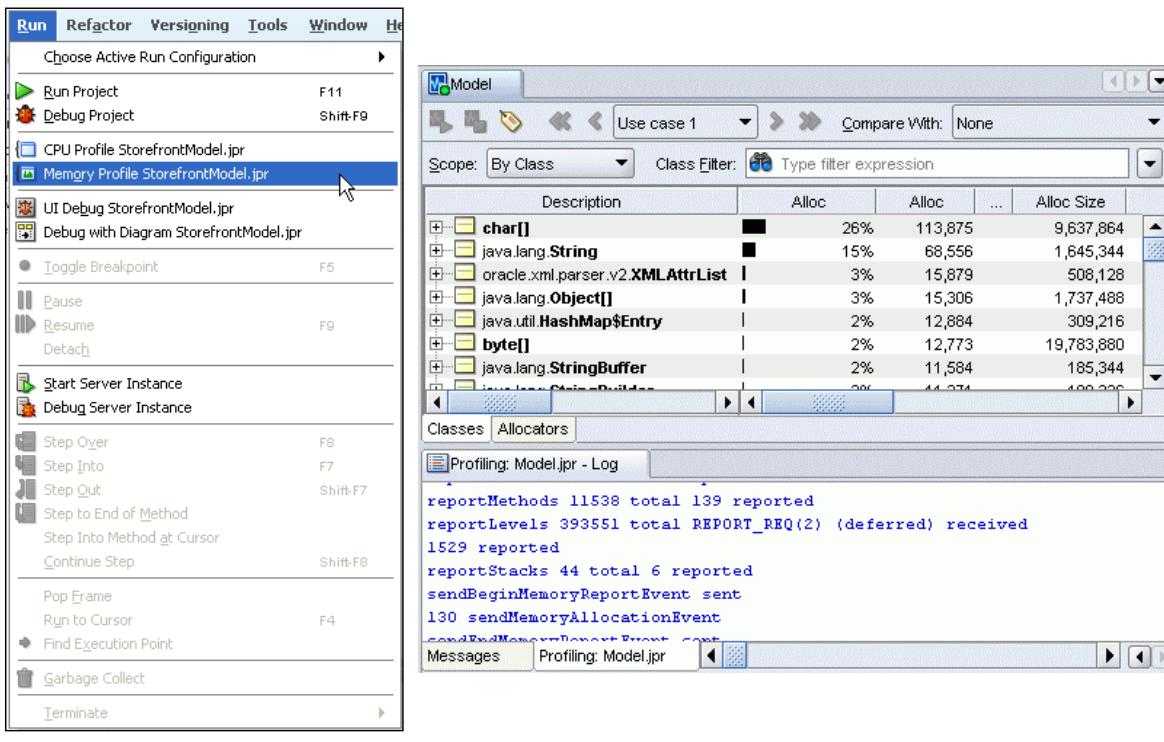
JDeveloper includes a profiler that you can use to gather statistics on your program so that you can more easily diagnose performance issues. With it you can examine and analyze the performance of your program. You can perform the following types of profiling:

- **CPU profiling:** Tabulates the processing time of specific methods
- **Memory profiling:** Tabulates the allocation and freeing of data objects

You can start a CPU or memory profiling session for a project or any of its runnable files, and you can have multiple sessions running, depending on the availability of system resources.

Note: For a simple CPU profile report that is sent to standard output, you can use the `-Xprof` JVM option. To use this utility, edit the project properties and select Run/Debug/Profile. Edit an existing configuration or create a new one. In the Launch Settings, add `-Xprof` to the Java Options field. You can also easily get garbage collection information by using the `-verbose:gc` JVM option.

Running the Profiler



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

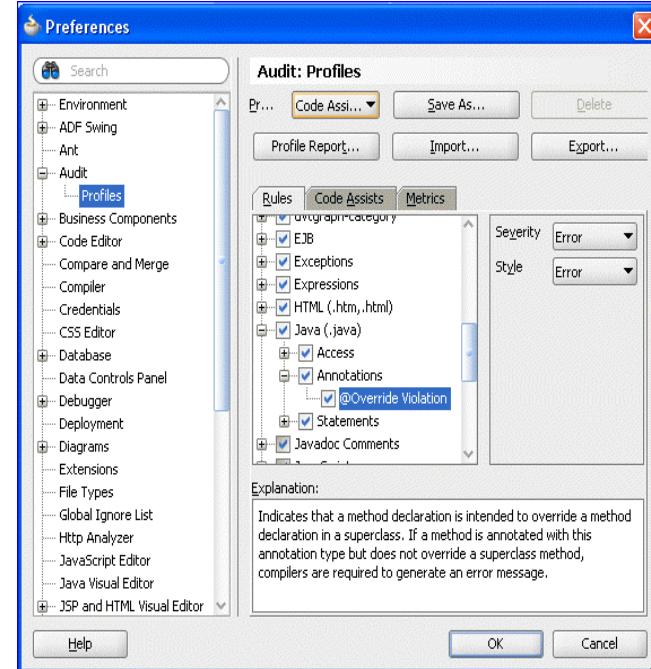
Running the Profiler

Profilers are invoked from the Run menu, as alternative ways to run a program. First from the Run menu, you select the active run configuration that corresponds to the profiler options that you defined. Then you select from the Run menu again to specify which profiler to run.

As you run the program (such as an ADF BC application module in the Business Components Browser), the profiler information is displayed in the JDeveloper IDE.

Using Audit Profiles

- An audit profile defines the rules, code assists, and metrics that will be used to analyze source code.
- Some audit profiles are predefined, but you can create your own.
- You modify an audit profile by enabling or disabling rules, code assists, and metrics, or by changing its configuration.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

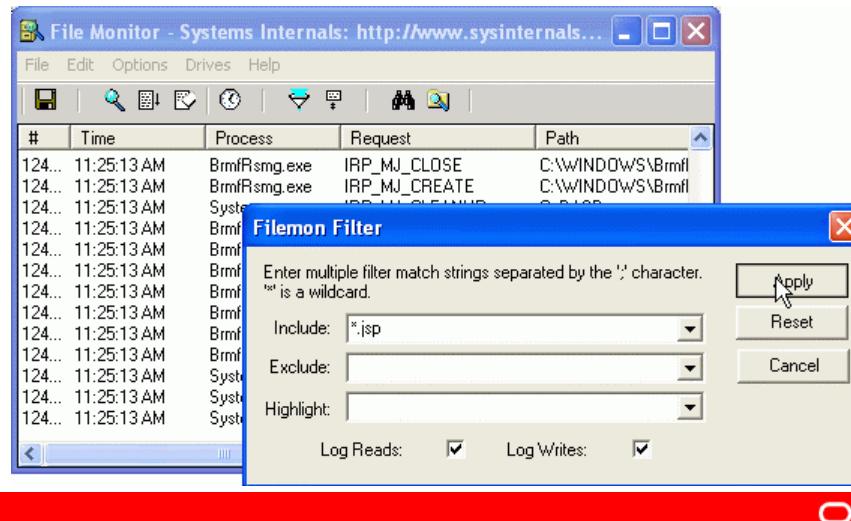
Using Audit Profiles

To create an audit profile:

1. From the main menu, select **Tools > Preferences**. The Preferences dialog box appears.
2. Select the **Audit > Profiles** page.
3. From the **Profile** drop-down menu, select a profile to copy.
4. Select the rules, assists, and metrics to enable in the new profile.
5. Configure the selected rules, assists, and metrics, if required.
6. Click **Save As**. Enter a name for the new profile and click **Save**. The new profile name is shown in the Audit Profiles preferences page's **Profile** box.
7. Click **OK**.

Identifying Search Paths on Windows with FileMon

- Is useful for troubleshooting CLASSPATH problems
- Shows you the path that your running application is looking in for classes and files



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using FileMon

FileMon is a free Windows utility that you can download from <http://www.sysinternals.com>. Its purpose is to show where a running application searches for the files that it needs, so it is useful for resolving problems with CLASSPATH settings.

To use FileMon, perform the following steps:

1. Run `filemon.exe`.
2. Set a filter for the file type in which you are interested. FileMon produces a lot of output; you probably want to set a filter to restrict the output to a more readable size. To set a filter, click the Filter button on the FileMon toolbar. For example, you can use the filter `**.jsp` to monitor all `.jsp` files that your application is trying to open.
3. If desired, save the output to a log file.

Note: You can also get information about the classes that load by using the JVM option `-verbose:class`.

Using the JDeveloper Debugger

- The Debugger is very useful for pinpointing problems in your application.
- Set source breakpoints to pinpoint problems in the custom code.
- Set exception breakpoints to stop when a particular exception is thrown.
- When a breakpoint is encountered at run time, you can step through the code and view the values of the variables.
- To run an application in debug mode, right-click and select Debug.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the JDeveloper Debugger

JDeveloper's integrated debugger is very useful for finding and fixing problems in your code.

The following are some key places to set breakpoints:

- Source breakpoint in the `doIt()` method of the `JUCtrlActionBinding` class (`oracle.jbo.uicli.binding` package): This is the method that executes when any ADF action binding is invoked, and you can step into the logic and look at parameters if relevant.
- Method breakpoints in:
 - `oracle.jbo.server.ViewObjectImpl.executeQueryForCollection` method: This is the method that is called when a view object executes its SQL query.
 - `oracle.jbo.server.ViewRowImpl.setAttributeInternal` method: This is the method that is called when any view row attribute is set.
 - `oracle.jbo.server.EntityImpl.setAttributeInternal` method: This is the method that is called when any entity object attribute is set.

Understanding Breakpoint Types

Type	Breaks when...
Exception	An exception of this class (or a subclass) is thrown
Source	A particular source line in a particular class in a particular package is run
Method	A method in a given class is invoked
Class	Any method in a given class is invoked
Watchpoint	A given field is accessed or modified



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Understanding Breakpoint Types

There are several different types of breakpoints, each with different uses:

- **Exception:** An exception of this class (or a subclass) is thrown. This is useful when you do not know where the exception occurs, but you know what kind of exception it is, such as a `java.lang.NullPointerException`. The check box options enable you to control whether to break on caught or uncaught exceptions of this class. The Browse button helps you find the fully qualified class name of the exception. The Exception Class combo box remembers the most recently used exception breakpoint classes. Note that this is the default breakpoint type when you create a breakpoint in the breakpoints window.
- **Source:** A particular source line in a particular class in a particular package is run. You can create a source breakpoint in the New Breakpoint window, but it is usually easier to create it by clicking in the breakpoint margin in the editor at the left of the line you want to break on.
- **Method:** A method in a given class is invoked. This is handy to set breakpoints on a particular method that you might have seen in the call stack while debugging a problem. Of course, if you have the source, you can set a source breakpoint wherever you want in that class, but this kind of breakpoint enables you to stop in the debugger even when you do not have a source for a class.

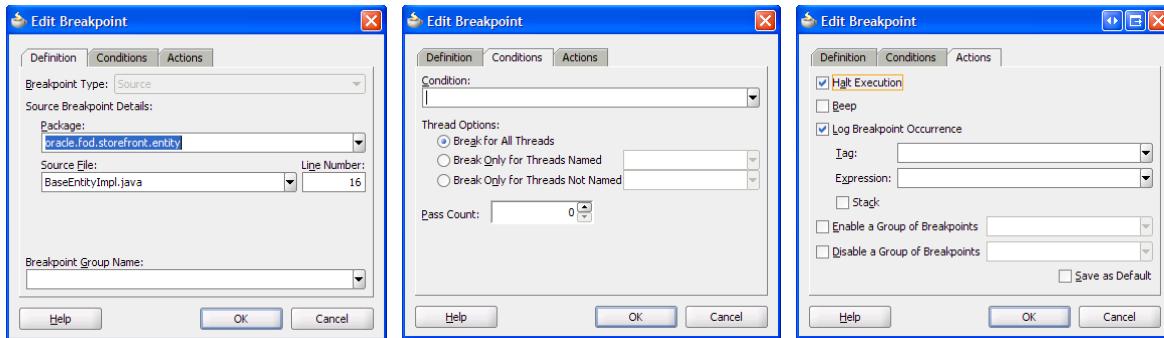
Understanding Breakpoint Types (continued)

- **Class:** Any method in a given class is invoked. This can be handy when you might only know the class involved in the problem, but not the exact method you want to stop on. Again, this kind of breakpoint does not require source. The Browse button helps you quickly find the fully qualified class name you want to break on.
- **Watchpoint:** A given field is accessed or modified. This can be helpful to find a problem if the code inside a class modifies a member field directly from several different places (instead of going through setter or getter methods each time). You can stop the debugger in its tracks when any field is modified. You can create a breakpoint of this type by using the Toggle Watchpoint menu item on the shortcut menu when pointing at a member field in the source of the class.

To see the Debugger Breakpoints window, use the View > Breakpoints menu choice from the main JDeveloper menu, or optionally the key accelerator for this: Ctrl + Shift + R. You can create a new breakpoint by selecting Add Breakpoint from the context menu anywhere in the Breakpoints window or by clicking the plus sign on the Breakpoints toolbar.

Using Breakpoints

- You can create groups of breakpoints that can be enabled or disabled all at once.
- You can use conditional breakpoints. Examples:
 - value instanceof oracle.jbo.domain.Date
 - status.equalsIgnoreCase ("shipped")
 - i > 50
- You can use actions other than stop with breakpoints.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Breakpoints

You can use breakpoints for more than just stopping code execution. JDeveloper supports advanced breakpoint actions and conditions for fine-tuning when a breakpoint is triggered and what action happens. Virtually any Boolean expression can be a breakpoint condition, and it can be applied to all threads, or a specific set of named threads.

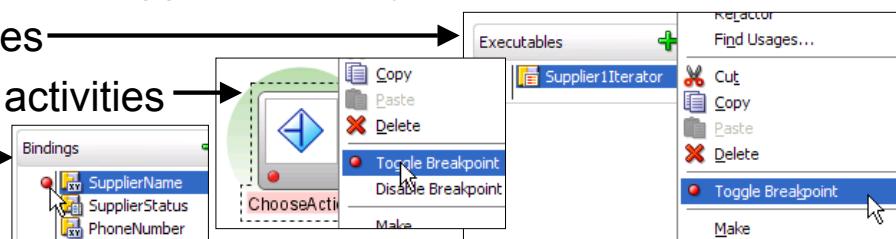
The first of the conditional breakpoint examples in the slide breaks only when the variable value is a Date type. The second breaks only when the status is “shipped.” The third breaks only when the variable *i* is greater than 50. The third example (*i*>50) is particularly useful if you have a breakpoint inside a loop (with *i* as the loop variable) but you know your problem occurs only when the loop has executed a number of times.

For actions, breakpoints typically just break into the code. However, you can beep, log a message or an expression result to the log console, dump the stack, or enable or disable a group of breakpoints.

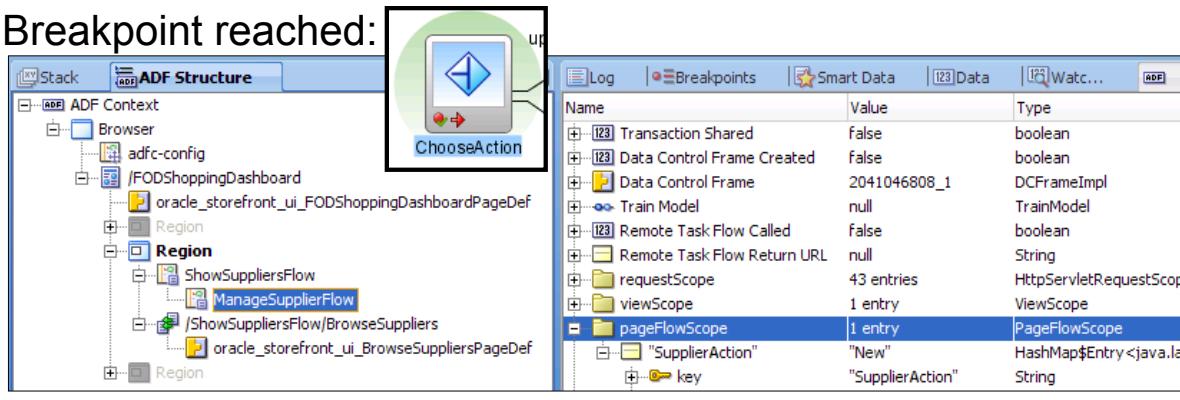
ADF Framework Debugging

The Declarative Debugger enables you to set breakpoints on:

- Executables
- Task flow activities
- Bindings



Breakpoint reached:



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Declarative UI Debugging

You use the ADF Declarative Debugger in JDeveloper to set breakpoints on ADF task flow activities, page definition executables, and method, action, and value bindings. Instead of needing to know all the internal constructs of the ADF code, such as method names and class names, you can set breakpoints at the highest level of object abstraction.

You can set or remove such breakpoints by right-clicking task flow activities or bindings and selecting Toggle Breakpoint; you can also set or remove breakpoints on executables or bindings by clicking the margin in the binding editor.

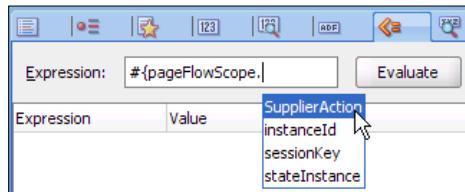
For example, when you set a breakpoint on the Choose Action router activity in the CheckoutFlow, a red dot icon appears on the activity, as shown in the slide. When the breakpoint is reached, the application pauses and the icon changes, as shown.

After the application pauses at the breakpoint, you can view the run-time structure of the objects as a tree in the ADF Structure window, as shown in the slide. The ADF Data window displays a list of data for a given object selected in the ADF Structure window.

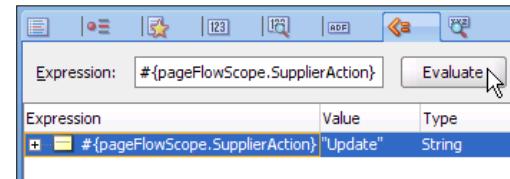
Using the EL Evaluator

The Expression Language evaluator enables:

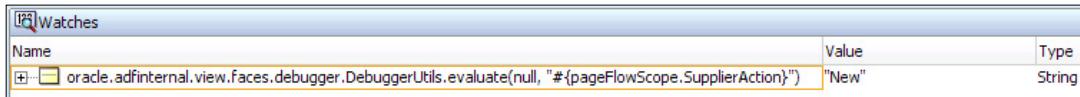
- Introspection of the values between the view and model
- Watches based on EL



Using the discovery function



An evaluated expression



An evaluated expression on the watch list displays the changed value.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the EL Evaluator

When you debug and reach a breakpoint, you can use the EL Evaluator to examine the value of an EL expression. The EL Evaluator appears as a tab in the debugger window area; you can select View > Debugger > EL Evaluator if it is not visible.

Enter an EL expression in the input field. When you enter # { in the field, a discovery function helps you to select the correct expression to evaluate, and auto-completion also facilitates expression entry. You can evaluate several EL expressions at the same time by separating them with semicolons. After you have entered the expression or expressions, click Evaluate. If the EL expression no longer applies within the current context, the expression evaluates to null.

The EL Evaluator is different from the Watches window in that EL evaluation occurs only when stopped at a breakpoint, not when stopped at subsequent debugging steps.

ADF Structure Window and ADF Data Window

During debugging:

- ADF Structure window shows items in the current viewport
- ADF Data window shows data within:
 - The item selected in the ADF Structure window
 - The ADF context
 - Scoped variables



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

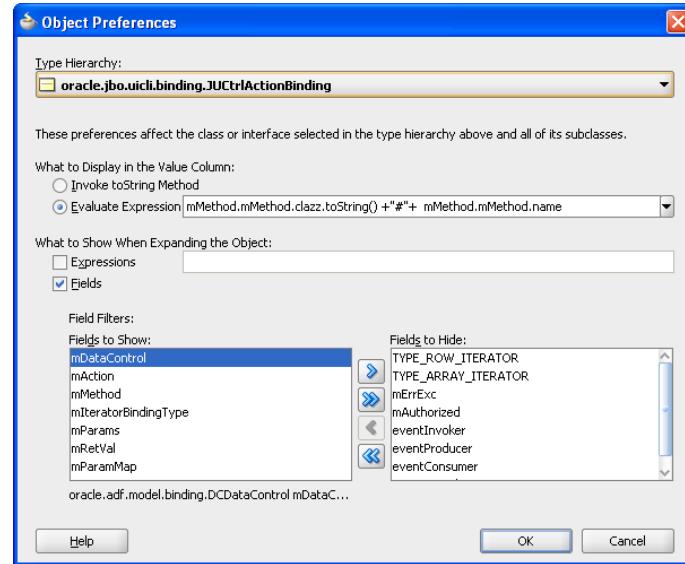
ADF Structure Window and ADF Data Window

The ADF Declarative Debugger provides standard debugging features such as the ability to examine variable and stack data. When an application pauses at any breakpoint (ADF Declarative or Java code breakpoint), you can examine the application status using a variety of windows:

- You can check where the break occurs in the Breakpoint window.
- You can check the call stack for the current thread using the Stack window. When you select a line in the Stack window, information in the Data window, Watches window, and all Inspector windows is updated to show relevant data.
- You can use the Data window to display information about arguments, local variables, and static fields in your application.
- The ADF Structure window displays the run-time structure of the project. The ADF Data window automatically changes its display information based on the selection in the ADF Structure window. For example, if a task flow node is selected, the ADF Data window displays the relevant debugging information for task flows.

Object Preferences

- Filter out the fields that are displayed in the debugger.
- Change the default value displayed in the debugger.
- Show expressions as details instead of fields.
- Customize types throughout the type hierarchy.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Filtering Your View of Class Members

An excellent but often overlooked feature of the JDeveloper Debugger is the ability to filter for the members that you want to see in the debugger window for any class. In the debugger's Data window, pointing at any item and selecting Object Preferences from the context menu opens a dialog box that enables you to customize which members appear in the debugger and (more importantly sometimes) which members *do not* appear. These preferences are set by class type and can really simplify the amount of scrolling you need to do in the debugger Data window. This is especially useful while debugging when you might be interested only in a few members of a class.

Displaying fewer fields narrows your focus when debugging and may make it easier to locate and isolate potential problems in your program. For example, you can create filters for classes in the Data windows so that the debugger displays only the fields of interest to you. This drastically reduces clutter and allows you to find the relevant data more quickly.

Using Oracle ADF Source Code for Debugging

Adding ADF source code enables you to:

- Access Javadocs in the code editor
- Set breakpoints on Oracle code as well as your own
- Access symbolic names via Debug libraries



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Oracle ADF Source Code for Debugging

Adding Oracle ADF source code to your application debugging session will:

- Provide access to the JDeveloper Quick Javadocs feature in the code editor. Without the source code, you will have only standard Javadocs.
- Enhance the use of breakpoints by showing the Oracle source code that is being executed when the breakpoint is encountered. You can also set breakpoints more easily by clicking the margin in the source code line you want to break on. Without the source code, you will have to know the class, method, or line number in order to set a breakpoint within Oracle code.
- Allow you to see the values of all local variables and member fields in the debugger.

If you have valid Oracle ADF support, you can obtain the complete source code for Oracle ADF by opening a service request with Oracle Worldwide Support. You can request a specific version of the Oracle ADF source code.

Setting Up Oracle ADF Source Code for Debugging

To use source code for debugging, you can:

- Add the source zip into the user library
- Add the library to a project



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Setting Up Oracle ADF Source Code for Debugging

After you receive or download the source file archive, extract it to a working directory. You create libraries of the source code Java Archive (JAR) files in this directory, and then add the libraries to your JDeveloper projects.

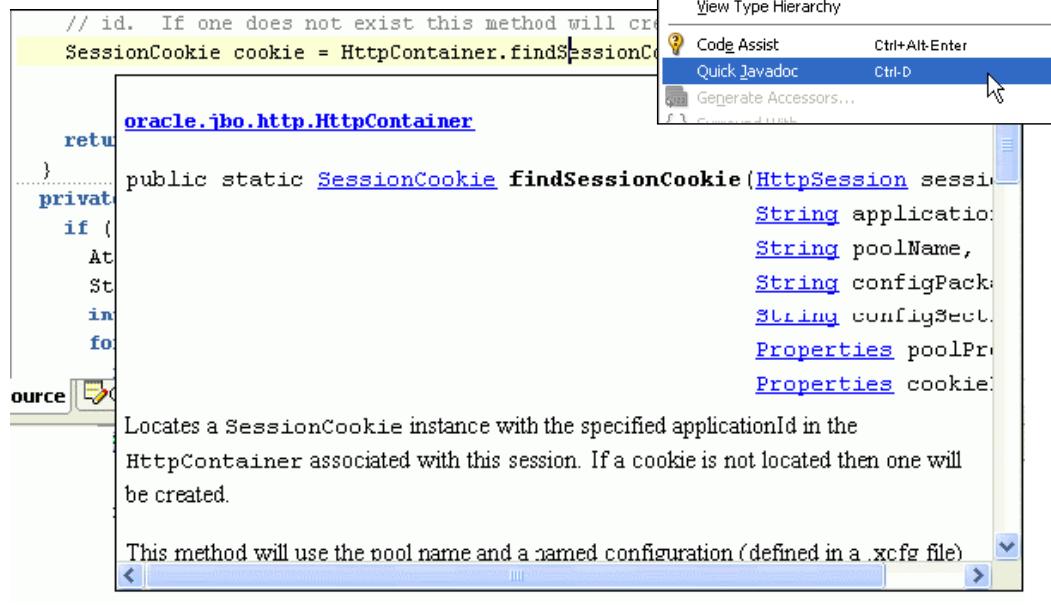
To add the ADF source zip file into the user library, perform the following steps:

1. Select Tools > Manage Libraries to display the Manage Libraries dialog box.
2. With the Libraries tab selected, click New.
3. In the Create Library window, enter a library name for the source that identifies the type of library and select the Source Path node in the tree structure. Do not enter a value for the Class Path. Click Add Entry.
4. In the Select Path Entry window, browse to the source file directory and select the source zip file. Click Select.
5. In the Create Library window, verify that the Source Path entry has the correct path to the source zip file, and deselect the “Deployed by Default” check box. Click OK.
6. Click OK to close the Manage Libraries dialog box.

After the source library has been added to the list of available libraries, add it to the project you want to debug, as you would add any library.

Using Quick Javadoc

Right-click a method and select Quick Javadoc.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

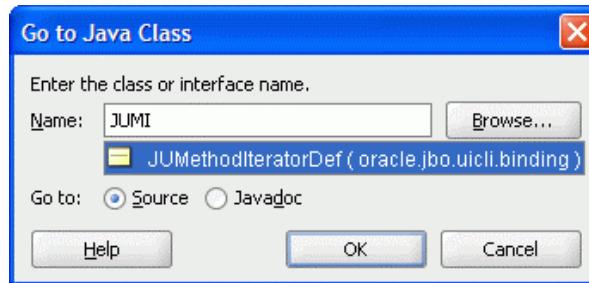
Using Quick Javadoc

After you have added the ADF Source library to your project, you instantly have access to the helpful Quick Javadoc feature that the JDeveloper code editor makes available. You can right-click a class or method, or press Ctrl + D, to invoke it. This not only helps when writing code, but also when debugging it.

Setting Breakpoints in Source Code

To set a breakpoint in Oracle code, perform the following steps:

1. Press Ctrl + – (the minus sign).
2. Enter an Oracle ADF class name or its uppercase letters.
3. Set breakpoints in the source file that JDeveloper opens.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Setting Breakpoints in Source Code

After adding the source code to your project, you can debug any Oracle ADF code for the current project in the same way as you have been doing for your own Java code. This means that you can press Ctrl + – (the minus sign) to open a “Go to Java Class” dialog box, in which you can enter any class name in Oracle ADF. When you select the class and click OK, JDeveloper opens the source file automatically, so you can set breakpoints as desired.

Hint: In the “Go to Java Class” dialog box, you can enter just the uppercase letters from a class name. For example, to find `JUMethodIteratorDef`, you can enter `JUMI` and the class is located.

Using Common Oracle ADF Breakpoints

- ADF breakpoints are useful for debugging declarative functionality.
- In the `oracle.jbo` package, you can set breakpoints on:
 - `JboException`
 - `DMLEception`
 - `uicli.binding.JUCtrlActionBinding.doIt()`
 - `server.ViewObjectImpl.executeQueryForCollection()`
 - `server.ViewRowImpl.setAttributeInternal()`
 - `server.EntityImpl.setAttributeInternal()`



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Common Oracle ADF Breakpoints

If you loaded Oracle ADF source code, you can use additional breakpoints to debug your application. By looking at the stack window when you hit these breakpoints and stepping through the source, you can get a better idea of what is going on. These breakpoints are useful for debugging declarative functionality. Commonly used ADF breakpoints include:

- `oracle.jbo.JboException`: Exception breakpoint that is the base class of all ADF BC run-time exceptions
- `oracle.jbo.DMLEception`: Exception breakpoint that is the base class for exceptions originating from the database
- `doIt()`: Method exception from the class `oracle.jbo.uicli.binding.JUCtrlActionBinding`: Executes when any ADF action binding is invoked, so you can step into the logic and look at parameters if relevant
- `oracle.jbo.server.ViewObjectImpl.executeQueryForCollection`: Method exception; called when a VO executes its SQL query
- `oracle.jbo.server.ViewRowImpl.setAttributeInternal`: Method exception; called when any view row attribute is set
- `oracle.jbo.server.EntityImpl.setAttributeInternal`: Method exception; called when any entity object attribute is set

Debugging Interactions with the Model Layer

Controlled by two classes:

```
oracle.adf.controller.faces.lifecycle.FacesPageLifecycle  
oracle.adf.controller.v2.lifecycle.PageLifecycleImpl
```

Set breakpoints if encountering problems such as:

- Components not displaying correctly with complete data
- Validation errors not rendering properly



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Debugging Interactions with the Model Layer

The processing of your JSF page in combination with Oracle ADF Model is controlled by two classes:

- `oracle.adf.controller.faces.lifecycle.FacesPageLifecycle`: Provides the default implementation of the phase of the ADF life cycle.
- `oracle.adf.controller.v2.lifecycle.PageLifecycleImpl`: Provides the starting point for creating the objects of the Oracle ADF binding context

`FacesPageLifecycle` implements certain methods of `PageLifecycleImpl` to provide customized error-handling behavior for ADF Faces applications. A good place to set a breakpoint in the `FacesPageLifecycle` class is on the `prepareModel()` method because it initiates the first phase of the ADF life cycle. Generally, however, you set breakpoints on `PageLifecycleImpl` because it is the starting point.

The successful interaction between the Web page and these objects of the Oracle ADF binding context ensures that the page's components are displayed with correct and complete data, that methods and actions produce the desired result, and that the page renders properly with the appropriate validation errors.

Correcting Failures to Display Data

- To debug all executables:
`oracle.adf.model.binding.DCBindingContainer
internalRefreshControl(int, boolean)`
- To debug the method iterator:
`oracle.jbo.uicli.binding.JUMethodIteratorDef
initSourceRSI()`
- To debug an attribute binding:
`oracle.jbo.uicli.binding.JUCtrlValueBinding
getInputValue()`



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Correcting Failures to Display Data

After BindingContainer is created by BindingContext, the ADF life cycle initiates the Prepare Model and the Render Model phases before data can be displayed on the Web page. Before the bindings are resolved and data can appear on the Web page, the page parameters must be set and the Iterator and Method executables must be refreshed by executing the named service methods and ADF iterator bindings.

The ADF life cycle enters the Prepare Model phase by calling `BindingContainer.refresh(PREPARE_MODEL)`. During the Prepare Model phase, BindingContainer page parameters get prepared and then evaluated. Next, BindingContainer executables get refreshed based on the order of entry in the pagedef.xml file's `<executables>` section and on the evaluation of their Refresh and RefreshCondition properties (if present). When an executable leads to an iterator binding refresh, the corresponding data control is executed, and that leads to the execution of one or more collections in the service objects. If an iterator binding fails to refresh, a JBO exception is thrown and the data is not available to display.

Correcting Failures to Display Data (continued)

To debug all executables for the binding container, perform the following steps:

1. In the `oracle.adf.model.binding.DCBindingContainer` class, set a breakpoint on `internalRefreshControl(int, boolean)` as the entry point to debug the executables.
2. In the `oracle.adf.model.binding.DCIteratorBinding` class, set a breakpoint on `callInitSourceRSI()` to halt processing and step into the method.
3. When processing pauses, look for `callInitSourceRSI()` in the Stack window. The result displayed in the Smart Data window should show the result that you expect.

When your Web page fails to display data from a method iterator binding, you can drill down to the entry point in `JUMethodIteratorDef.java` and its nested class `JUMethodIteratorBinding` to debug its execution.

To debug the method iterator executable for the binding container, perform the following steps:

1. In the `oracle.jbo.uicli.binding.JUMethodIteratorDef` class, set a breakpoint on `initSourceRSI()` as the entry point to debug a method iterator binding executable.
2. Set a breakpoint on `invokeMethodAction()` to halt processing and step into the method.

Note that if the method returns a valid collection or a bean, then that object becomes the data source for the rowset iterator that this iterator binding is bound to. For bean data controls, an instance of `DCRowSetIteratorImpl` is created to provide the rowset iterator functionality for the iterator binding to work with. (Note that for ADF Business Components, this method would ideally return an ADF Business Components rowset iterator so that ADF Business Components can manage the state of the collection.)

3. When `initSourceRSI()` returns a rowset iterator, pause processing and look for `mProvider` in the Smart Data window. The `mProvider` variable is the data source fetched for this rowset iterator. If the method completes successfully, it should show a collection bound to an iterator or a bean.

When the executable that produced the exception is identified, check that the `<executables>` element in the page definition file specifies the correct attribute settings.

Whether the executable is refreshed during the Prepare Model phase depends on the value of Refresh and RefreshCondition (if they exist). If Refresh is set to `prepareModel`, or if no value is supplied (that is, it uses the default, `ifneeded`), then the RefreshCondition attribute value is evaluated. If no RefreshCondition value exists, the executable is invoked. If a value for RefreshCondition exists, then that value is evaluated, and if the return value of the evaluation is `true`, then the executable is invoked. If the value evaluates to `false`, the executable is not invoked. The default value always enforces execution.

Correcting Failures to Display Data (continued)

Fixing Render Value Errors Before Submit

During the `prepareRender` phase of the ADF life cycle, the bindings determine the data to display, and properties on the bindings determine the conditions in which to display the data. When the Web page is rendered the first time, each EL expression that points to a binding gets resolved by the `BindingContainer` instance for that page.

Based on the appropriate expression values such as `format`, `isEnabled`, and `isViewable`, the data value for a binding is returned from `BindingContainer`. If the binding is unable to return the data, a JBO exception is thrown.

To debug the binding resolution for the binding container, perform the following steps:

1. In the `oracle.jbo.uicli.binding.JUCtrlValueBinding` class, set a breakpoint in `getInputValue()` and step into the method.
2. If `getInputValue()` returns an error, pause processing and look for the binding name in the Data window.
3. Continue stepping into `getInputValue()` and look for a return value in the Data window that you expect for the current row that this binding represents.

When the binding that produced the exception is identified, check whether the `<bindings>` element in the page definition file specifies the correct attribute settings.

In case of submit, again the life cycle first looks up and prepares the `BindingContainer` instance. If the life cycle finds a state token that was persisted for this `BindingContainer`, it asks the `BindingContainer` to process this state token.

Processing the state token restores the variable values that were saved in the previous rendering.

If you need to debug this processing, set breakpoints in

`DCIIteratorBinding.processFormToken()` and

`DCIIteratorBinding.buildFormToken()`. After this, all posts are applied to the bindings through `setInputValue()` on the value bindings.

Correcting Failures to Invoke Actions and Methods

- Actions are ignored if an executable or its target binding is not executed.
- You can debug the action or method invocation by breaking on `DCDataControl.invokeOperation()`, which is the entry point for action and method execution.
- You can debug the invocation of coded methods by breaking on `DCGenericDataControl.invokeOperation()`.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Correcting Failures to Invoke Actions and Methods

When the executables are refreshed, actions and custom methods may be invoked on the page. At this stage, the corresponding action or method binding is refreshed. If an executable or its target binding is not executed, the action is ignored.

The entry point for action and method execution is the `DCDataControl.invokeOperation()` method. Debugging on that method enables you to work with the same method that the data control uses to invoke the method. This is preferred because some adapter data controls can interpret the method name in a custom way rather than leave it to ADF to call the method.

To debug the action or method invocation for the binding container, perform the following steps:

1. In the `oracle.adf.model.binding.DCDataControl` class, set a breakpoint on `invokeOperation()` as the entry point to debug an action or method invocation.
2. When processing pauses, step into the method to verify that `instanceName` in the Data window shows that the method being invoked is the intended method on the desired object.
3. Verify that `args` in the Data window shows that the parameter value for each parameter being passed into your method is as expected.

Correcting Failures to Invoke Actions and Methods (continued)

To debug an invocation of a coded method for the binding container, perform the following steps:

1. In your class, set a breakpoint on the desired method.
2. In the `oracle.adf.model.generic.DCGenericDataControl` class, set a breakpoint on `invokeMethod()` to halt processing before looking into the Data window.
3. When processing pauses, step into the method to verify that `instanceName` in the Data window shows that the method being invoked is the intended method on the desired object.
4. Verify that `args` in the Data window shows that the parameter value for each parameter being passed into your method is as expected.

If the debugger does not reach a breakpoint that you set on an action in the binding container, then the error is most likely a result of the way the executable's Refresh and RefreshCondition attributes were defined. Examine the attribute definitions.

Whether the `<invokeAction>` executable is refreshed during the Prepare Model phase, depends on the value of Refresh and RefreshCondition (if they exist). If Refresh is set to `prepareModel`, or if no value is supplied (that is, it uses the default `ifneeded`), then the RefreshCondition attribute value is evaluated. If no RefreshCondition value exists, the executable is invoked. If a value for RefreshCondition exists, then that value is evaluated, and if the return value of the evaluation is `true`, then the executable is invoked. If the value evaluates to `false`, the executable is not invoked. The default value always enforces execution.

Debugging Life Cycle Events: Task Flows

- Before a task flow is called:
`oracle.adfinternal.controller.activity.TaskFlowCallActivityLogic.execute()`
- Before a task flow's input parameters are resolved:
`oracle.adfinternal.controller.activity.TaskFlowCallActivityLogic.getInputValues()`
- Before and after a task flow returns:
`oracle.adfinternal.controller.activity.TaskFlowReturnActivityLogic.execute()`

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Debugging Life Cycle Events: Task Flows

This slide describes where to set breakpoints if you want to stop at specific points in a task flow.

Debugging Life Cycle Events: Parameters and Methods

- For debugging the ADF controller's interpretation of the navigation routing:
`oracle.adfinternal.controller.engine.
ControlFlowEngine.doRouting()`
- For debugging before the view activity's input parameters are evaluated:
`oracle.adfinternal.controller.
application.PageParameterPagePhaseListener
.beforePhase()`
- For calling an ADF controller method action:
`oracle.adfinternal.controller.activity.
MethodCallActivityLogic.execute()`



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Debugging Life Cycle Events: Parameters and Methods

The `execute()` method for each activity type's implementation logic is always called from `oracle.adfinternal.controller.engine.ControlFlowEngine.doRouting()`. If you want to track your progression through the page flow graph, this is a useful place to set a breakpoint.

Debugging Life Cycle Events: Switching Between the Main Page and Regions

To keep track of the context switching between the main page and regions, set a breakpoint in:

```
oracle.adfinternal.controller.state.  
RequestState.setCurrentViewPortContext()
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Obtaining Help

Sources of help include:

- Online Help
- Javadoc
- Support (My Oracle Support)
- Forums
- Blogs
- Internet Search



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Obtaining Help

The information that you are able to obtain from troubleshooting can be used to search various sources of help. You can search for error numbers, exceptions, and so on.

Requesting Help

Post your request to the appropriate forum:

- Use a meaningful subject line.
- Describe exact steps to reproduce the problem.
- Include the exact error information (error number, exception) if you receive an error message.
- Include a stack trace if appropriate.
- Include a test case if possible.
- List technologies and product version numbers.
- Provide concrete examples rather than abstract descriptions.
- List the troubleshooting steps that you have tried and describe how they affected the problem.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Requesting Help

If you ask a forum for help, try to include as much relevant information as possible. You are not expected to include all the information listed for every problem, just what is appropriate. Give your post a meaningful subject line. For example, “Need help” is less likely to get anyone to look at your post than “Unable to refresh detail when clicking tree node.”

Including stack trace information in your posting can be extremely useful. JDeveloper’s Stack window makes communicating this information easy. Whenever the debugger is paused, you can view the Stack window to see the program flow as a stack of method calls that got you to the current line, and use the context menu to set some helpful options:

- **Preferences:** You can set the Stack window to include the line number information as well as the class and method name that are there by default.
- **Export:** Save the current stack information to an external text file.

Provide concrete examples in your request for help rather than abstract descriptions. For example, you might start a problem description as follows: “My page contains an input form for editing customer information. Each customer can have one or more phone numbers.” This is easier to understand than an abstract description, such as: “My page contains an input form for editing an object, one of whose attributes can have one or more values.”

When you solve your problem, you should post the solution to the forum thread, so that others can benefit from it.

Summary

In this lesson, you should have learned how to use:

- Tools for logging and diagnostics
- Design-time code validation
- FileMon and JUnit
- JDeveloper Profiler
- JDeveloper Debugger
- Sources of help



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 9 Overview: Troubleshooting

This practice covers the following topics:

- Discovering Application Problems
- Setting Breakpoints in the Debugger
- Running the Application Module in Debug Mode



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 9 Overview

Most techniques for troubleshooting are used mainly for debugging a user interface. However, you can run the JDeveloper Debugger with the Business Components Browser, which enables you to troubleshoot problems with your ADF BC model apart from running a Fusion Web application. In the practices for this lesson, you debug a client method in an application module, and you explore features of the JDeveloper Debugger.

10

Understanding UI Technologies

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Describe the use of Web browsers and HTML
- Explain how Java has come into widespread use as a language for developing Web applications
- Describe the function of servlets and JSPs
- Define JavaServer Faces
- Explain the JSF component architecture
- List some JSF component types included in the standard implementation
- Describe the purpose of backing beans
- Explain the use of managed beans
- Describe the JSF life cycle
- Explain how ADF Faces augments the JSF life cycle



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

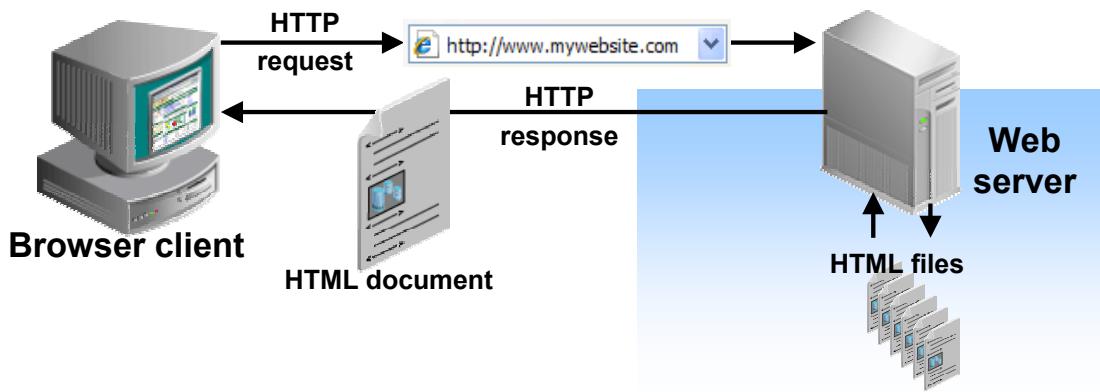
Lesson Aim

ADF Faces, used in this course, is built on top of the JavaServer Faces technology. This lesson describes UI technologies, especially JSF, and explains how ADF enhances the ability to build rich user interfaces.

Enabling the World Wide Web with HTML and HTTP

HTML: A markup language for defining Web pages

HTTP: A stateless request/response protocol for serving Web content, such as HTML pages



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Enabling the World Wide Web with HTML and HTTP

HyperText Markup Language (HTML) is a markup language for defining Web pages. It consists of tags that describe the elements of a document, such as links, text, paragraphs, lists, and forms. The ability to define hyperlinks to other Web pages was instrumental in the development of the World Wide Web.

HTML documents are transmitted to a Web browser from a Web server via the HyperText Transfer Protocol (HTTP), which can also serve images, sound, and other types of content. HTTP is simply a set of rules and standards for exchanging files on the Web.

HTTP is a request/response protocol. It sends a request from the client browser to the Web server and returns the response in the form of the requested HTML document or other content. It is a stateless protocol; each request-and-response set has no knowledge of previous requests. With HTTP it is difficult to implement Web sites that react intelligently to user input. One of the technologies intended to overcome this difficulty is Java.

Describing the Java Programming Language

Java:

- Is both a platform and an object-oriented language
- Was originally designed by Sun Microsystems for consumer electronics
- Contains a class library
- Uses a virtual machine for program execution
- Provides the following benefits:
 - Object-oriented
 - Interpreted and platform independent
 - Dynamic and distributed
 - Multithreaded
 - Robust and secure



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

What Is Java?

Designed by Sun Microsystems

Originally developed by Sun Microsystems, Inc., Java is a platform and an object-oriented programming language. It was created by James Gosling for use in consumer electronics. Because of the robustness and platform-independent nature of the language, Java soon moved beyond the consumer electronics industry and found a home on the World Wide Web. Java is a platform, which means that it is a complete development and deployment environment.

Class Libraries

Java contains a broad set of predefined classes that contain attributes and methods that handle most of the fundamental requirements of programs. Window management, input/output, and network communication classes are included in the Java Development Kit (JDK). The class library makes Java programming significantly easier and faster to develop when compared with other languages. JDK also contains several utilities to facilitate development processes. These utilities handle operations such as debugging, deployment, and documentation.

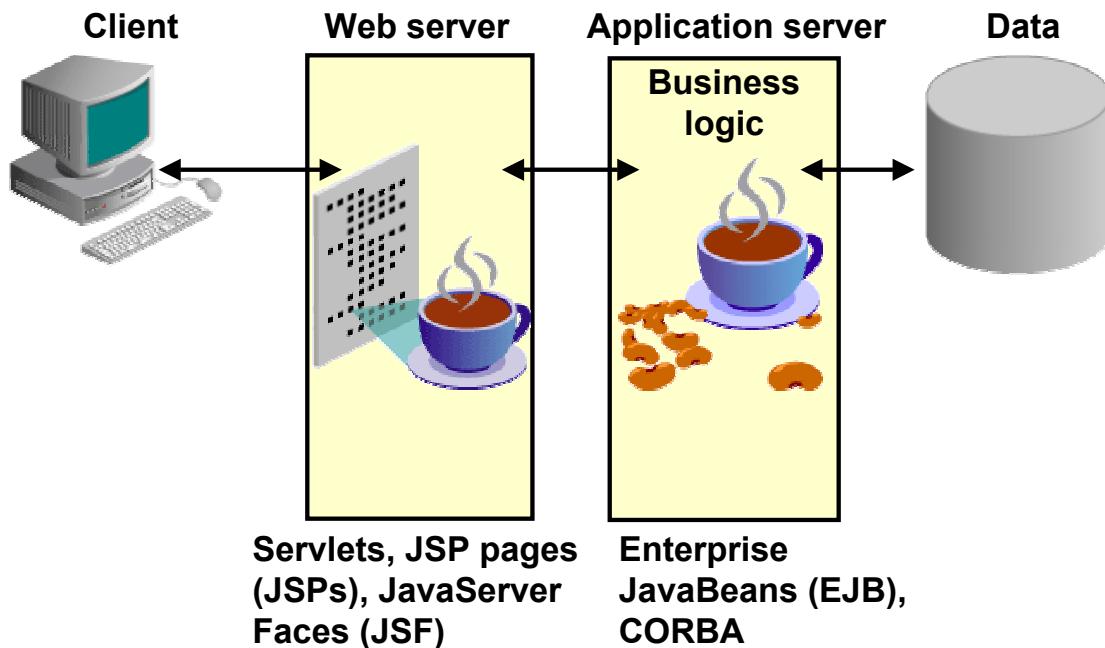
What Is Java? (continued)

Benefits of Java

Java provides several benefits, including the following:

- **Object-Oriented:** An object is an entity that has data attributes, plus a set of functions that are used to manipulate the object. Java is a strongly typed language, which means that everything in Java must have a data type—there are no variables that assume data types (as in Visual Basic).
- **Interpreted and Platform Independent:** One of the key elements of Java is platform independence. A Java program that is written on one platform can be deployed on any other platform. This is usually referred to as “write once, run anywhere” (WORA) and is accomplished through the use of the Java Virtual Machine (JVM). The JVM runs on a local machine, interprets the Java bytecode, and converts it into platform-specific machine code.
- **Dynamic and Distributed:** Java classes can be downloaded dynamically over the network when required. In addition, Java provides extensive support for client/server and distributed programming.
- **Multithreaded:** Java programs can contain multiple threads to carry out many tasks in parallel. The multithreading capability of Java is built-in and is under the control of the platform-dependent JVM.
- **Robust and Secure:** Java has built-in capabilities to prevent memory corruption. Java automatically manages the processes of memory allocation and array-bounds checking. It prohibits pointer arithmetic and restricts objects to named spaces in memory.

Using Java as a Language for Web Development



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Java with Enterprise Internet Computing

You can design Java programs as server-based components that form scalable Internet applications.

The currently accepted model for Java Internet computing divides the end-to-end application process into several logical tiers. To make use of this model, JavaSoft defined Java Platform, Enterprise Edition (Java EE), which has four logical tiers:

Client Tier

When Java executes on client machines, it is typically implemented as a browser-based application. But a thin client can be simply Web pages that are delivered from a server as HTML.

Web (Presentation) Tier

This is executed on a Web server. Code in this tier handles the application's presentation to the client. Common Java features for this function are servlets, JSP pages (JSPs), and JavaServer Faces (JSF). Servlets and JSPs can each generate dynamic HTML for display as Web pages to clients.

Using Java with Enterprise Internet Computing (continued)

Application (Business Logic) Tier

You can use Java on an application server to implement shareable, reusable business logic as application components. A common way to implement this is to use component models, such as Enterprise JavaBeans (EJB) and Common Object Request Broker Architecture (CORBA) objects. These two components are also to be considered during design time, when a distributed environment is required.

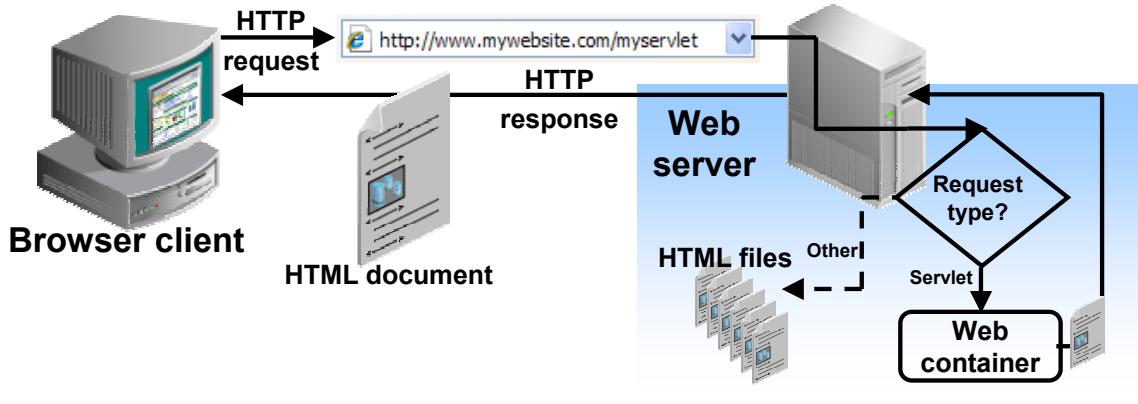
Data Tier

The data server not only stores data but also stores and executes Java code, particularly where this code is data intensive or enforces validation rules pertaining to the data. You can also use Business Components, from the Oracle Application Development Framework (ADF), to support your application's data access.

What Are Servlets?

Servlets:

- Are written in Java
- Extend the Web server
- Can be used to provide:
 - Back-end functionality
 - User interface



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

What Are Servlets?

The servlet technology is the foundation of Web application programming in Java. A servlet is a server-side Java program that gives Java-enabled servers additional functionality via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by Web servers. Servlets can perform back-end functionality or can dynamically construct user interfaces.

The life cycle of a servlet is controlled by the container in which the servlet is deployed. When a request is mapped to a servlet, the container performs the following steps:

- It loads the servlet class if there is no instance of it, creates an instance of the servlet class, and initializes the servlet instance by calling the `init` method.
- It invokes the `service` method, passing a request and response object.
- If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's `destroy` method.

When the browser request is for a servlet, the Web server forwards the request to the servlet container. The servlet dynamically constructs the HTML file, using any parameters that are sent in the URL request, and then the Web server returns the HTML file to the client browser.

JavaServer Pages (JSP)

JSP:

- Combines HTML and JSP tags
- Is saved as .jsp
- Is compiled to a servlet on the first request
- Is recompiled only when changed on the server
- Enables servlet functionality with less programming



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JavaServer Pages

Although servlets are powerful, they have some disadvantages. They can be cumbersome to program, and every change requires the intervention of the servlet programmer.

Sun understood this problem and developed JavaServer Pages (JSP) as the solution. JSP combines HTML with JSP tags in a file that has a .jsp extension. When the HTTP request contains this extension, the Web server runs the JSP in the servlet container, which converts it into a servlet. After it is compiled, further requests for the .jsp file run the servlet without the compilation process, unless the .jsp file is changed on the server.

Division of labor between a programmer and a page designer is much easier in JSP, and compilation takes place automatically when a JSP page is modified. Note that though JSP is an extension of the servlet technology, it does not make servlets obsolete. In real-world applications, servlets and JSP pages are used together.

What Are JavaBeans?

JavaBeans:

- Are Java classes that comply with certain standards
- Consist of properties, methods, and events
- Can be used by builder tools such as JDeveloper
- Can store state based on defined memory scope
- Can be used in JSPs



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

What Are JavaBeans?

The JavaBeans technology is a platform-neutral software component model for Java; it enables creating components that are easily assembled into sophisticated applications. JavaBeans are simple Java classes that conform to the following standards:

- Implement the `java.io.Serializable` interface so that applications can save the bean state
- Have a public, no argument constructor
- Have public accessors (getters and setters) for private variables (properties)

A JavaBean class consists of its properties, methods, and events.

Because of this conformance to standards, JavaBeans can be visually manipulated in a builder tool, such as JDeveloper. Tools can introspect the bean to analyze how it works, can customize the bean, and can allow users to interact with it based on events.

JavaBeans remain in memory for a defined period of time, based on their scope:

- `page`: Created and destroyed for every page view
- `request`: Exists for the life of the request
- `session`: Exists for the life of the user's session
- `application`: Exists as long as an application is loaded, for any user of the application

JSPs have tags to call JavaBeans and access their properties: `useBean`, `setProperty`, and `getProperty`.

What Is JavaServer Faces (JSF)?

- Is component based (not mark-up)
- Simplifies Web development
 - Insulates the developer from UI implementations
 - Declarative definition—more robust and extensible
- Is a server-side UI technology
- Is a standard Java EE Web UI framework with a huge base of vendor support; contributors include Sun, Oracle, IBM
- Supports automatic state and event handling
- Has a diverse client base (not just HTML)
- Is designed with tooling in mind
- Is display-agnostic
- Is applicable to a wide spectrum of programmer types



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JavaServer Faces

JavaServer Faces (JSF) is a component-based architecture, which means that instead of working with markup, the developer works with UI components, similar to Swing. Thus JSF simplifies Web development. The developer need not be concerned with UI implementations, and is able to use declarative definition, making it more robust and extensible.

JSF is a server-side UI technology, as opposed to a client-side UI technology such as ADF Swing. Requests are sent from the Web browser or other client to the server, where JSF translates the request into an event that the application logic can process. JSF also ensures that every widget that is defined on the server is correctly displayed on the client.

JSF technology establishes the standard for building server-side user interfaces, managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility. JSF hides the complexities of UI management, simplifying the process of building Web Uis.

JavaServer Faces (continued)

For example, consider what is required to develop a simple order form using traditional Web technologies: HTML for your presentation, state management over HTTP, JavaScript for input validation and behavior, and a servlet on the server side to process the order. In JSF, the developer is not concerned with these underlying technologies, but considers a field or a button on a form as a component in terms of a traditional UI. A button is as you think of it in the physical world: It is not an HTML or a JSP tag, but an object that when pressed has an associated behavior.

Developers can attach listeners to interpret user interactions with components into business logic, and the JSF infrastructure manages the rest for you. Thus, developing JSF applications is more about user interface design and assigning business logic to components rather than programming against low-level APIs.

JSF can automatically synchronize UI components with backing beans, which are Java objects that collect user input and respond to events. It can be used with a variety of clients.

The JSF specification was written with tool support facilities built in. The standard JSR-276 (Design Time Metadata for JSF Components, <http://www.jcp.org/en/jsr/detail?id=276>) aims to ensure that components from different vendors render correctly in IDEs that support JSF.

Because JSF is a UI component framework that works with tools and runs inside containers, it appeals to many types of programmers and has wide industry support.

There are many Web sites where you can get further information about JSF, such as <http://myfaces.apache.org> and <http://java.sun.com/javaee/javaserverfaces>.

JSF Key Concepts

JSF includes the following concepts:

- UI component, including renderers and render kits
- Controller: The navigation rules that govern page flow
- Managed beans and backing beans to encapsulate business logic, data, and properties of components (ADF BC applications define business logic mainly in the model layer, not the UI layer.)
- Life cycle: Creation, validation, data binding, and business service interactions of a page



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JSF Key Concepts

JSF encompasses several key concepts:

- **UI component:** Button, table of data, and menu tab
- **Controller:** Navigation rules that define how the user progresses through the application
- **Managed bean:** How business logic and data are specified; note that in an ADF BC application, most business logic and data are defined in the ADF BC model; and most component properties are defined in the ADF binding layer
- **Backing bean:** How the properties of a component are specified
- **EL:** Binding data and methods to UI components (You learn about EL in the lesson titled “Binding UI Components to Data.”)
- **Life cycle:** How the application behaves at run time

JSF Component Model

- Server-side UI components:
 - Provide functionality, definition, or behavior
 - Can be nested
- Renderers: Display UI components in a technology understood by the output device
- Render kits:
 - A render kit is a library of renderers.
 - The basic HTML Render Kit is part of the reference implementation.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JSF Component Model

As discussed previously, components are used to build JSF views, rather than mark-up documents. The coding paradigm for UI components is based on the JavaBeans model. You set property values or attributes to define the state and attach listeners to respond to value changes or other events, thereby defining behavior.

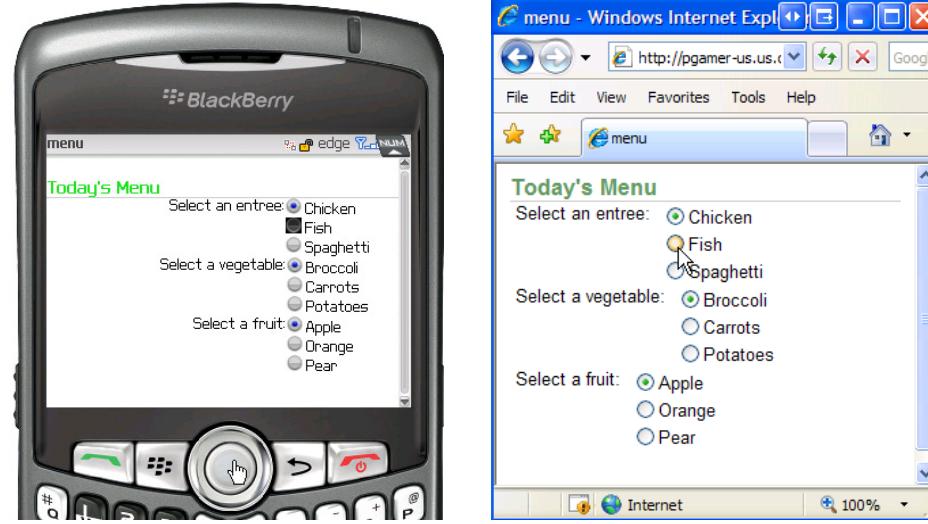
Some components can be nested. In most cases, the purpose of nesting components is to provide a layout for other components, such as a table, within a view. UI components are standardized within JSF, so you can mix and match components from different vendors. However, if you do so, you may not be able to achieve a standard look and feel or be able to support internationalization.

JavaServer Faces includes:

- **UI components:** Server-side objects that are independent of the target client and render back to the client using external renderer classes. UI components consist of attributes, behaviors, and renderers.
- **Renderers:** Classes that reside on the server and are responsible for displaying the UI component in a graphical representation that is understood by the client
- **Render kits:** Renderer classes packaged into libraries that enable you to develop a JSF application for different client devices (such as browsers or mobile devices) without altering application code. A render kit for an HTML client is part of standard JSF.

JSF Multiple Renderers

Option buttons: When an item has focus, its highlighting looks different depending on the client.



Render kits enable the same component to be rendered differently depending on the client.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JSF Multiple Renderers

The slide shows how Web applications can be rendered through different clients. Standard JSF functionality renders the same application on different clients without modifying the application. An example of a part of the JSF component is the option button. It has a set of behaviors (on or off, only one in a group selected, and so on). A renderer would define how the option button appears, for example, in HTML. As shown in the slide, when the cursor is positioned over an option in a browser, the highlighting appears differently from the highlighting on a mobile device, but the behavior and function are the same on each. The render kit then specifies how all components render for a set of clients.

Traditional Navigation

In traditional navigation, a button or hyperlink is hard coded to navigate to a specific page:

```
<body>
    <form action="myNewPage.html">
        <button type="button"/>
    </form>
</body>
```

**Hard-coded
button
navigation**

**Hard-coded
hyperlink
navigation**

```
<body>
    <a href="http://myNewPage.html">
        Go To My New Page
    </a>
</body>
```

ORACLE

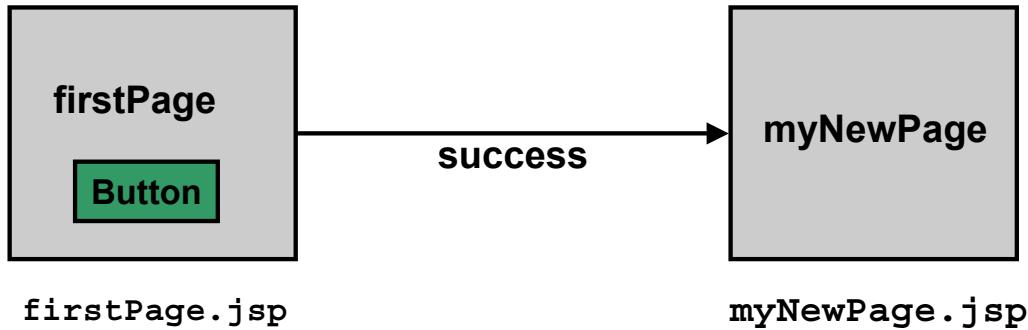
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Traditional Navigation

This type of navigation is also referred to as the Model 1 type. In both cases, if the physical location of the page is changed, you have to modify all the links and forms using it. How can you avoid this maintenance nightmare?

JSF controls the pages using a separate XML file, `faces-config.xml`, that contains all the navigation mechanisms.

Defining Navigation by Using the JSF Controller



**JSF Navigation Rules
defined in
faces-config.xml**

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Defining Navigation by Using the JSF Controller

In JavaServer Faces, navigation between application pages is not hard coded in the file, but is defined by a set of rules that are centrally defined in the `faces-config.xml` file. Navigation rules determine the next page to display when a user clicks a navigation component, such as a button or a hyperlink.

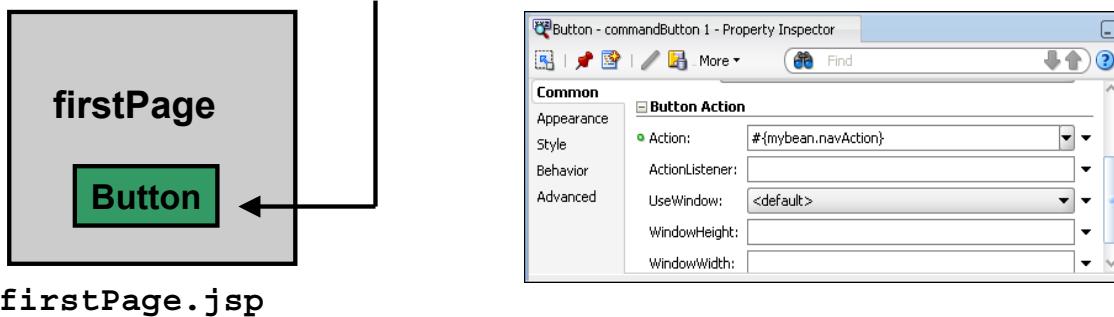
A navigation rule defines the navigation from one page to one or more other pages. Each navigation rule can have one or more cases, which define where a user can go from that page. For example, if a page has links to several other pages in the application, you can create a single navigation rule for that page and one navigation case for each link to the different pages. The rule itself can define the navigation from:

- A specific JSF page
- All pages whose paths match a specified pattern, such as all the pages in one directory, which is called a pattern-based rule
- All pages in a Java EE Web application, which is called a global navigation rule

JSF Navigation: Example

A command UI component can be bound to an action:

```
<h:commandButton action="#{mybean.navAction}" />
```



`firstPage.jsp`

**mybean: The logical name of a managed bean
(MyBean.java) with an action method, navAction(),
which returns a string**

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JSF Navigation: Example

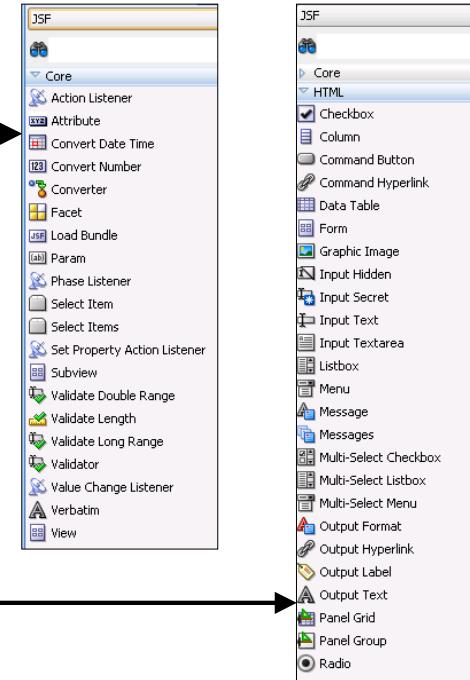
Managed beans, explained later in this lesson, are Java classes that are managed in the faces-config.xml configuration file. In this example, the code from the managed bean class determines the value that is returned and then used by the navigation case.

```
public class MyBean {
    public MyBean() { }
    public String navAction()
    {
        if (<check some condition>) {
            return "success";
        }
        else
        {
            return "failure";
        }
    }
}
```

Using JSF Components

JSF components include:

- Core
- HTML



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using JSF Components

The standard JSF implementation includes components in the following categories:

- **JSF Core Tag Library:** Used with other components to perform core actions that do not depend on a particular render kit. Tags start with `f:`. The library includes such elements as listeners, converters, validators, and subcomponents such as list items, and it also includes `f:view`, a required part of a JSF page
- **HTML Tag Library:** Used to represent form controls and other basic HTML elements that control display data or accept user input. Tags start with `h:`. The library includes such elements as buttons, links, messages, tables, check boxes, graphics, input and output text, menus, and option buttons. The `h:form` tag is required on a JSF page to contain all JSF tags.

Using JSF Managed Beans

Managed beans are optional and can be used to:

- Hold presentation and controller logic
- Store state
- Execute Java routines when, for example, a user clicks a button
- Define handler for Event Listeners
- Add any code needed by UI



Backing beans:

- Are a specialized type of managed beans that contain accessors for UI components
- Have a one-to-one relationship with a page

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using JSF Managed Beans

JSF components are a type of JavaBean, and they are also designed to work with another type of JavaBean called a managed bean. Often managed beans handle events or some manipulation of data that is best handled at the front end rather than placing the logic in a business component. However, all application data and processing should be handled by logic in the business layer of the application.

Some typical uses for managed beans are:

- To hold values that can be used by multiple pages
- To execute a method in response to user interaction, such as a button click
- To execute a method when a UI event, such as a value change, takes place

Backing beans are a specialized type of managed bean that exposes some or all of the UI components on the page by defining accessors for them. When you create a JSF page, the Page Implementation section of the Create JSF Page dialog box enables you to choose whether to expose the UI components, which you would need to do only if the page requires special code to manipulate its components programmatically. For example, you may set a condition to determine whether to render the component. If you choose to expose components in a managed bean, JDeveloper automatically adds stubs for accessors as you add components to the page. Backing beans are directly related to pages, whereas other types of managed beans can be made available to multiple pages.

Overview of JSF Page Life Cycle

- A JSF page is represented by a tree of UI components, called a view.
- Request processing is managed by `FacesServlet`, which creates the `FacesContext` object.
- When a client makes a request for the page, the life cycle starts.
- During the life cycle, JSF implementation builds the view while considering the state saved from the previous postback.
- When the client performs a postback of the page, JSF implementation must perform life-cycle steps:
 - Conversion
 - Validation



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

JSF Life Cycle

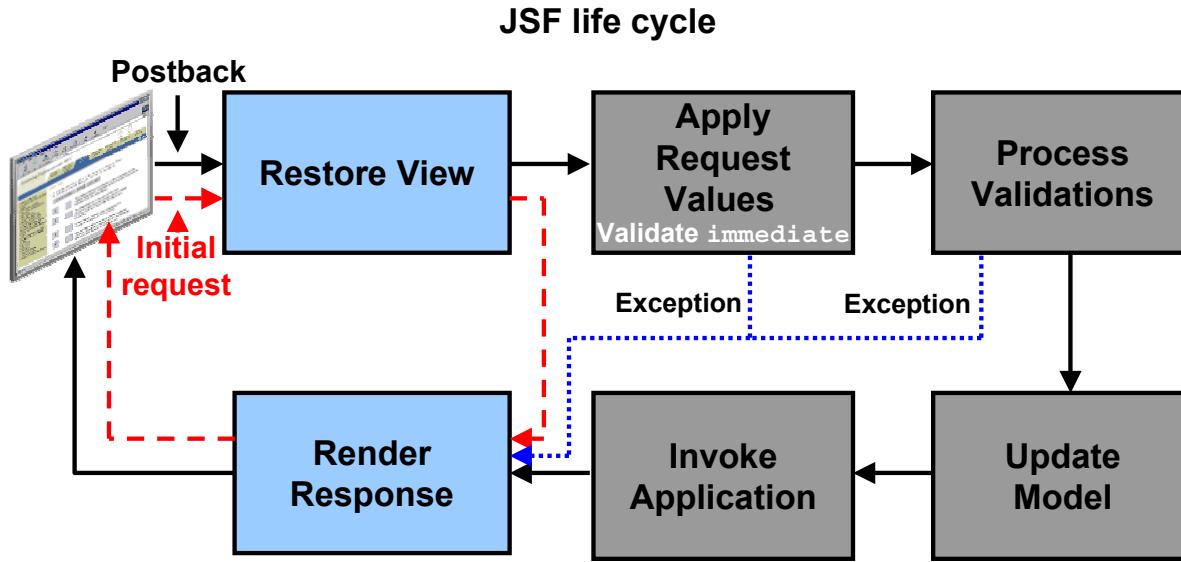
The life cycle of a JavaServer Faces page is similar to that of a JSP: The client makes an HTTP request for the page, and the server responds with the page translated to HTML. However, because of the extra features that the JavaServer Faces technology offers, the life cycle provides some additional services to process a page.

A JavaServer Faces page is represented by a tree of UI components called a view. This tree is a run-time representation of a JSF page. Each UI component tag on a page corresponds to a UI component instance in the tree.

The `FacesServlet` object manages the request processing life cycle in JSF applications. `FacesServlet` creates an object called `FacesContext`, which contains the information necessary for request processing, and invokes an object that executes the life cycle.

When a client makes a request for the page, the life cycle starts. During the life cycle, the JSF implementation must build the view while considering the state saved from the previous postback. When the client performs a postback of the page, the JavaServer Faces implementation performs several tasks, such as validating the data input of components in the view and converting input data to types specified on the server side. The JSF implementation performs all these tasks as a series of steps in the life cycle.

Formal Phases of the JSF Life Cycle



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Formal Phases of the JSF Life Cycle

The life cycle handles both initial requests and postbacks.

When a user makes an initial request for a page, the life cycle executes only the **Restore View** and **Render Response** phases because there is no user input or actions to process. The life cycle builds the UI tree from the view, and then renders the view, returning a response to the user and saving the view tree for future requests in the `FacesContext` object. All the component tags, event handlers, converters, and validators on the page have access to the `FacesContext` instance.

When a user executes a postback, a form is submitted that is contained on a page that was previously loaded into the browser as a result of executing an initial request. When the life cycle handles a postback, it restores the view based on the data saved on the server, and then executes the following additional phases:

- Apply Request Values:** Values provided in components that hold a value (such as input fields) have their values applied to their counterparts in the view tree. All events such as `ValueChangeEvent`s (VCE) or `ActionEvent`s (AE) are queued. A VCE means that a value has changed for a specific UI component. An AE means that a button (or any UI component that is a source of an action) was clicked. If a component has its `immediate` attribute set to `true`, then validation, conversion, and events associated with the component are processed during this phase.

Formal Phases of the JSF Life Cycle (continued)

- **Process Validation:** Conversion and validation logic is executed for each component. This means both built-in validation/data conversion and custom validation/conversion are added onto the components. If a validation error is reported, an exception is thrown. The life cycle halts and the response is rendered with validation error messages. At the end of this phase, new component values are set, any validation or conversion error messages and events are queued on FacesContext, and any value change events are delivered.
- **Update Model:** The component's validated local values are moved to the model and the local copies are discarded. If you are using a backing bean for a JSF page to manage your UI components, any managed bean properties that are value-bound to the UI component using the value attribute are updated with the value of the component.
- **Invoke Application:** Any action bindings for command components or events are invoked. Navigation is handled here depending on the outcome of action method (if any).

Setting the `immediate` Attribute

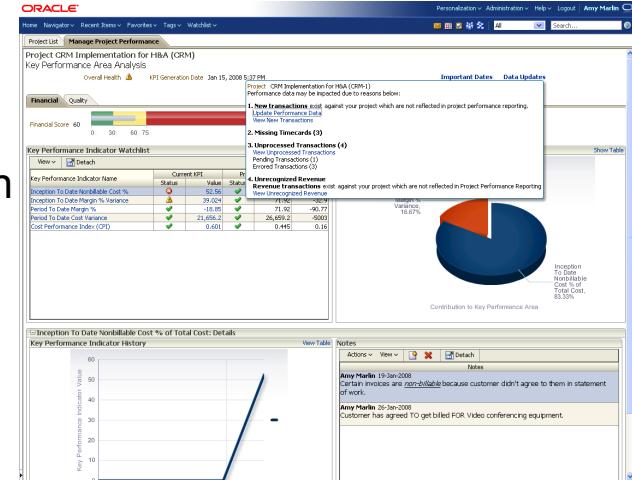
You can use the `immediate` attribute to allow components on the page to be validated in the Apply Request Values phase of the life cycle as opposed to the Process Validations phase:

- Using `immediate` on an input component means that that component's value is validated before any input components that do not have the `immediate` attribute set to `true`. Therefore, if a validation error occurs on an `immediate` input component, the life cycle moves from the Apply Request Values phase (where the `immediate` attribute is evaluated) to the render phase, and validation does not run on any “nonimmediate” input components. Additionally, if the new value of an immediate input component is different from the existing value, then a `ValueChangeEvent` is raised. However, instead of the event being processed during the Process Validations phase, the event is processed at the end of the Apply Request Values phase. Therefore, any `ValueChangeListener` associated with the immediate input component executes before any command component's `ActionListener` (assuming the command component occurs later on the page). You learn about event listeners in the lesson titled “Responding to Application Events.”
- For command components, if set to `immediate` and the component has an action that returns a value for navigation, the life cycle proceeds directly to the Render Response phase. The validation and model update phases are skipped. A Cancel button is an example of when this would be used.

Key Characteristics of Rich User Interfaces

Rich Web UIs include the following characteristics:

- Increased responsiveness over traditional Web applications
- Asynchronous communication with the server
- Behaviors not available in HTML alone



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Key Characteristics of Rich User Interfaces

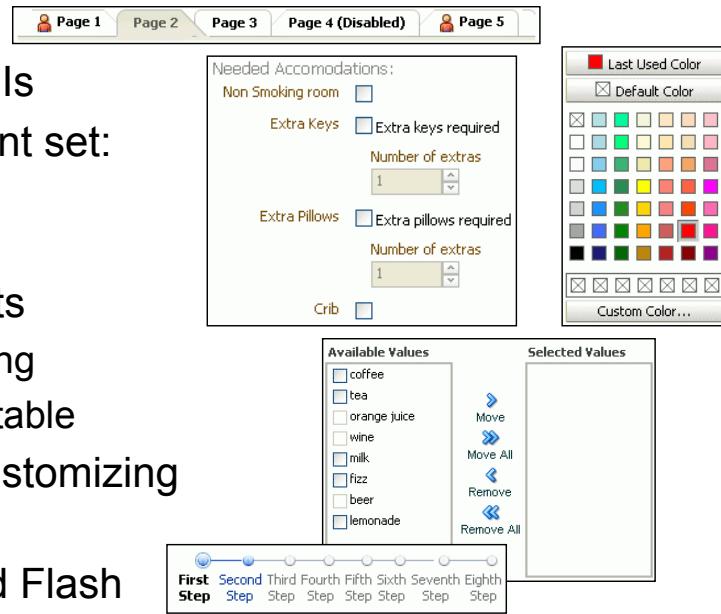
Increased Responsiveness: Traditional Web applications interact with data on the server, resulting in elongated round-trips even for simple activities, such as sorting and filtering. Rich UIs typically cache data sets on the client side, making activities such as sorting and filtering much more responsive because no round-trip to the server is necessary.

Asynchronous communication with the server: Traditional Web applications typically render pages in a top-down fashion, so that the length of time that a page takes to load will be no less than the longest operation on the page. Utilizing asynchronous communication with the server eliminates the traditional top-down rendering of a page and can, therefore, give the perception of improved performance. Long operations can be performed asynchronously, with a callback mechanism triggering the rendering of a particular section of the page when the operation is complete. This leads to a natural tendency to create fragments of markup (something both portlets and JSF make very easy) to take advantage of parallelization.

Behaviors not available in HTML alone: Rich UIs clearly differ from traditional Web application in their look and feel and in how the end user interacts with and navigates the application; that look and feel cannot be produced through HTML alone. For example, HTML alone cannot open new windows or change color on mouse over. However, such dynamic features, coupled with asynchronous behavior and parallelization, create a highly decorated window into an application.

Adding to JSF with ADF Faces

- Built on top of JSF APIs
- Much larger component set: over 100 types
- More advanced and interesting components
 - Partial-page rendering
 - Scrollable, sortable table
- Rich feature set for customizing applications
- Uses AJAX, SVG, and Flash
- ADF model support
- Runs on any JSF-compliant implementation



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Adding to JSF with ADF Faces

Oracle ADF Faces is a rich set of user interface components based on the JavaServer Faces JSR. Oracle ADF Faces uses the flexible JSF rendering architecture to combine the power of AJAX and JSF to provide over 100 components with built-in functionality—such as data tables, hierarchical tables, and color and date pickers—that can be customized and reused in your application.

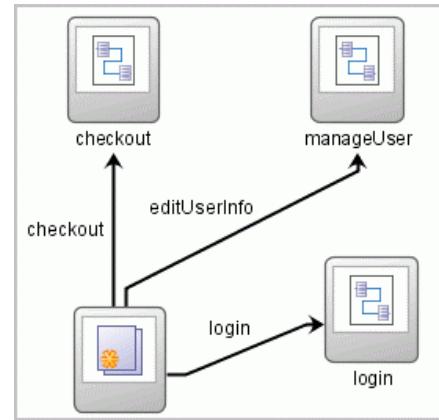
Each component also offers complete customization, skinning, and support for internationalization and accessibility. ADF Faces also provides a rich set of Flash- and SVG-enabled components that are capable of rendering dynamic charts, graphs, gauges, and other graphics that provide real-time updates. This built-in support for asynchronous JavaScript and XML (AJAX), scalable vector graphics (SVG), and Flash (Adobe's multimedia playback) enables developers to build rich UIs without an extensive knowledge of the underlying technologies.

ADF Faces also simplifies the code necessary to connect application logic in existing business services to UI components. Its components are JSR-227 compliant, enabling developers to easily bind services to their UIs at design time.

ADF Faces components can be used in any IDE that supports JSF.

Using the ADF Controller

- Is built on top of the JSF navigation model
- Handles page flow
 - Promotes page reuse through abstraction
 - Increases flexibility and manageability
- Is the place for code execution, such as programmatic interaction with the model and business service
- Supports application tasks
 - Input validation
 - State management
 - Security
 - Declarative transaction control
- Provides declarative Back button control



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the ADF Controller

ADF Controller is an enhanced navigation and state management model on top of JavaServer Faces. It enables you to describe page flows declaratively. Page flows can be encapsulated in a task flow, which promotes reuse. Task flows represent activities and the control flow rules that define the transitions between the activities. JSF navigation alone does not provide this task flow concept.

In addition to navigation, the ADF Controller provides the ability to call methods, either on managed beans or on Java objects that are not managed beans. It provides support for application tasks such as validation, state management, security, and declarative transaction control.

The ADF Controller also enables you to control behavior when navigation is the result of the user clicking the Back button.

Subsequent lessons elaborate on the functionality of the ADF Controller.

ADF Life Cycle Phases

JSF Phases	ADF Phases	Events
Restore View	JSF Restore View	before(JSF Restore View) after(JSF Restore View)
	Init Context	before(Init Context) after(Init Context)
	Prepare Model	before, after
Apply Request Values	JSF Apply Request Values	before, after
Process Validations	JSF Process Validations	before, after
Update Model Values	JSF Update Model Values	before, after
	Validate Model Updates	before, after
Invoke Application	JSF Invoke Application	before, after
	Metadata Commit	before, after
Render Response	JSF Render Response	before, after
	<i>Only after navigation</i> Init Context	before, after
	<i>Only after navigation</i> Prepare Model	before, after
	Prepare Render	before, after



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ADF Life Cycle Phases

Because ADF is built on top of JSF, its life cycle includes all the JSF phases. In addition, the following phases have been added:

- **Init Context and Prepare Model:** Executed after JSF Restore View, and also after JSF Render Response when navigation has occurred
- **Validate Model Updates:** Occurs after JSF Update Model Values
- **Metadata Commit:** Occurs after JSF Invoke Application
- **Prepare Render:** Occurs after JSF Render Response and, if applicable, the Init Context and Prepare Model phases that apply when navigation has occurred

You can define before and after events for all these phases, such as before (Validate Model Updates) or after (Prepare Render). ADF phase events are always executed after the event of the JSF phase on which it depends.

You learn more about events and the role of the ADF life cycle in the lesson titled “Responding to Application Events.”

Summary

In this lesson, you should have learned how to:

- Describe the use of Web browsers and HTML
- Explain how Java has come into widespread use as a language for developing Web applications
- Describe the function of Servlets and JSPs
- Define JavaServer Faces
- Explain the JSF component architecture
- List some JSF component types included in the standard implementation
- Describe the purpose of backing beans
- Explain the use of managed beans
- Describe the JSF life cycle
- Explain how ADF Faces augments the JSF life cycle



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

11

Binding UI Components to Data

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Create a JSF page
- Add ADF Faces UI components to a page
- Include databound components on a page
- Create and edit data bindings



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson describes how to use ADF Faces to create a simple page that includes databound components. ADF Model and data binding are described in detail.

Creating a JSF Page

The screenshot shows the Oracle JDeveloper interface. On the left, the 'New Gallery' window is open under 'Current Project Technologies'. It lists various item types, with 'JSF Page' selected. A tooltip for 'JSF Page' explains it launches the 'Create JSF Page' dialog to create a skeleton JavaServer Faces (.jsp or .jspx) file. On the right, a task flow diagram titled 'adf-config.xml' is shown. A context menu is open over a view icon, with 'Create Page...' highlighted. Below the interface, a code snippet of XML-based JSP code is displayed, showing the structure of a JSF page definition.

```

<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
           xmlns:h="http://java.sun.com/jsf/html"
           xmlns:f="http://java.sun.com/jsf/core"
           xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
    <jsp:directive.page contentType="text/html;charset=UTF8"/>
    <f:view>
        <af:document>
            <af:form/>
        </af:document>
    </f:view>
</jsp:root>

```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating a JSF Page

To create a JSF JSP, you can invoke the New Gallery as shown in the slide. If you have generated task flows during the planning stages, instead of using the New Gallery, you can double-click a view icon in the task flow diagram to create the actual JSP file, or right-click the icon and select Create Page.

Oracle recommends that when creating an ADF application, you create an XML-based JSP document, which uses the extension `.jspx`, rather than creating a `.jsp` file. Using an XML-based document provides the following benefits:

- Treats your page as a well-formed tree of UI component tags
- Discourages you from mixing Java code and component tags
- Enables you to easily parse the page to create documentation or audit reports
- Enables the page to be used as metadata

All JSF pages that use ADF Faces components must have `af:document` enclosed within `f:view`, as shown in the following code snippet (elements beginning with “`af:`” are ADF Faces elements, whereas those that begin with “`f:`” are JSF elements):

```

<f:view>
    <af:document />
</f:view>

```

Creating a JSF Page (continued)

By default, when you create a JSF page in a project that uses ADF Faces technology, JDeveloper automatically inserts the `af:document` tags for you. All other components that make up the page then go in between `<af:document>` and `</af:document>`. The `af:document` component renders nothing itself, but the contents within it are rendered, where appropriate.

At run time, the `af:document` component creates the root elements for the client page. For example in HTML output, the standard root elements of an HTML page are generated: `<html>`, `<head>`, and `<body>`.

Typically you would use `af:form` within `af:document` to contain your page contents, so JDeveloper creates that tag for you also, as shown in the example in the slide of the XML code for a page when it is first created.

After your page files are created, you can add databound UI components.

Adding UI Components to the Page

You can create components on a page by:

- Dragging a component from the Component Palette
- Using the context menu in editor or Structure window
- Dragging a data element from the Data Controls panel



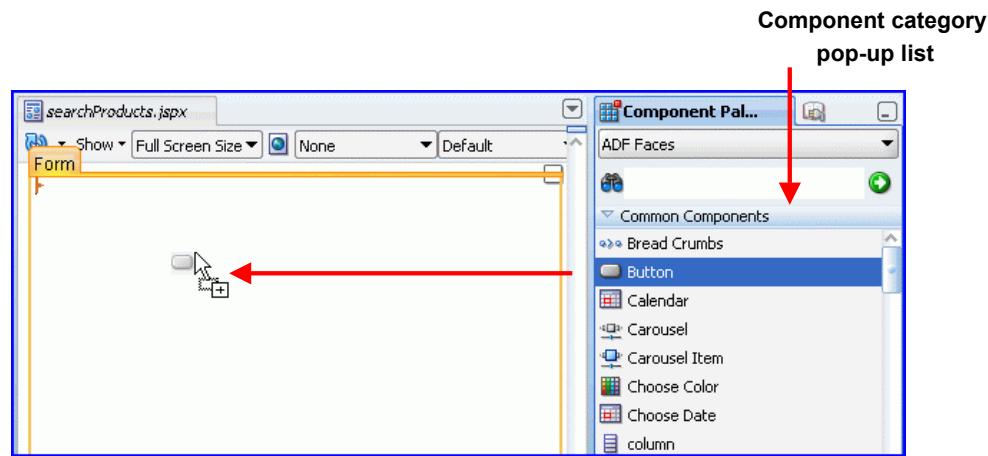
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Adding UI Components to the Page

You can add databound components to a page by using either the Component Palette or the Data Controls panel. You also can right-click in the Structure window or in the Design view in the editor and select from the context menu to insert a component. You can also add components through the Code Editor, using XML. Click the Source tab to display the Code Editor.

Using the Component Palette

Drag the component from the Component Palette:



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Component Palette

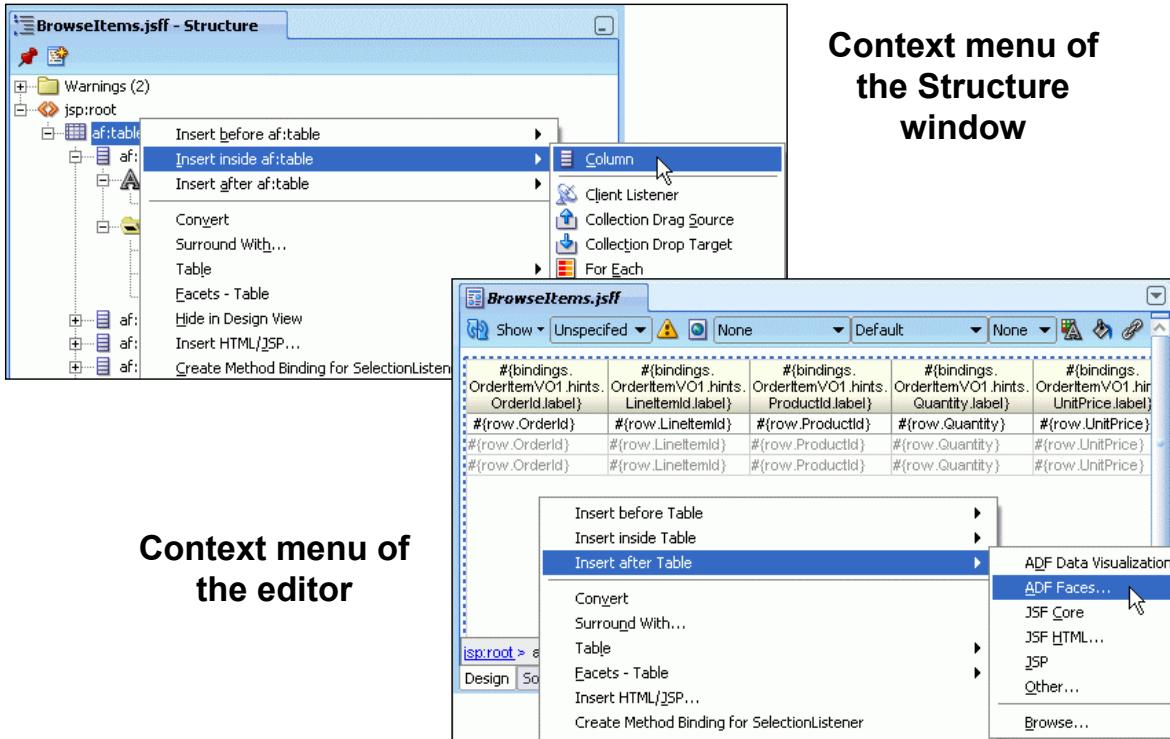
JDeveloper provides a Component Palette that enables you to easily add components by dragging them to the page in the visual editor or to the Structure panel. You can select a component category from the pop-up list at the top. By default, the following categories are available: ADF Data Visualization (graph components), ADF Faces, CSS (stylesheets), HTML, JSF, JSP, and JSP Standard Tag Library (JSTL).

For ADF Faces, the following types of components are provided:

- **Common components:** The building blocks of a JSF page with properties and behaviors
- **Layout components:** Containers to lay out components on a page and define the resize behavior
- **Operations:** Specification of client-side behavior or user interaction

When you create components by dragging them from the Component Palette, you can use Expression Language (EL) to bind them to data. You learn more about data binding and EL later in this lesson.

Using the Context Menu



Context menu of the Structure window

Context menu of the editor

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Context Menu

In both the Structure window and in the design view of the editor, you can right-click and select from the context menu to:

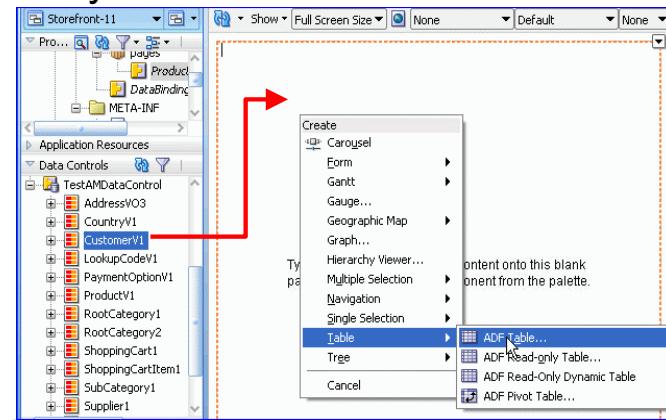
- Insert another UI component inside, before, or after the current one
- Surround the selected component with another

If inserting inside, you are able to choose from components that are valid for that location or browse for others. If inserting before or after, or surrounding with, you are presented with a list of component categories from which to choose. You can browse for others, or if you select a category, you can then choose from all components of that category.

Using the Data Controls Panel

The Data Controls panel:

- Is a visual representation of your business service that contains:
 - Methods
 - Parameters and results
 - Attributes
 - Collections
 - Built-in operations
- Provides automatic data binding for any business service
 - For example, there automatically is a data control for every ADF BC application module.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using the Data Controls Panel

The Data Controls panel shows all the data controls that have been created for the application's business services and exposes all the data objects, data collections, methods, and operations that are available for binding to UI components. A different icon is used for each type of data control object. Each root node in the Data Controls panel represents a specific data control. Under each data control is a hierarchical list of objects, collections, methods, and operations. How this hierarchy appears on the Data Controls panel depends on the type of business service represented by the data control and how it was defined.

ADF BC automatically creates a data control for each application module in an application.

The Data Controls panel enables you to easily create databound components on a page. When you drag a data element to a page, you are given a choice of the type of component to use to contain the data element, based on whatever is appropriate for that particular element.

Dragging a collection (such as a view object [VO]) to the page, gives you the choice of creating the component as an ADF form, table, Gantt chart, graph, gauge, single selection, tree, navigation component, or geographic map. When you select a category, you are presented with additional choices. After you select a component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control that you selected.

Oracle ADF Data Controls

- Provide an abstraction of the business service's data model
- Give the ADF binding layer access to the service layer
- Allow you to bind page components to data without writing any code
- Are available for the following (in addition to ADF Business Components):
 - JavaBeans, including TopLink-enabled objects
 - EJB session beans
 - Web services



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Oracle ADF Data Controls

Oracle ADF data controls provide an abstraction of the business service's data model. Examples of business services include Web services, EJB session beans, or any Java class being used as an interface to some functionality. Therefore, each of these technologies is able to work with data controls, in addition to ADF Business Components.

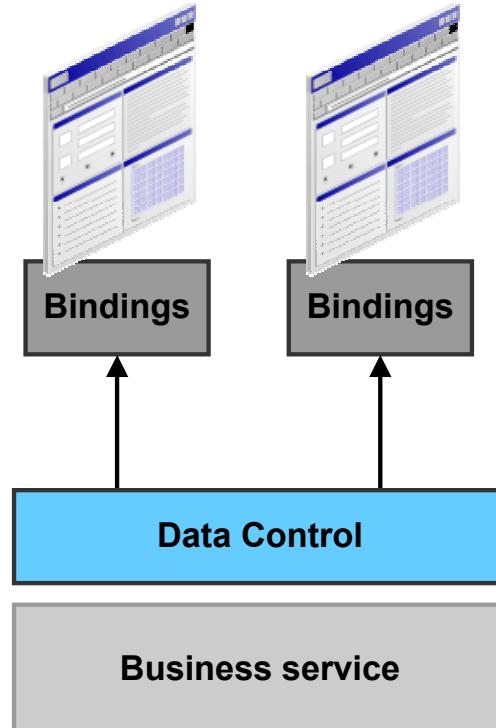
To create a data control for a Java class, for example:

1. Select the Java class in the Application Navigator.
2. Select Create Data Control from the context menu.

After you have created the data control, you bind it to the page by dragging it in the same way as you saw for ADF Business Components in the previous slide.

Describing the ADF Model Layer

- Data controls describe the public interface of a business service.
- Bindings connect UI components to data or actions.
- Data controls and bindings are defined by using XML metadata.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Model Layer Components

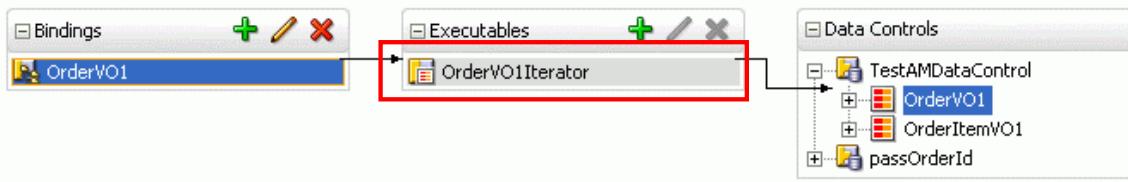
The Oracle ADF Model layer implements the two concepts in the JSR-227 specification that enable decoupling the user interface technology from the business service implementation:

- **Data controls:** The ADF data controls provide an abstraction of an application's business services, giving the ADF binding layer access to the service data. Data controls define the data model returned by the business service. You can bind UI components to data controls to populate a page with data from your data model at run time.
- **Declarative data binding:** The UI components created by the Data Controls panel use declarative data binding, which means that the data binding expressions are automatically configured, and that in most cases, you do not have to write any additional code. When you use the Data Controls panel to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

Types of Data Bindings

- Iterator binding: Keeps track of the current row in a data collection

- Iterator
- Method iterator
- Accessor iterator
- Variable iterator



- Value binding: Connects UI components to attributes in a data collection—for example, attribute binding, tree binding, list binding, and table binding
- Action binding: Invokes a method or operation

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Types of Bindings

Oracle ADF provides several types of binding objects to support the attributes and operations exposed by the Oracle ADF data controls for a particular business object:

- **Iterator binding:** One per accessor attribute that your page or panel displays; iterates over the business objects of the data collection and maintains the row currency and state. The slide shows an iterator binding as depicted in a page definition file (more on this shortly).
- **Value binding:** One for each databound UI component. It provides access to data.
- **Action binding:** Specifically defined for command components. It provides access to operations or methods defined by the business object.

Iterators are usually created for you so that you do not have to create them explicitly. One exception is that when creating a list binding, you often have to create a new iterator for the displayed items. This is done from the list binding editor, as described later in this lesson. There are four types of iterator bindings:

- **Iterator:** Iterates over a collection. When you drop a VO from the Data Controls panel onto a page, an iterator is created automatically, as depicted in the slide.
- **Method iterator:** Iterates over the results returned by a method

Note: The screenshot in the slide is from a page definition file, which you learn about shortly.

Types of Bindings (continued)

- **Accessor iterator:** In a master-detail relationship, iterates over detail objects returned by accessors; created automatically when an accessor return is dropped on the page from the Data Controls panel. Accessor iterators are always related to a master iterator, which is the method iterator for the parent object. The accessor iterator returns the detail objects related to the current object in the master (or method) iterator.
- **Variable iterator:** Iterates over local variables and method parameters created within the binding container. These variables and parameters are local to the binding container and exist only while the binding container object exists. When you use a Data Control method or operation that requires a parameter that is to be collected from the page, JDeveloper automatically defines a variable for the parameter in the page definition file. Attribute bindings can reference the binding container variables.

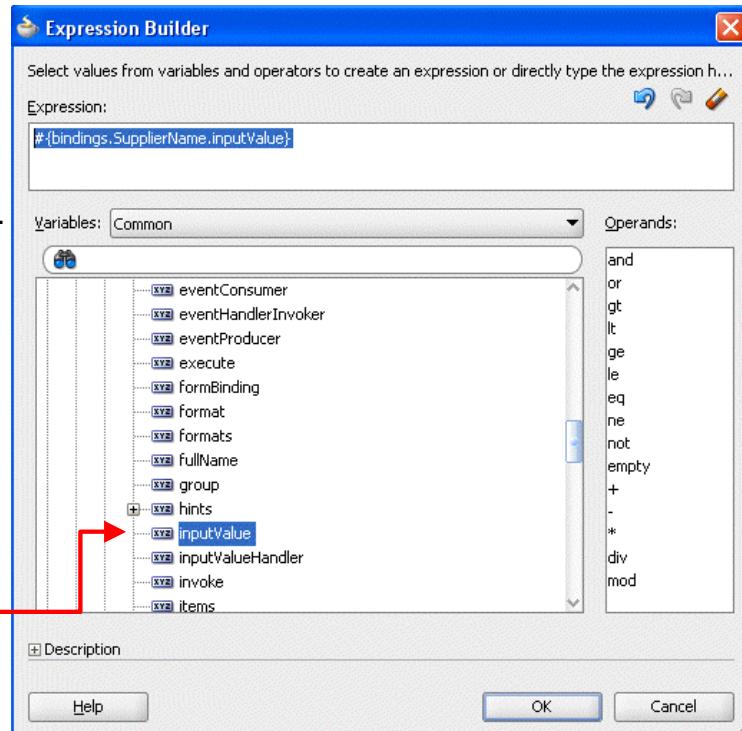
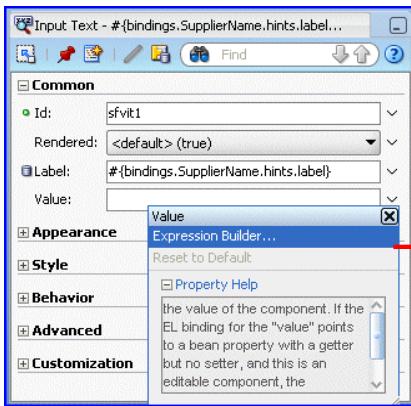
Value bindings are of different types depending on data, such as in the following examples:

- A page has a drop-down list of department names. The combo box uses a list binding to display department names and update department numbers.
- A form shows the employee's last name as a text field. The text field uses an attribute binding to bind to the LastName attribute.

Action bindings, which are for a command component such as a button, provide access to methods or operations defined by the business object. For example, a Next button on a form navigates to the next record. The Next button uses an action binding to bind to the Next operation.

Using Expression Language (EL)

Use Expression Builder to declaratively create EL expressions.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Using Expression Language

You can use a simple Expression Language (EL) to work with information. EL is part of JSP Standard Tag Library (JSTL), and it defines a standard syntax for accessing dynamic values without using Java scriptlets in JSPs. The standard syntax is `#{object.attribute}` for a JSF expression or `${expression}` for a standard JSTL expression. Binding expressions can use either a \$ or #prefix, but EL expressions in JSF pages can use only the # prefix.

At run time, a generic expression evaluator returns the value of expressions, automating access to the individual objects and their properties without requiring code. You use EL expressions throughout an ADF Faces application to bind attributes to object values determined at run time.

The value of certain UI components (such as an input text component) is determined at run time by the `value` attribute. Though a component can have static text as its value, typically the `value` attribute contains an EL expression that the run-time infrastructure evaluates to determine what data to display. Because any attribute of a component (and not just the `value` attribute) can be assigned a value by using an EL expression, it is easy to build dynamic, data-driven user interfaces.

You can create EL expressions declaratively by using the JDeveloper Expression Builder. You can access the builder from the Property Inspector by selecting Expression Builder from the drop-down list of an attribute.

Expression Language and Bindings

- Data binding expressions are written using EL.
- They are evaluated at run time to determine what data to display.
- ADF EL expressions typically have the form:
`#{bindingVariable.BindingObject.propertyName}`
 - Example of an `inputText` component on a JSF page:

```
<af:inputText  
    value="#{bindings.SupplierName.inputValue}"  
    label="#{bindings.SupplierName.label}"  
    required="#{bindings.SupplierName.mandatory}">
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Expression Language and Bindings

When you use the Data Controls panel to create a component, the ADF data binding expressions are created for you. The expressions are added to every component attribute that displays data from, or references properties of, a binding object.

Each expression references the appropriate binding objects defined in the page definition file, the file that contains data bindings for a page. You can edit these binding expressions or create your own, as long as you adhere to the basic ADF binding expression syntax. ADF data binding expressions can be added to any component attribute that you want to populate with data from a binding object.

In JSF pages, a typical ADF data binding EL expression uses the following syntax to reference any of the different types of binding objects in the binding container:

`#{bindings.BindingObject.propertyName}`

where:

- `bindings` is a variable that identifies that the binding object being referenced by the expression is located in the binding container of the current page. All ADF data binding EL expressions must start with the `bindings` variable.

Expression Language and Bindings (continued)

- *BindingObject* is the ID (or, in the case of attributes, the name) of the binding object as it is defined in the page definition file. The binding objectID or name is unique to that page definition file. An EL expression can reference any binding object in the page definition file, including parameters, executables, or value bindings.
- *propertyName* is a variable that determines the default display characteristics of each databound UI component and sets properties for the binding object at run time. There are different binding properties for each type of binding object.

The JDeveloper Expression Builder is a dialog box that helps you build EL expressions by providing lists of binding objects defined in the page definition files, as well as other valid objects to which a UI component may be bound. It is particularly useful when creating or editing ADF databound expressions because it provides a hierarchical list of ADF binding objects and their most commonly used properties. For information about these objects and properties, see the *Oracle Fusion Middleware Fusion Developer's Guide for Application Development Framework 11g*, which is available on OTN.

The example in the slide shows an ADF Faces `inputText` component (a text field with label) that displays the supplier name. The attributes of the text field are all set using EL expressions; the dot-separated expressions in the example all start with `bindings`, the binding variable that represents the binding context of the current page. This is explained further later in this lesson.

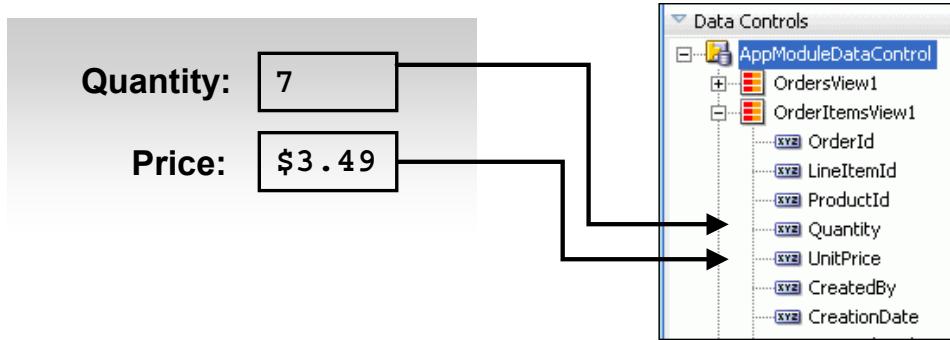
The attributes are evaluated as follows:

- `value="#{bindings.SupplierName.inputValue}"` : The value of the `inputText` component is set to the value of the `SupplierName` attribute.
- `label="#{bindings.SupplierName.label}"` : The label is set to the `label` property of the `SupplierName` attribute. If you have assigned a custom label to `SupplierName`, then the `label` property is your custom label; otherwise the `label` property is the same as the attribute name.
- `required="#{bindings.SupplierName.mandatory}"` : `required` is a boolean attribute that determines whether a value must be entered in the field. In this case, `required` evaluates to the `mandatory` property of the `SupplierName` attribute. If `SupplierName` is a mandatory attribute, the `inputText` requires a value and is displayed with a required indicator (*) next to the field.

Creating and Editing Data Bindings

Data Bindings:

- Are created automatically when you drag items from the Data Controls panel to a page or panel
- Can also be created and edited in the editor, the Property Palette, or the Structure window



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Creating and Editing Data Bindings

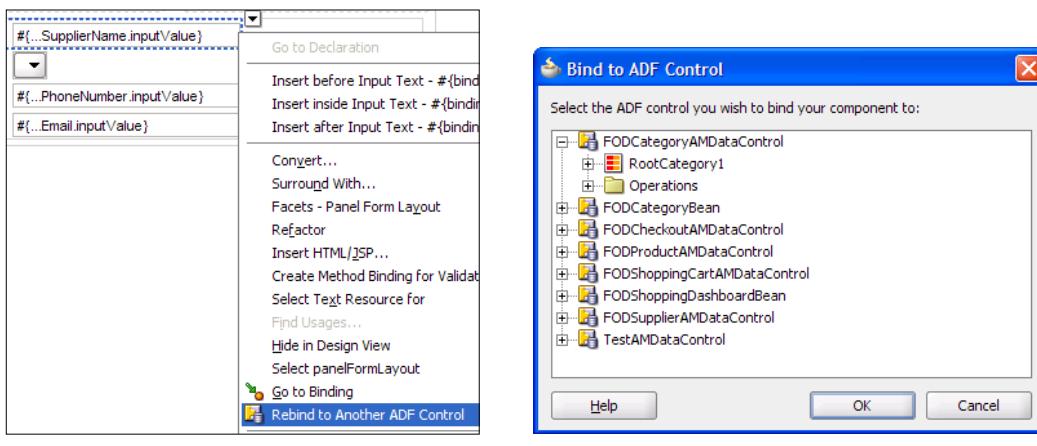
You can create or edit a data binding in one of the following ways:

- When you drag a component from the Data Controls panel to a page or panel, a binding is created automatically.
- You can select the component in the editor and edit the binding in the Property Inspector. This enables you to use Expression Builder to create an EL expression.
- With the component selected in the editor, you can right-click the component in the editor or the Structure window and select “Rebind to Another ADF Control.” If there is no existing binding, you can select “Bind to ADF Control.” This is explained further in the next slide.
- You can view and edit bindings contained in the page definition file by clicking the Bindings tab at the bottom of the editor, and then selecting a binding and clicking Edit (the Pencil icon.)
- You can also open the page definition file and create or edit a binding in the Structure window or on the Overview tab. This is explained further later in this lesson.

Rebinding: Example

In the visual editor or its Structure window, perform the following steps:

1. Right-click the component.
2. Select “Rebind to Another ADF Control.”
3. Select a different control to bind to.



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Rebinding: Example

If you want to use a different data control element for the binding, right-click the component in either the visual editor or its Structure window and select “Rebind to Another ADF Control” from the context menu.

The example in the slide shows how you rebinding an existing data control binding and select a different data control to bind to the component.

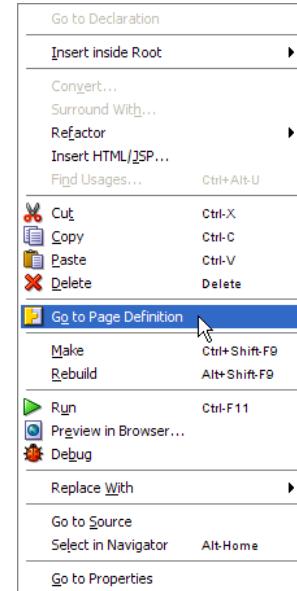
Opening a Page Definition File

The `<pagename>PageDef.xml` page definition file (for example, `browseOrdersPageDef.xml`):

- Is created automatically when you add a databound component to a page
- Contains all the binding definitions for a page

To open a page definition:

1. Right-click the page in the editor or Application Navigator.
2. Select “Go to Page Definition.”



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Opening a Page Definition File

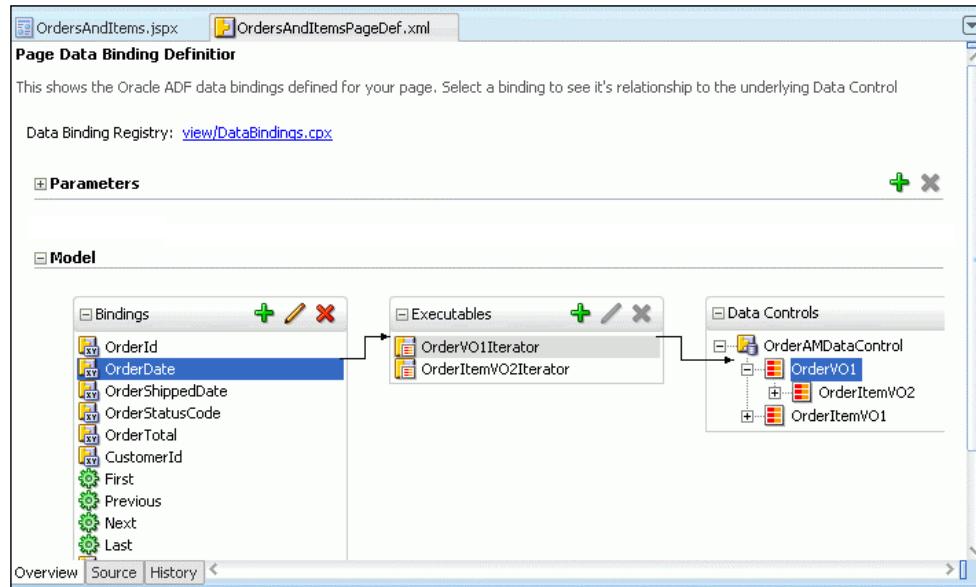
Information about data bindings is not held on the page itself, but in a separate metadata file, `<pagename>PageDef.xml`. This file is created automatically when you first add a data binding to a page, and each time you add databound components to the page, JDeveloper adds appropriate declarative binding entries into this page definition file.

The page definition file is used at run time to instantiate the page’s bindings, which are held in a map called the binding container, accessible during each page request using the EL expression `#{bindings}`. This expression always evaluates to the binding container for the current page. The binding container provides access to the bindings within the page, so there is one page definition file for each databound Web page.

To edit a page definition file, open it in the visual editor or code editor by right-clicking anywhere on the page or page fragment and selecting “Go to Page Definition” from the context menu. You can also view the page definition file bindings by clicking the Bindings tab while editing the `.jsp` or `.jspx` page.

In the Applications Navigator, the page definition file is under the Application Sources node.

Editing Bindings in a Page Definition File



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

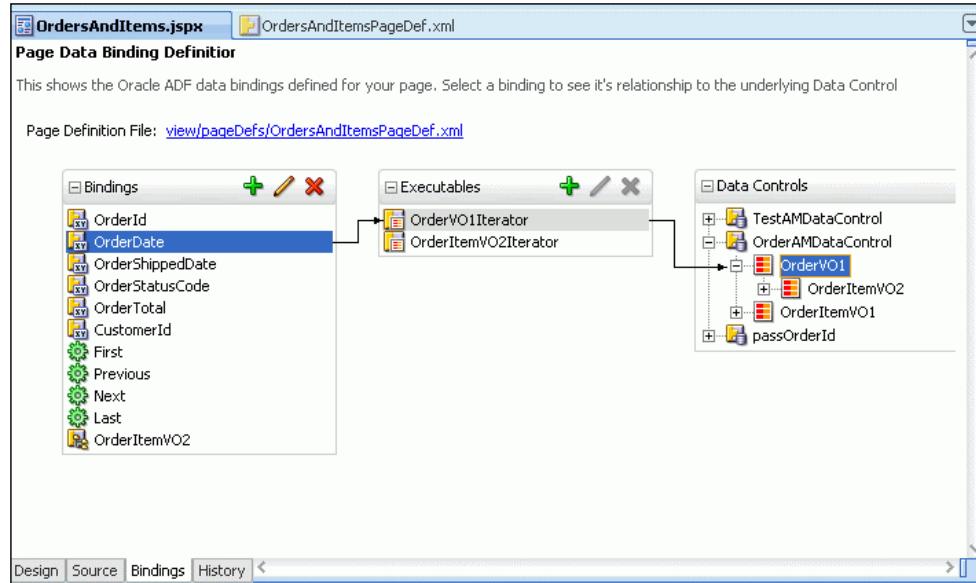
Editing Bindings in a Page Definition File

The elements of the page definition file are displayed in the editor. There are four types of elements:

- **Parameters:** Page parameters, evaluated at the beginning of the request. This is where you would put any parameters that are passed in when the page is requested, and then used as parameters by methods invoked on the page. Page parameters are explained in detail in the lesson titled “Passing Values Between UI Elements.”
- **Bindings:** Value and action bindings on the page
- **Executables:** All iterator bindings used in the page and any method actions that need to be invoked when the page is loaded. At run time, the bindings in the executables element are executed in the order in which they appear in the page definition file
- **Data Controls:** The data source or sources for the page

As you select different bindings in the visual editor, arrows link the selected binding with its iterator and its data control element. You can add, remove, or edit bindings and executables. You can also view the Source or History by clicking those tabs at the bottom of the editor.

Editing Bindings from a Page



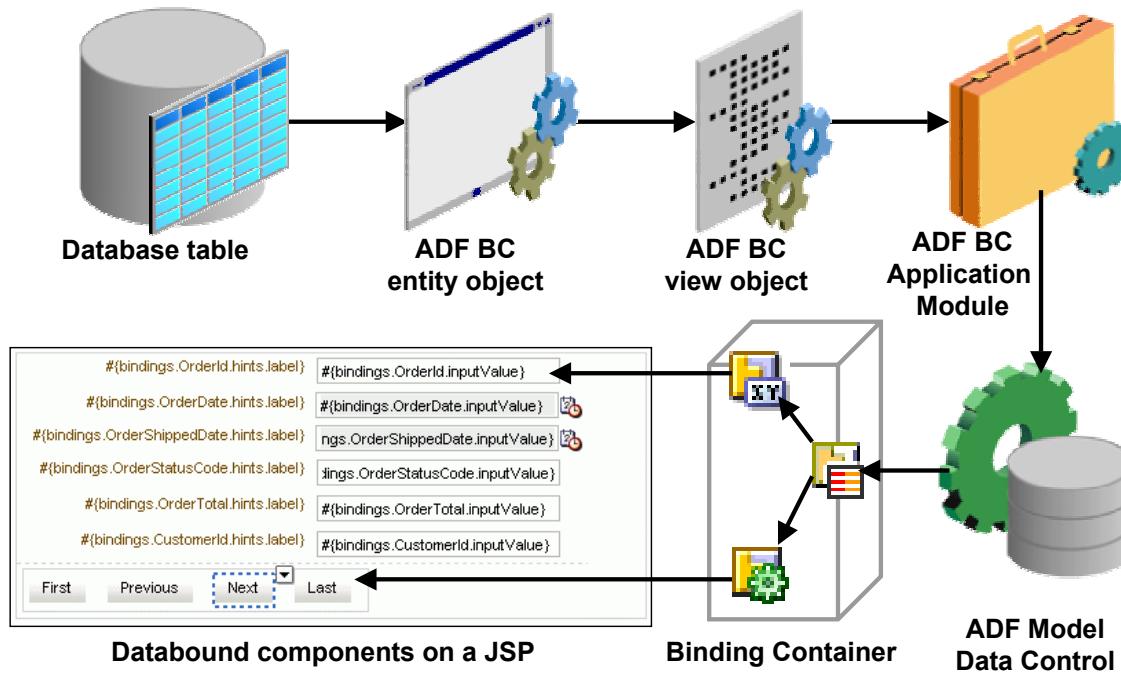
ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Editing Bindings from a Page

When you have a page open in the visual editor, the Bindings tab displays the bindings from its page definition file. Edits that you make on this tab are to the page definition file, not the page—that is, you are editing only the data bindings. Parameters that are available in the page definition file are not displayed on the Bindings tab for the page. On the Bindings tab of the page, there is a link that you can click to open the page definition file editor.

Tracing Data Binding: From Database to Databound Components



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

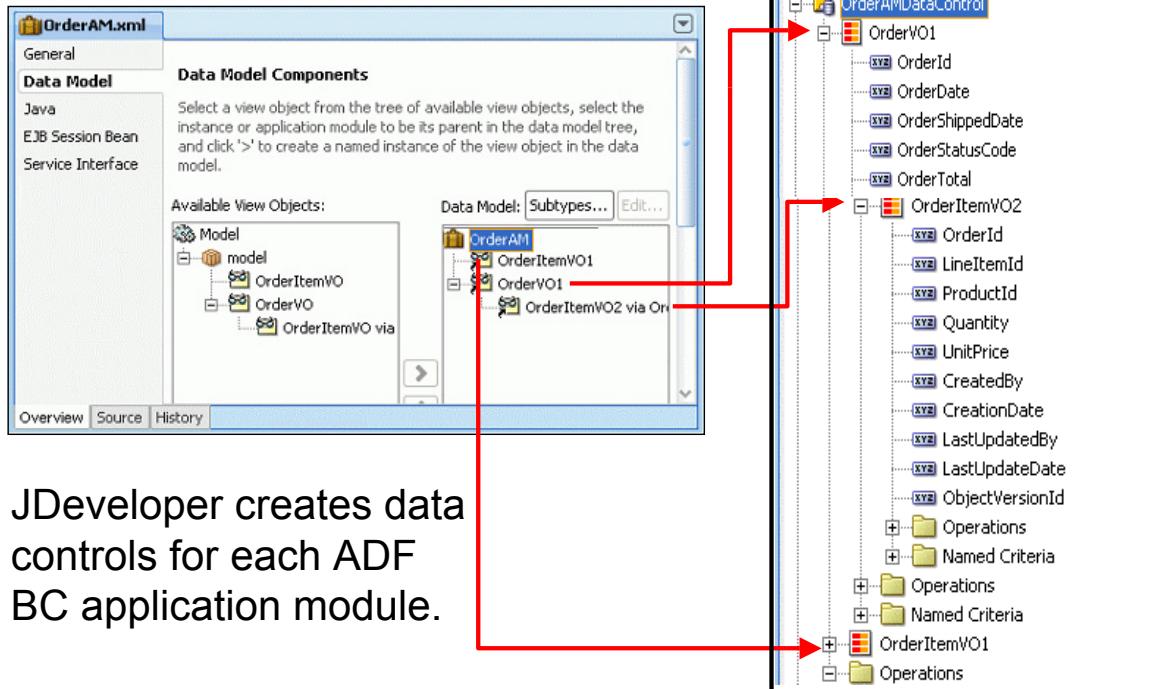
Tracing Data Binding: From Database to Databound Components

It is easy for you as a developer to create databound components, but it is helpful for you to also understand what goes on behind the scenes to enable this simplicity.

This slide illustrates the source of data that appears on the page. Its ultimate source in an ADF BC application is a table in the database. An ADF BC entity object represents that table, and a developer creates an ADF BC view object to present a specific view of the EO data that is required by an application. The developer exposes the VO to an application in an ADF BC application module. When you create an AM, JDeveloper automatically creates a data control for it. You then bind data on a page to elements in the data control, thus enabling access to the back-end ADF BC entity object, and through that to the database table.

The next several slides examine the concept of data binding in more detail, expanding upon the bottom portion of the slide above.

Tracing Data Binding: From AM to Data Control



JDeveloper creates data controls for each ADF BC application module.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Tracing Data Binding: From AM to Data Control

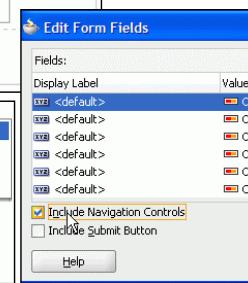
The data control is what ties the back-end data model to the front-end UI.

Tracing Data Binding: Creating Databound Components

Data Control

Page

The screenshot shows a page editor interface with several input fields. Each field has a binding expression like '#{bindings.OrderId.inputValue}' or '#{bindings.CustomerId.inputValue}'. Below the fields is a navigation bar with buttons for 'First', 'Previous', 'Next', and 'Last'.



The Data Controls panel shows the OrderAMDataControl expanded. It contains OrderVO1 with attributes OrderId, OrderDate, OrderShippedDate, OrderStatusCode, OrderTotal, and ObjectVersionId. It also contains OrderItemVO2 and a folder for Operations with various methods like CreateInsert, Create, Delete, Find, Execute, and navigation methods (First, Last, Previous, Next, Previous Set, Next Set).

Example: Drag OrderVO1 to a page and create as an ADF Form with navigation.

ORACLE

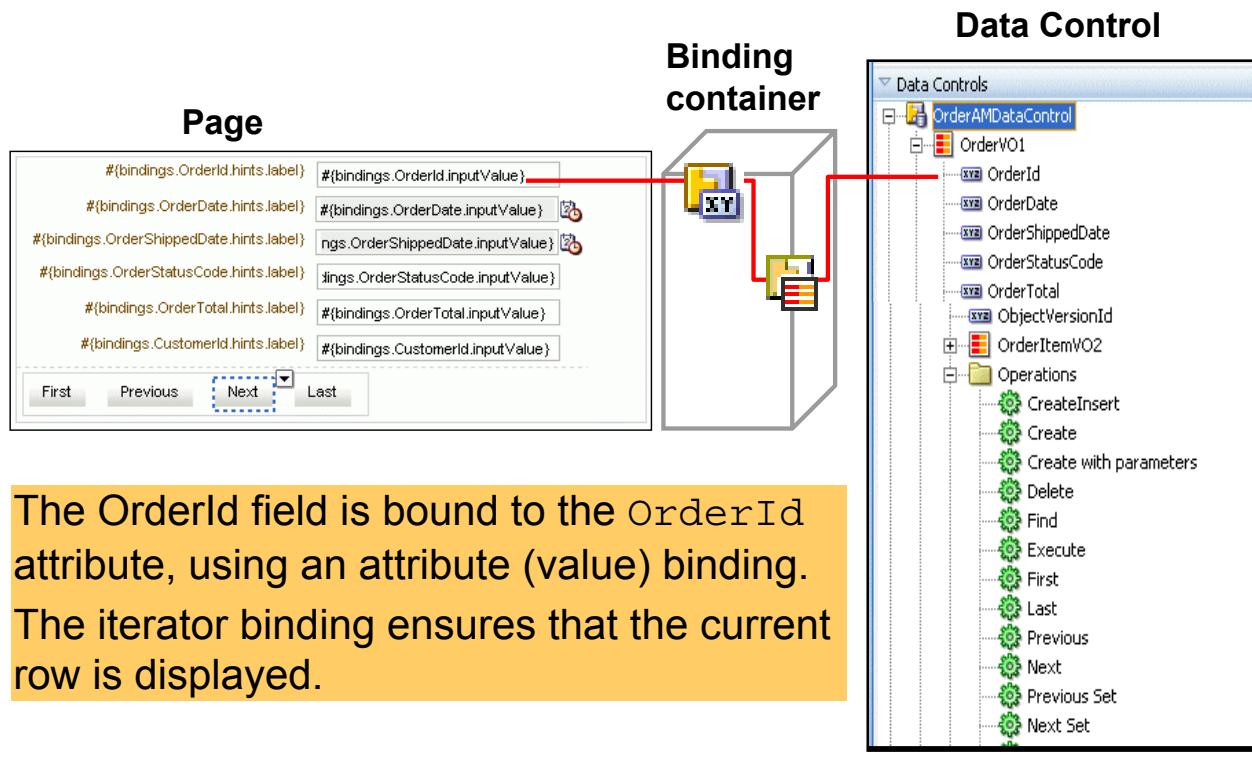
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Tracing Data Binding: Creating Databound Components

When you drag a data control to a page, components thus created are automatically bound to data.

In the slide, the components on the page are bound to the OrderAM data control. They call operations in OrderAM and display data from attributes in OrderAM. Next, you step through the bindings that the page uses.

Tracing Data Binding: From Data Control to Databound Components



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Tracing Data Binding: From Data Control to Databound Components

The attribute binding actually binds to the iterator binding, which then binds to the OrderVO1 object. This is necessary to make sure that the page is displaying the OrderId from the current row. (Remember that the iterator binding keeps track of the current row.)

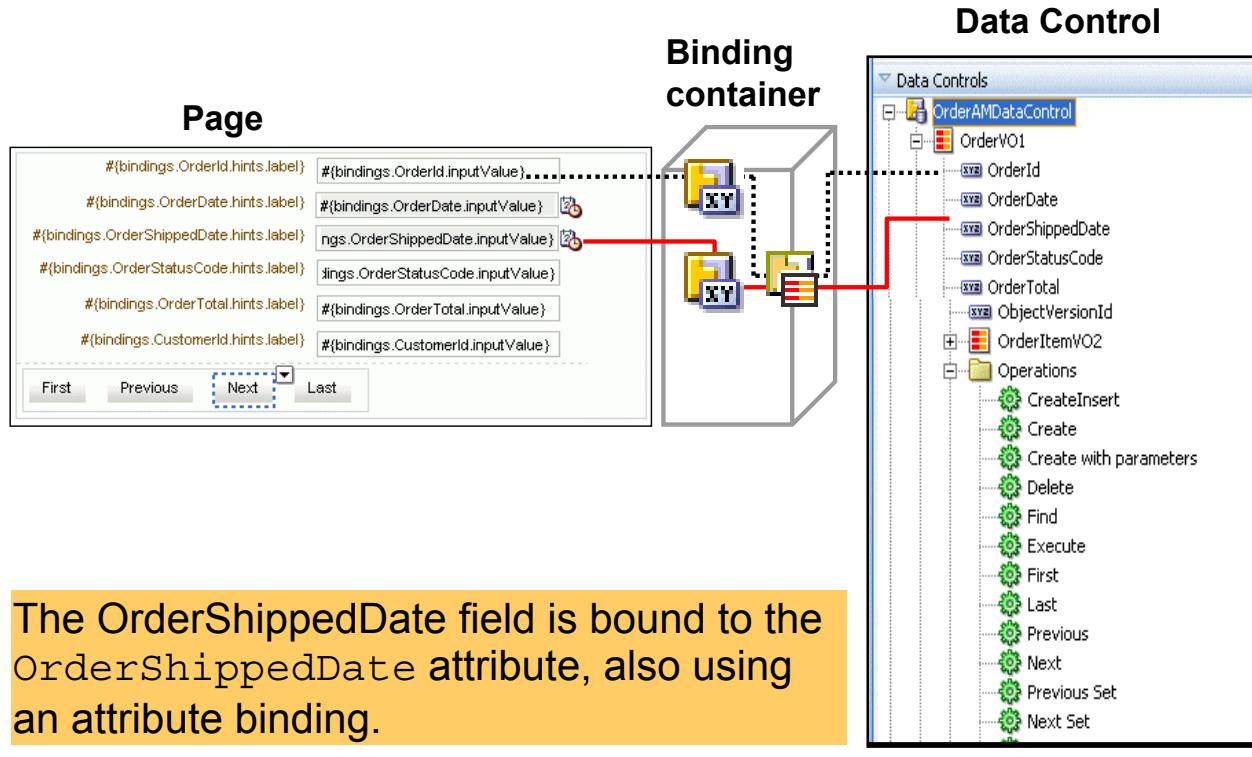
The EL expression used in the data binding for the value of the input text component is:
`#{bindings.OrderId.inputValue}`.

The EL expression used in the data binding for the label of the input text component is:
`#{bindings.OrderId.label}`.

The XML in the page definition file is as follows:

```
<attributeValues IterBinding="OrderVO1Iterator" id="OrderId">
  <AttrNames>
    <Item Value="OrderId"/>
  </AttrNames>
</attributeValues>
```

Tracing Data Binding: From Data Control to Databound Components



The OrderShippedDate field is bound to the OrderShippedDate attribute, also using an attribute binding.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Tracing Data Binding: From Data Control to Databound Components (continued)

Note that an attribute binding is used for any field that displays text, whether it is a read-only value or an editable text field. The EL expression and XML for the OrderShippedDate are similar to the usage for OrderId.

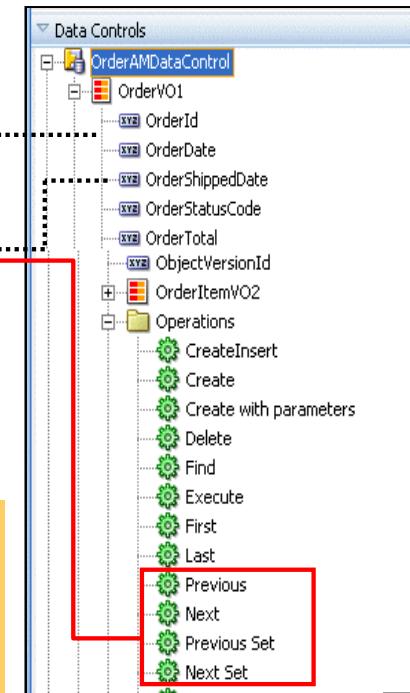
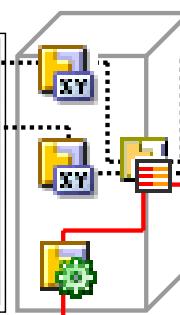
Tracing Data Binding: From Data Control to Databound Components

Data Control

Page

#(bindings.OrderId.hints.label)	#(bindings.OrderId.inputValue).....
#(bindings.OrderDate.hints.label)	#(bindings.OrderDate.inputValue).....
#(bindings.OrderShippedDate.hints.label)	#(bindings.OrderShippedDate.inputValue).....
#(bindings.OrderStatusCode.hints.label)	#(bindings.OrderStatusCode.inputValue).....
#(bindings.OrderTotal.hints.label)	#(bindings.OrderTotal.inputValue).....
#(bindings.CustomerId.hints.label)	#(bindings.CustomerId.inputValue).....
<input type="button" value="First"/> <input type="button" value="Previous"/> <input type="button" value="Next"/> <input type="button" value="Last"/>	

Binding container



The Next button uses an action binding to call the Next operation, which increments the current row in the iterator binding.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Tracing Data Binding: From Data Control to Databound Components (continued)

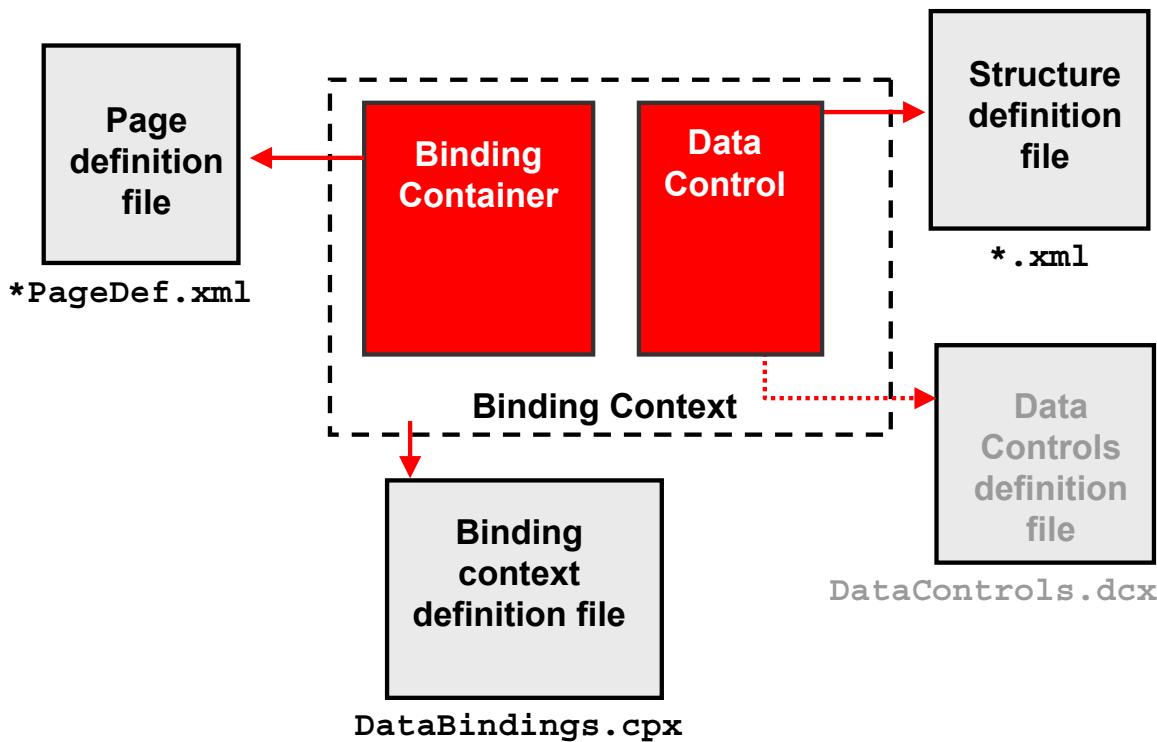
Next is a built-in operation that is available for all collections.

The Previous button uses an operation binding to call the Previous built-in operation. The Previous button is disabled when the page is showing the first row; similarly, the Next button is disabled when the page is showing the last row. This is controlled by the button's disabled attribute:

```
<af:commandButton actionListener="#{bindings.Next.execute}"  
text="Next"  
disabled="#{!bindings.Next.enabled}"/>
```

The disabled attribute is set by a conditional expression. If the Next operation is not enabled, the button is disabled. In the underlying ADF code, the Next operation is enabled only if there are more rows of data to step through.

Examining Data Binding Objects and Metadata Files



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

ORACLE

Data Binding Objects and Metadata Files

All of the bindings used by a particular page or form are grouped in a binding container. In general, there is one binding container per page or form.

The binding context is the handle through which the client accesses the data binding layer. The **DataBindings.cpx** file maps each page to its page definition file and the data controls it uses. It lists all the data controls that are in use.

The page definition file is created automatically the first time that you create a binding on a page. Alternatively, if you select “Go to Page Definition” for a page that does not currently have a page definition, you are prompted to create the page definition file. Page definition files are named *pagenamePageDef.xml*. If you rename a page definition file, you must also change the name of the file in **DataBindings.cpx**. Use the Refactor > Rename operation to perform a consistent rename.

Each binding context, binding container, and data control has its own metadata file. A metadata file is an **.xml** file that contains all the application-specific information that the ADF model run time needs.

Note that ADF BC is different from other data controls in that it does not have a Data Controls definition file, because each application module automatically becomes a data control.

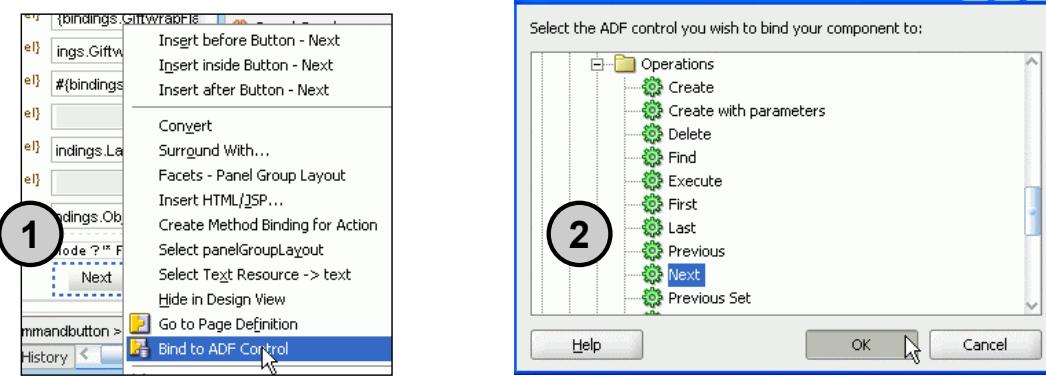
Data Binding Objects and Metadata Files (continued)

One additional metadata file is META-INF/adfm.xml. This file is the registry for the data controls and is only used at design time. The Data Controls panel uses the file to locate the DataControls.dcx file that appears in the data model project when data controls are created manually. (This does not apply to ADF BC data controls, which are created automatically.) The adfm.xml file is deployed with the rest of the application, but is ignored at run time.

Binding Existing Components to Data

You can add data binding to existing components:

1. Right-click the component and select “Bind to ADF Control.”
2. Select a data control object from the “Bind to ADF Control” dialog box.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Binding Existing Components to Data

In the example, you have an existing button on the page. When it is clicked, you want navigation to occur, so that the user can view more than just the first record that is returned by a query. The easiest way to do this is to perform the following steps:

1. Right-click the button and select “Bind to ADF Control.”
2. In the “Bind to ADF Control” dialog box, perform the following steps:
 - a. Expand the view object instance to which the form is bound.
 - b. Expand the Operations node under that view.
 - c. Select the Next operation and click OK.

Alternatively, you can edit the appropriate property in the Property Inspector (such as Label or Value for a text item and ActionListener for a button). You can enter the expression directly, but for some properties, you can click the arrow to the right of the property and select Expression Builder from the context menu. You can then expand the ADF Bindings > bindings nodes to have access to most of the available bindings.

Accessing Data Controls and Bindings Programmatically

- Use methods in DCBindingContainer.
 - Example of accessing an ADF BC application module:

```
DCBindingContainer bc = getBindingContainer();
My AppModule myAM = (My AppModule)bc.findDataControl
("My AppModule Data Control").getDataProvider();
//Now use myAM to call app module methods.
```

- Example of getting an attribute from an iterator binding:

```
DCBindingContainer bc = getBindingContainer();
String empname = (String)bc.findIteratorBinding("empIter")
.getCurrentRow().getAttribute("EmpName");
```

- Access binding container with convenience API:

```
BindingContainer bc =
BindingContext.getCurrent().getCurrentBindingsEntry();
```



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

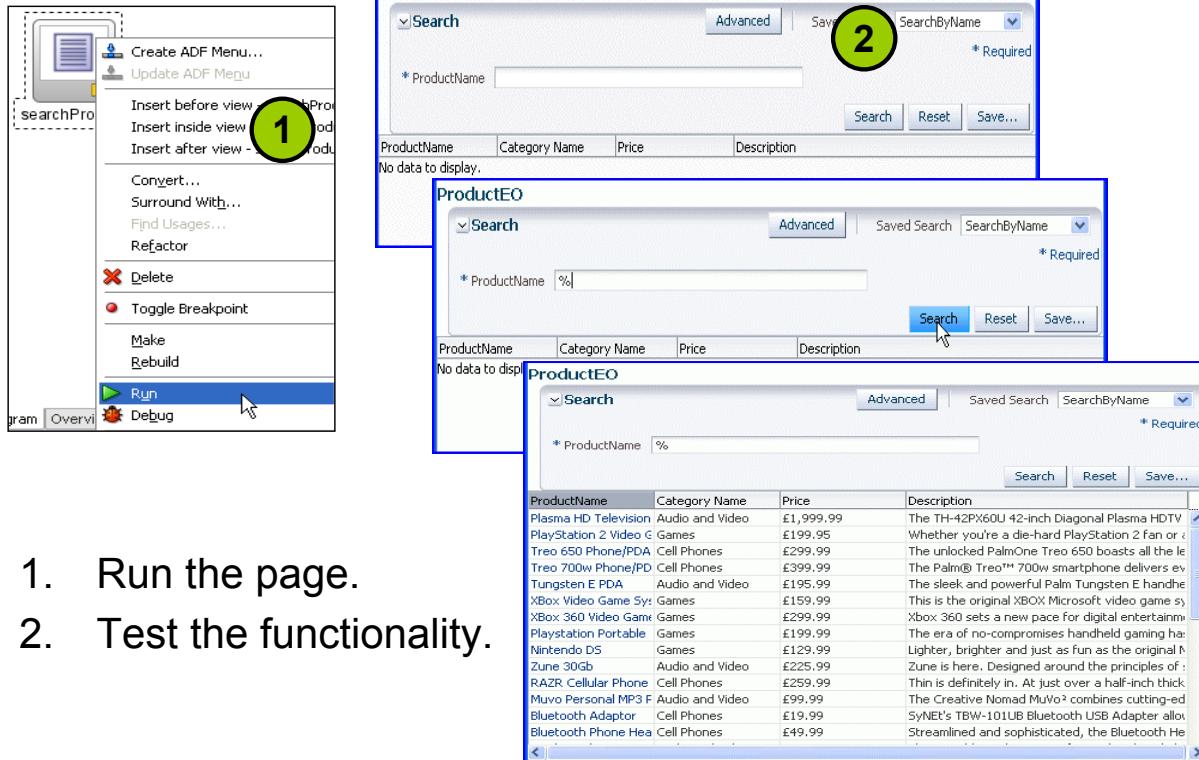
Accessing Data Controls Programmatically

You may need to add code to your JSF-managed bean or ADF Controller action to access data or methods from your data source. In general, it is a better practice to write methods on your data source and expose them in the Data Controls panel, but sometimes that might not be possible. The first example in the slide shows code that accesses the data control to get a handle to its application module.

The second example in the slide shows some sample code that gets an attribute value from an iterator binding. You might need to do this, for example, from a backing bean.

One of the more common requirements in backing bean code is to access the current ADF binding container. A convenience API in JDeveloper enables you to access the binding container as in the code shown in the third example in the slide.

Running and Testing the Page



1. Run the page.
2. Test the functionality.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Running and Testing the Page

To test the page, run it either from the task flow diagram or from the page itself. To run the page, right-click it in the editor and select Run. To run as part of the task flow, right-click the view in the task flow and select Run.

To make testing quicker, you can retest the page without rerunning it if the page definition (data binding information) has not changed. To do so, remove the parameters (everything from ? to the end) from the URL in the browser and reload the page. You can also retest it by clicking the link that displays in the Log window. Make sure to recompile and save any changes before retesting.

Then test the functionality. The example in the slide depicts a default ADF Search Form. To test the functionality, you must enter one or more query criteria, and then click Search to retrieve the queried data.

Summary

In this lesson, you should have learned how to:

- Create a JSF page
- Add ADF Faces UI components to a page
- Include databound components on a page
- Create and edit data bindings



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 11 Overview: Creating Databound Pages

This practice covers the following topics:

- Creating Databound Pages
- Examining the Page's Data Bindings
- Adding an Additional Detail Table
- Renaming a Page



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Practice 11 Overview: Creating Databound Pages

In this set of practices, you create a databound JSF page. You then examine the data bindings and manipulate the data binding files.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Osi S.R.L. use only