

Oracle Fusion Middleware 12c: Build Rich Client Applications with ADF

Student Guide – Volume I

D76564GC10

Edition 1.0

July 2014

D87570

ORACLE®

Authors

DeDe Morton
Lynn Munsinger
Gary Williams

Technical Contributors and Reviewers

Matthew Cooper
Martin Deh
Steven Davelaar
Frédéric Desbiens
Susan Duncan
Brian Fry
Jeff Gallus
Joe Greenwald
Taj-ul Islam
Peter Laseau
Duncan Mills
Chris Muir
Frank Nimphius
Gary Williams
Lynn Munsinger
Katarina Obradovic
Grant Ronald
Shay Shmeltzer
Richard Wright

Editors

Daniel Milne
Richard Wallis
Aju Kumar
Vijayalakshmi Narasimhan

Graphic Designers

Divya Thallap
Maheshwari Krishnamurthy

Publishers

Pavithran Adka
Jayanthy Keshavamurthy
Joseph Fernandez

Copyright © 2014, 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

1 Introduction

Objectives 1-2

Course Agenda: Day 1 1-3

Course Agenda: Day 2 1-4

Course Agenda: Day 3 1-5

Course Agenda: Day 4 1-6

Course Agenda: Day 5 1-7

2 Introduction to Oracle ADF and JDeveloper

Objectives 2-2

Oracle Fusion Middleware Architecture 2-3

How ADF Fits into the Architecture: SOA Integration Example 2-4

Benefits of Oracle ADF 2-6

How Oracle ADF Implements MVC 2-7

Introduction to Oracle JDeveloper 2-9

Launching JDeveloper on Windows 2-11

Selecting a Role 2-13

Setting Preferences 2-14

Applications Window 2-15

JDeveloper Editors 2-17

Types of Editors 2-18

Team Menu for Source Control Integration 2-19

Components Window 2-21

Structure Window 2-22

Properties Window 2-23

Log Window 2-24

Other Useful Windows 2-25

Obtaining Help 2-26

Getting Started in JDeveloper 2-27

Creating an Application in JDeveloper 2-28

Application Overview and Checklist	2-30
Editing Application Properties	2-31
Creating a Project in JDeveloper	2-32
Editing Project Properties	2-34
Creating a Database Connection in JDeveloper	2-35
Summary	2-37
Practice 2 Overview: Using JDeveloper	2-38
3 Building a Business Model with ADF Business Components	
Objectives	3-2
Building the Business Services Layer	3-3
ADF Business Components	3-4
Types of ADF Business Components	3-5
Entity Objects (EOs)	3-7
Associations	3-8
Types of ADF Business Components: Entity Objects Summary	3-9
View Objects (VOs)	3-10
Types of View Objects	3-11
View Links	3-13
Master-Detail Relationships in the UI	3-14
Interaction Between View Objects and Entity Objects: Retrieving Data	3-15
Interaction Between View Objects and Entity Objects: Retrieving Read-Only Data	3-16
Interaction Between View Objects and Entity Objects: Persisting Data	3-18
Types of ADF Business Components: View Objects Summary	3-19
Application Modules	3-20
Types of ADF Business Components: Summary	3-22
Creating ADF Business Components	3-23
Create Business Components from Tables Wizard: Entity Objects	3-25
Create Business Components from Tables Wizard: Entity-Based View Objects	3-26
Create Business Components from Tables Wizard: Query-Based View Objects	3-27
Create Business Components from Tables Wizard: Application Module	3-29
Create Business Components from Tables Wizard: Diagram	3-30
Launching Individual Wizards to Create Business Components	3-31
Testing the Data Model	3-32
Refactoring Components	3-33
Summary	3-34
Practice 3 Overview: Creating and Testing ADF Business Components	3-35

4 Creating Data-Bound UI Components

- Objectives 4-2
- Building the View Layer 4-3
- In the Beginning: Static HTML 4-4
- Dynamic Webpage Technologies: CGI 4-5
- Dynamic Webpage Technologies: Servlets 4-6
- Dynamic Webpage Technologies: JavaServer Pages (JSP) 4-7
- Dynamic Webpage Technologies: JavaServer Faces (JSF) 1.1 and 1.2 4-9
- Dynamic Webpage Technologies: JSF 2.0 Enhancements 4-11
- The JSF Component Architecture: Overview 4-13
- The JSF Component Architecture: UI Components 4-14
- The JSF Component Architecture: Managed Beans 4-15
- The JSF Component Architecture: Expression Language 4-16
- The JSF Component Architecture: JSF Controller 4-17
- The JSF Component Architecture: Renderers and Render Kits 4-18
- Standard JSF Components 4-20
- ADF Faces Rich Client Components 4-21
- Standard JSF and ADF Compared 4-23
- Creating a JSF Page in JDeveloper 4-25
- Example: Two Column Layout 4-27
- Adding UI Components to the Page 4-29
- Using the Data Controls Panel 4-30
- Example: Input Text Component with Bindings 4-31
- Using the Components Window 4-32
- Using the Context Menu 4-34
- Using the Code Editor 4-35
- Running and Testing the Page 4-36
- Summary 4-38
- Practice 4 Overview: Creating and Running a JSF Page 4-39

5 Defining Task Flows and Adding Navigation

- Objectives 5-2
- Building the Controller Layer 5-3
- Traditional Navigation Compared to JSF Controller 5-4
- JSF Navigation: Example 5-5
- ADF Controller 5-7
- ADF Controller: Task Flow Example 5-9
- Creating Task Flows 5-11
- Defining Activities and Control Flow Rules 5-12
- Defining Global Navigation with Wildcards 5-14
- Defining a Wildcard Control Flow Rule 5-16

ADF Faces Navigation Components	5-17
Adding Navigation Components to a Page	5-19
Summary	5-20
Practice 5 Overview: Defining Task Flows and Adding Navigation	5-21

6 Declaratively Customizing ADF Business Components

Objectives	6-2
Customizing Business Components	6-3
Editing Business Components	6-4
Editing Entity Objects	6-5
Modifying the Default Behavior of Entity Objects	6-7
Defining Attribute UI Hints	6-8
Creating Transient Attributes	6-10
Defaulting values	6-11
Synchronizing with Trigger-Assigned Values	6-12
Defining Alternate Key Values	6-13
Synchronizing an Entity Object with Changes to Its Database Table	6-14
Editing View Objects	6-15
Modifying the Default Behavior of View Objects	6-17
Tuning View Objects	6-18
Overriding Attribute UI Hints	6-19
Performing Calculations	6-20
Restricting and Reordering Columns	6-21
Restricting the Rows Retrieved by a Query	6-22
Changing the Order of Queried Rows	6-23
Using Parameterized WHERE Clauses	6-24
Creating Named Bind Variables	6-25
Creating Named View Criteria (Structured WHERE Clauses)	6-27
Applying Named View Criteria to a View Object Instance	6-29
Creating Join View Objects	6-30
Selecting Attributes in Join View Objects	6-32
Creating More Efficient Queries by Using Declarative View Objects	6-33
Creating Master-Detail Relationships between View Objects	6-35
Linking View Objects	6-36
Model-Driven List of Values (LOV)	6-38
Model-Driven List of Values: Example	6-39
Defining the LOV	6-40
Cascading (Dependent) LOVs	6-42
Defining Cascading LOVs	6-43
Modifying Application Modules	6-45
Changing the Database Connection	6-46

Determining the size of the Application Module	6-47
Application Module Nesting	6-48
Defining Nested Application Modules	6-49
Summary	6-50
Practice 6 Overview: Declaratively Customizing ADF BC	6-51

7 Validating User Input

Objectives	7-2
Adding Validation to Business Components	7-3
Validation Options for ADF BC Applications	7-4
Defining Validation Rules in ADF BC	7-5
Validation Rule Types: Entity and Attribute-level	7-6
Validation Rule Types: Entity-Level Only	7-7
Creating Validation Rules	7-8
Specifying the Rule Definition	7-9
Specifying Conditions for Executing Validation Rules	7-10
Specifying Error Messages to Handle Failures	7-11
Introduction to Groovy	7-12
Using Groovy Syntax in ADF	7-13
Editing Groovy Expressions	7-15
Internationalizing Messages and Other Translatable Text	7-16
Steps to Internationalize an Application	7-18
Resource Bundles	7-19
Creating Resource Bundles: Setting Resource Bundle Options	7-20
Creating Resource Bundles: Model	7-21
Creating Resource Bundles: ViewController	7-22
Creating Localized Resource Bundles	7-24
Configuring the Application to Support Locales	7-25
Other Approaches	7-26
Summary	7-28
Practice 7 Overview: Validating User Input	7-29

8 Modifying Data Bindings Between the UI and the Data Model

Objectives	8-2
Modifying Data Bindings in the Model Layer	8-3
Oracle ADF Model Layer: Review	8-4
ADF Data Controls	8-5
Creating ADF Data Controls	8-6
Using ADF Data Controls	8-7
ADF Bindings	8-8
Expression Language (EL) and Bindings	8-9

Viewing Data Bindings in the Page Definition File	8-11
Examining the Page Definition File	8-12
Categories of Data Bindings	8-13
Editing Data Bindings	8-15
Editing in the Page Definition File	8-16
Editing in the Properties Window	8-17
Editing by Binding an Existing Component to a Data Control	8-18
Editing by Rebinding an Existing Component to a Different Data Control	8-19
How Bindings Work Behind the Scenes	8-20
Example: Value Bindings for an Input Text Component	8-21
Examining the Binding Context and Metadata Files	8-22
Summary	8-23
Practice 8 Overview: Modifying Data Bindings	8-24

9 Adding Functionality to Pages

Objectives	9-2
Adding Functionality to Pages in the View Layer	9-3
ADF Faces Rich Client Components	9-4
Defining General Controls	9-5
Defining Text and Selection Components	9-6
Defining Lists for Selection Components	9-7
Defining Data Views	9-9
Defining Tables	9-10
Specifying Table Properties	9-11
Examining Table Bindings	9-13
How Table Data Is Rendered through Stamping	9-14
Defining Trees	9-15
Examining Tree Bindings	9-17
Defining Query Forms	9-19
More About Named View Criteria Used for Queries	9-21
Specifying Query Panel Properties	9-22
Specifying a Display Component for a Query Panel	9-24
Defining Menus and Toolbars	9-25
Defining Layout Components	9-26
Defining Data Visualization (DVT) Components	9-27
Shaping Data for DVT components	9-28
ADF Faces Resources	9-29
Adding Application Code to Managed Beans and Backing Beans	9-30
Creating a Backing Bean as a Managed Bean	9-31
Registering a Bean Class as a Managed Bean	9-33
Calling a Managed Bean from a JSF Page	9-34

Example: Enabling and Disabling Components Programmatically 9-36
Example: Enabling and Disabling Components Declaratively 9-37
Summary 9-38
Practice 9 Overview: Adding Functionality to Pages 9-39

10 Adding Advanced Features to Task Flows and Page Navigation

Objectives 10-2
Adding Advanced Features to Task Flows and Navigation 10-3
More about Task Flows 10-4
Unbounded ADF Task Flows 10-5
Bounded Task Flows 10-6
Task Flow Example 10-7
Review: Creating Task Flows 10-8
Working with the Task Flow Editor 10-9
Task Flow Activities 10-10
Adding Conditional Navigation to a Task Flow 10-11
Defining Router Activities 10-12
Calling a Method from a Task Flow 10-13
Calling Other Task Flows from a Task Flow 10-15
Returning from a Task Flow 10-16
Making View Activities in Unbounded Task Flows Bookmarkable
(or Redirecting) 10-17
Converting Task Flows Between Bounded and Unbounded 10-18
Converting a Bounded Task Flow to Use Page Fragments 10-19
Using Bounded Task Flows 10-20
Using Regions on a Page 10-21
Defining a Bounded Task Flow as a Region 10-22
Creating Navigation Components 10-23
Navigation Menus and Toolbars 10-24
Defining Navigation Menus 10-25
Defining Access (Shortcut) Keys 10-27
Defining Cascading Menus 10-29
Creating Pop-Up Menus 10-30
Creating Context Menus 10-31
Creating Breadcrumbs 10-32
Creating a Navigation Pane 10-33
Using Trains 10-35
Creating Trains 10-36
Skipping a Train Stop 10-38
Summary 10-39
Practice 10 Overview: Adding Advanced Features to Task Flows 10-40

11 Passing Values Between UI Elements

- Objectives 11-2
- Passing Values Between UI Elements 11-3
- Holding Values in the Data Model (Business Components) 11-4
- Holding Values in Managed Beans 11-5
- Storing Values in Bean Properties 11-6
- Standard JSF Memory Scopes 11-7
- ADF Memory Scopes 11-8
- Memory Scope Duration with a Called Task Flow 11-10
- Memory Scope Duration with a Region 11-11
- Accessing ADF Memory Scopes 11-12
- Storing Values in Memory Scoped Attributes 11-13
- Setting the Value of a Memory-Scoped Attribute by Using a Set Property Listener 11-14
- Using Parameters to Pass Values 11-15
- Passing Values from a Containing Page to a Reusable Page Fragment in a Region 11-16
- Using Page Parameters 11-17
- Role of the Page Parameter 11-18
- Using Task Flow Parameters 11-19
- Role of the Task Flow Parameter 11-20
- Using Task Flow Binding Parameters 11-21
- Role of the Task Flow Binding Parameter 11-22
- Using View Activity Parameters 11-23
- Role of the View Activity Parameter 11-24
- Passing Values to a Task Flow from a Task Flow Call Activity 11-26
- Returning Values to a Calling Task Flow 11-28
- Deciding Which Kinds of Parameters to Use 11-29
- Summary 11-30
- Practice 11 Overview: Passing Values Between UI Elements 11-31

12 Responding to Application Events

- Objectives 12-2
- Responding to Application Events 12-3
- JSF Lifecycle Phases 12-4
- Phases of the ADF Life Cycle 12-6
- Partial Page Rendering (PPR) 12-9
- Guidelines for Using PPR 12-10
- Native PPR 12-12
- Enabling PPR Declaratively 12-13
- Enabling Automatic PPR 12-14

Using the immediate Attribute	12-15
Value Change Events	12-16
Using Value Change Event Listeners	12-17
Action Events	12-18
Creating Action Methods	12-19
Using Action Event Listeners	12-20
Other ADF Faces Server Events	12-21
Using Tree Model Methods in Selection Listeners	12-22
Summary	12-23
Practice 12 Overview: Responding to Application Events	12-24

13 Programmatically Implementing Business Service Functionality

Objectives	13-2
Programmatically Customizing the Data Model	13-3
Deciding Where to Add Custom Code	13-4
Overview of the Framework Classes for ADF Business Components	13-6
Generating Java Classes to Add Custom Code	13-7
Customizing Entity Objects Programmatically	13-8
Commonly Used Methods in EntityImpl	13-9
Generating Accessors	13-11
Overriding Base Class Methods to Modify Behavior	13-12
Example: Overriding remove() and doDML()	13-13
Example: Traversing Associations	13-14
Creating a Method Validator for an Entity Object or Attribute	13-15
Customizing View Objects Programmatically	13-17
Commonly Used Methods in ViewObjectImpl	13-18
Example: Setting WHERE Clauses Programmatically	13-19
Using Code Insight	13-20
Example: Setting the Value of Named Variables at Run Time	13-21
Using View Criteria with View Objects	13-22
Example: Applying View Criteria Programmatically	13-23
Working with View Rows	13-24
Commonly Used Methods in RowIterator and RowSet (or RowSetImpl)	13-25
Commonly Used Methods in ViewRowImpl and Row	13-26
Example: Finding a Row and Setting It as the Current Row	13-27
Customizing View Object Rows by Subclassing ViewRowImpl	13-28
Example: Iterating through Detail Rows to Update an Attribute	13-29
Exposing Custom Methods in the View Object and View Object Row Client Interfaces	13-30
Customizing Application Modules Programmatically	13-31
Creating Custom Service Methods	13-32

Exposing Custom Service Methods in the Client Interface	13-33
Testing the Client Interface in the Oracle ADF Model Tester	13-35
Creating Extension Classes for ADF Business Components	13-36
Accessing Data and Operations Through the Model Layer	13-38
ADF Binding Types	13-40
Java Classes Behind the ADF Bindings	13-41
Exploring the Class Hierarchy	13-42
Commonly Used Methods in the BindingContext Class	13-43
Commonly Used Methods in the BindingContainer Class	13-44
Example: Accessing ADF Bindings from a Backing Bean	13-45
A Closer Look at the Example	13-46
Summary	13-47
Practice 13 Overview: Programmatically Customizing the Data Model	13-48

14 Implementing Transactional Capabilities

Objectives	14-2
Implementing Transactional Capabilities in the Controller Layer	14-3
Handling Transactions with ADF Business Components	14-4
Default ADF Model Transactions	14-5
Task Flows and Application Modules	14-6
Data Control Scopes	14-7
Shared Data Control Scope	14-8
Isolated Data Control Scope	14-9
Data Control Frame	14-10
Data Control Frame: Examples	14-11
Specifying Transaction Options	14-12
“No Controller” Option	14-13
<No Controller Transaction> Option	14-14
Controlling Transactions in Task Flows	14-15
“Always Begin New Transaction” Option	14-16
“Always Use Existing Transaction” Option	14-17
“Use Existing Transaction if Possible” Option	14-18
Using the Task Flow Transaction Options	14-19
Transaction Support Features of Bounded Task Flows	14-20
Specifying Task Flow Return Options	14-21
Handling Transaction Exceptions	14-22
Designating an Exception Handler	14-23
Defining Responses to the Back Button	14-24
Summary	14-26
Practice 14 Overview: Implementing Transactional Capabilities	14-27

15 Building Reusability into Pages

- Objectives 15-2
- Building Reusability into Pages 15-3
- ADF Reusability Features 15-4
- Page Templates 15-5
 - Components of a Page Template 15-6
 - Creating Page Templates 15-7
 - Defining Page Templates 15-8
 - Creating a Page Template 15-9
 - Editing Page Templates 15-11
 - Nesting Page Templates 15-12
 - Applying a Page Template to a Page 15-13
- Page Fragments 15-14
 - Creating a Page Fragment 15-15
 - Using a Page Fragment on a Page 15-16
- Regions: Review 15-17
- ADF Libraries 15-18
 - Packaging Reusable Components into Libraries 15-20
 - Creating an ADF Library 15-21
 - Adding an ADF Library to a Project 15-22
 - Viewing the ADF Libraries That Are Applied to a Project 15-23
 - Guidelines for Using ADF Libraries 15-24
 - Restricting Business Component Visibility in Libraries 15-25
 - Removing an ADF Library from a Project 15-26
 - Summary 15-27
- Practice 15 Overview: Building Reusability into Pages 15-28

16 Achieving the Required Layout

- Objectives 16-2
- Achieving the Required Layout 16-3
- Overview of ADF Faces Layout Components 16-4
- Layout Containers 16-5
 - Interactive Layout Containers 16-6
 - Geometry Management of Layout Containers 16-7
 - Creating Layouts 16-9
 - Best Practices for Page Layout 16-11
 - Laying Out Components in a Grid: Panel Grid Layout Component 16-12
 - Laying Out Components to Stretch Across a Page: Panel Stretch Layout Component 16-14
 - Grouping Related Components: Panel Group Layout Component 16-15
 - Laying Out Components in Forms: Panel Form Layout Component 16-17

Laying Out Components in a Dashboard: Panel Dashboard Component	16-19
Creating Resizable Panes: Panel Splitter Component	16-20
Displaying or Hiding Content in Panels: Panel Accordion Component	16-21
Adding Blank Space and Lines to Layouts: Spacer and Separator Components	16-22
Using Quick Start Layouts	16-23
Adding a “Show Printable Page Behavior”	16-24
Adding the Ability to Export to Excel	16-25
ADF Faces and Skinning	16-26
Using Expression Language to Display Components Conditionally	16-27
Summary	16-29
Practice 16 Overview: Achieving the Required Layout	16-30

17 Debugging ADF Applications

Objectives	17-2
Overview of Troubleshooting Tools	17-3
Troubleshooting Techniques	17-4
Reading the Message Log	17-5
Using the Oracle ADF Model Tester	17-6
ADF Logger	17-7
Configuring ADF Logging	17-8
Creating Logging Messages	17-10
Examples of Key Log Points	17-11
Viewing Log Messages in the Log Analyzer	17-12
Logging ADF Trace Information	17-13
Vary Log Messages by Role	17-14
JDeveloper Debugger	17-15
Understanding Breakpoint Types	17-16
Setting Breakpoints	17-18
ADF Declarative Debugger	17-19
Using the EL Evaluator at Breakpoints	17-21
Developing Regression Tests with JUnit	17-22
Summary	17-23
Practice 17 Overview: Debugging	17-24

18 Implementing Security in ADF Applications

Objectives	18-2
Benefits of Securing Web Applications	18-3
Authentication and Authorization	18-4
ADF Security Framework and OPSS	18-5
Securing the Layers of an ADF Application	18-6

Steps for Implementing ADF Security	18-7
Configuring ADF Security	18-8
Creating Test Users and Enterprise Roles	18-10
Creating Application Roles	18-11
Defining Security Policies	18-13
Granting Access to Bounded Task Flows	18-15
Granting Access to Pages	18-17
Making ADF Resources Public	18-18
Granting Access to Business Services	18-19
Enabling Security on Entity Objects and Attributes	18-20
Granting Privileges on Entity Objects and Attributes	18-21
Application Authentication at Run Time	18-22
ADF Security: Implicit Authentication	18-23
ADF Security: Explicit Authentication	18-24
ADF Security: Authorization at Run Time	18-25
Implementing a Login Page for Implicit Authentication	18-26
Creating a Custom Login Page That Uses ADF Faces components	18-28
Step 1: Creating a Managed Bean for Login	18-29
Step 2: Creating a Custom Login Page with ADF Faces Components	18-31
Step 3: Configuring ADF Security to Use a Custom Login Page	18-32
Programmatically Accessing ADF Security	18-33
Using the Expression Language to Extend Security Capabilities	18-34
Using Global Security Expressions	18-35
Using a Security Proxy Bean	18-36
Summary	18-37
Practice 18 Overview: Implementing Security	18-38

19 Deploying ADF BC Applications

Objectives	19-2
Basic Deployment Steps	19-3
Preparing an Application for Deployment	19-4
Setting Security Deployment Options	19-6
Synchronizing JDBC Deployment Descriptors	19-7
Creating Deployment Profiles	19-10
Changing the Context Root for an Application	19-11
Preparing the Oracle WebLogic Server (WLS) for Deployment	19-12
Options for Deploying an Application	19-13
Creating a Connection to an Application Server	19-14
Deploying the Application	19-15
Using ojdeploy to Build Files for Deployment	19-16
Summary	19-17

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

Oracle University and NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

1 Introduction

Oracle Fusion Middleware 12c:
Build Rich Client Applications with ADF



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this course, you should be able to:

- Build reusable business services by implementing ADF Business Components
- Expose the data model in a web application by using ADF Model
- Build a rich user interface by using ADF Faces
- Develop reusable task flows for an application
- Test, debug, secure, and deploy the web application



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Aim

This course teaches developers how to build web applications by using ADF Business Components (ADF BC), ADF Faces components, ADF data binding, and task flows. Participants use Oracle JDeveloper 12c to build, test, and deploy web applications.

Course Agenda: Day 1

Learn the basics of ADF and build a simple application:

- Lesson 2: Introduction to Oracle ADF and JDeveloper
- Lesson 3: Building a Business Model with ADF Business Components
- Lesson 4: Creating Data-Bound UI Components
- Lesson 5: Defining Task Flows and Adding Navigation



Copyright © 2014, 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

Customize business services, add validation, and modify bindings:

- Lesson 6: Declaratively Customizing ADF Business Components
- Lesson 7: Validating User Input
- Lesson 8: Modifying Data Bindings Between the UI and the Data Model



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Agenda: Day 3

Add functionality to pages, add page navigation, and pass values between UI elements:

- Lesson 9: Adding Functionality to Pages
- Lesson 10: Adding Advanced Features to Task Flows and Page Navigation
- Lesson 11: Passing Values Between UI Elements



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Agenda: Day 4

Respond to application events, extend the application programmatically, add transactional capabilities, and build reusability into pages:

- Lesson 12: Responding to Application Events
- Lesson 13: Programmatically Implementing Business Service Functionality
- Lesson 14: Implementing Transactional Capabilities
- Lesson 15: Building Reusability into Pages



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Course Agenda: Day 5

Improve the user interface, troubleshoot problems, and secure and deploy the application:

- Lesson 16: Achieving the Required Layout
- Lesson 17: Debugging ADF Applications
- Lesson 18: Implementing Security in ADF Applications
- Lesson 19: Deploying ADF BC Applications



Copyright © 2014, 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 NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

Introduction to Oracle ADF and JDeveloper

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe the Oracle Fusion Middleware architecture
- Explain how ADF fits into the Oracle Fusion Middleware architecture
- Describe the ADF technology stack and how it implements the Model-View-Controller (MVC) design pattern
- List benefits that JDeveloper provides for application development
- Identify and describe the main windows and editors available in the JDeveloper IDE
- Create applications, projects, and database connections in JDeveloper

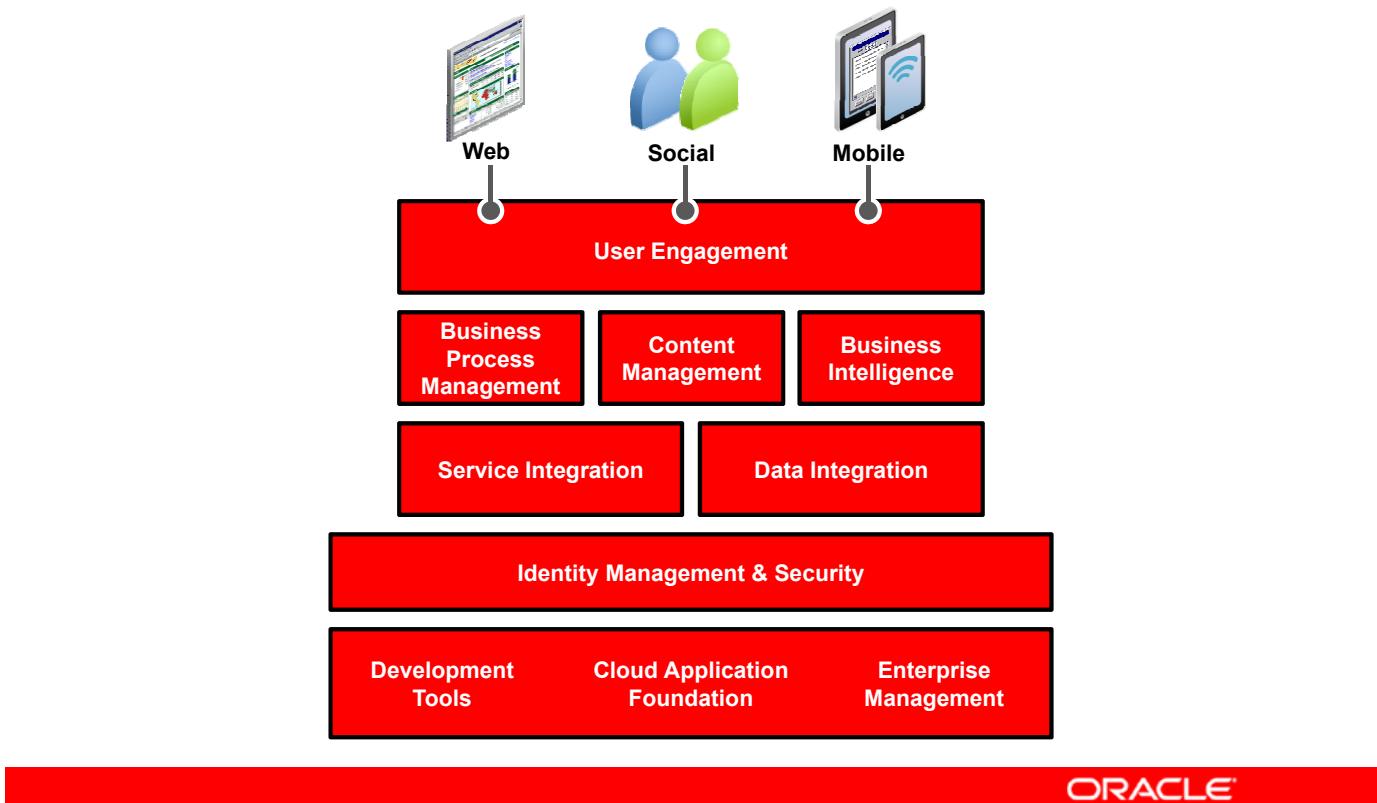


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson provides an overview of the Oracle Fusion Middleware architecture and describes how ADF fits into the overall architecture. This lesson also introduces you to the JDeveloper IDE.

Oracle Fusion Middleware Architecture

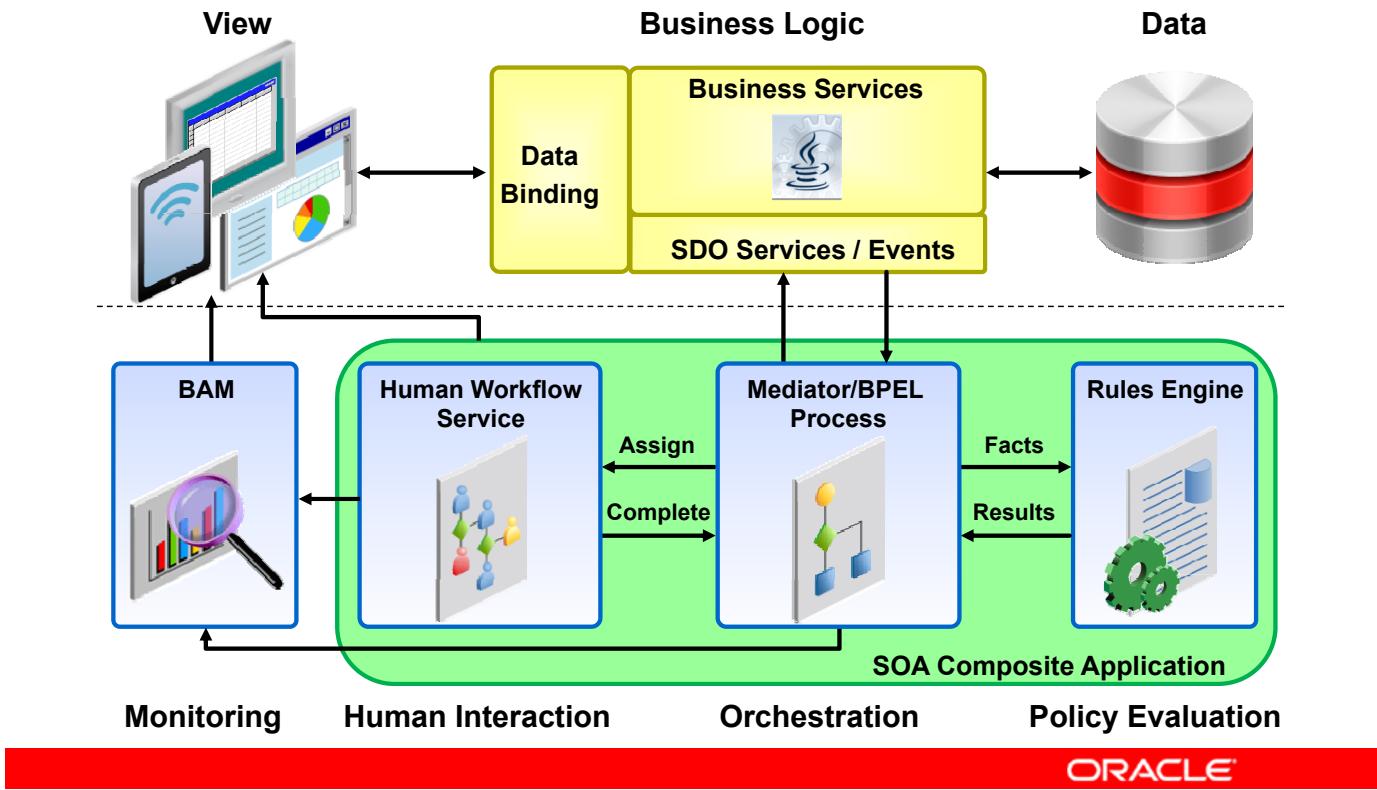


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle Fusion Middleware is a family of application infrastructure products that are used to create, run, and manage business applications. Oracle Fusion Middleware provides the benefits of common security, management, deployment architecture, and development tools. Oracle Fusion Middleware is based on industry standards, such as service-oriented architecture (SOA) and Java Platform, Enterprise Edition (Java EE), and is integrated with Oracle Applications and technologies that increase developer productivity.

Oracle Application Development Framework (ADF) is a critical part of the Oracle Fusion Middleware technology stack and is used in many of the blocks depicted in the diagram. Oracle ADF is also a key technology used internally by Oracle to implement all “next generation” web user interfaces, including Fusion Applications, and all middleware components, such as Enterprise Manager and administration consoles.

How ADF Fits into the Architecture: SOA Integration Example



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

Oracle ADF is an end-to-end application development framework for building Java Platform, Enterprise Edition applications. To illustrate how Oracle ADF fits into the overall architecture, this diagram shows a SOA integration scenario. The diagram is divided into two sections. The top section represents a simplified view of an enterprise application developed in ADF. It includes a database and various business services based upon the database objects. The arrow to the database represents the object-relational mapping between the business objects and the database. The business services interact with the database to read and write information, and they perform validation and execute operations on the underlying data. With Oracle ADF, you can develop the business services once, and build multiple user interfaces that access the same business services through a generic data-binding layer. You can access the business services from a variety of user interface implementations, including web, mobile, and Microsoft Office.

The lower part of the slide represents the Oracle SOA Suite. Everything depicted in the lower part of the slide is part of the Fusion Middleware platform. In this example, the SOA composite application includes three components:

- A Mediator and/or a Business Process Execution Language (BPEL) component. The Mediator component is responsible for subscribing to business events and routing events between different components. The BPEL component provides business process orchestration. It enables you to define business processes that integrate a series of business activities and services into an end-to-end process flow.

- The Business Rule component allows you to define rules that enable dynamic decisions at runtime. You automate policies, constraints, computations, and reasoning while separating rule logic from underlying application code.
- The Human Workflow component provides a mechanism for defining human interaction with the business process.

The Business Activity Monitoring (BAM) component within SOA Suite provides the ability to build interactive, real-time dashboards and proactive alerts to monitor business services and processes.

You can integrate ADF applications with SOA Suite components in the following ways:

- The Mediator component of the SOA composite application can subscribe to events that are published by ADF business components. When an event is generated, the Mediator component translates the contents of the event into a message and routes the message to the appropriate component in the SOA Composite Application.
- The SOA composite application can execute ADF business components that are exposed as web services in the form of Service Data Objects (SDO).
- ADF applications can invoke the service interfaces of the SOA Composite application.

Example of ADF and SOA Suite Integration

Imagine that you have an ADF application that provides the ability to submit purchase orders. When a user creates a new purchase order, an event defined in the related ADF business component fires and is published to the Event Delivery Network (EDN). The Mediator component in the SOA composite application detects the event and routes the payload to the business process flow, which executes the appropriate business action.

The business process flow might depend on rules. For example, you might define a rule that governs whether the flow proceeds automatically or requires human interaction. The rule can be extracted into a rules engine and then modified, as needed, when business requirements change. For example, you might create a rule that requires management approval for purchase orders that are over \$5,000. Later, if you discover that the rule creates a bottleneck, you can modify it and instead require approval for purchase orders over \$10,000, without changing the underlying service implementations or defined process.

For purchase orders that require approval, you use Human Workflow in the SOA Suite. You define a workflow that notifies the manager when approval is required and waits until approval or rejection is complete before the process continues. Because Human Workflow reuses the ADF Faces technology set, you can very easily generate pages for your application that are based on the workflow and customize them using the same set of skills required for building stand-alone ADF applications.

To monitor your business services, you use Business Activity Monitoring (BAM) capabilities. With BAM, you can monitor all purchase orders going through the process, and set thresholds to track variables that affect order fulfillment, such as when a warehouse runs out of inventory or when a vendor's web service is not responding quickly enough about pricing. You can then use data visualization components in BAM to create a dashboard that displays this information graphically. Because the data visualization components in BAM are ADF Faces components, you can in turn expose them in your own pages developed using ADF.

Recall that any web service can be exposed to ADF and used in various user interfaces. Because BPEL processes are themselves web services, they can also be exposed to ADF applications.

Benefits of Oracle ADF

- Builds on Java EE standards
- Supports exposing business services through a variety of rich user interfaces:
 - Web
 - Mobile
 - Microsoft Office
- Simplifies development and increases productivity:
 - Provides a visual and declarative development approach
 - Implements infrastructure code (ORM, security, and so on)
 - Creates reusable and maintainable code
 - Uses a metadata-driven framework
- Implements Java EE best practices and design patterns, including MVC and service facades

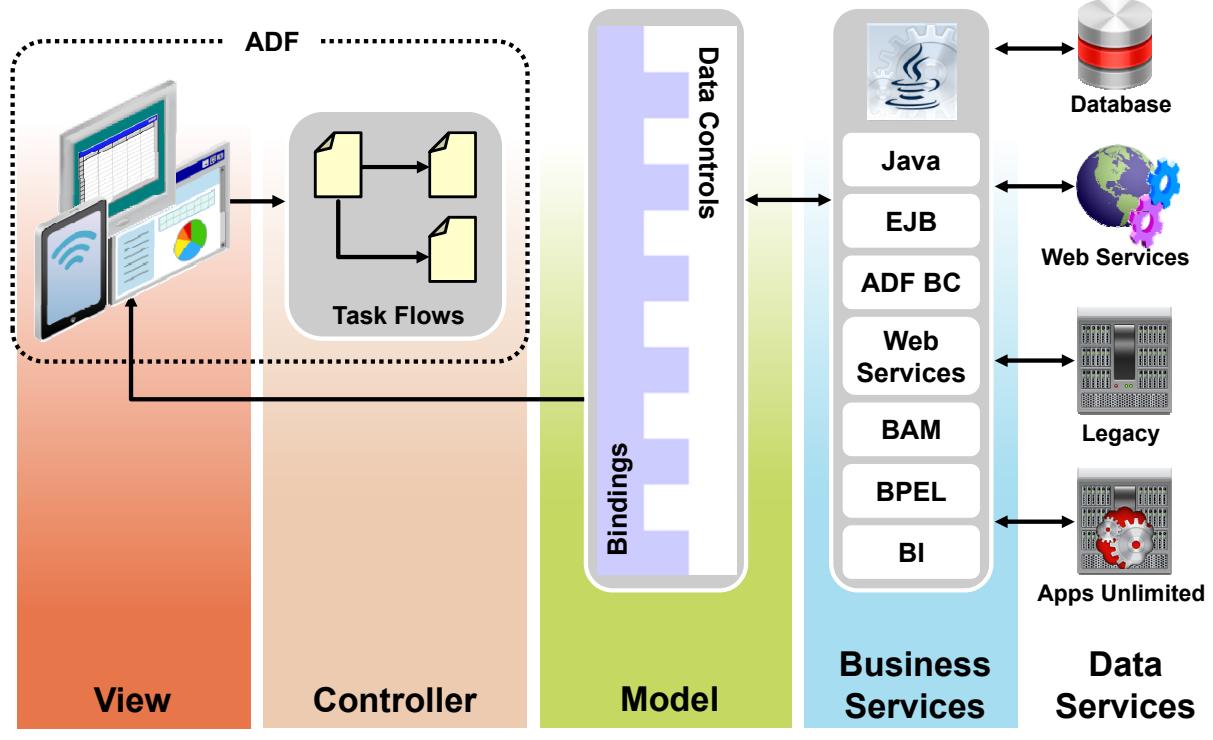


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A Common Development Framework

Oracle ADF was developed with the goal of increasing productivity and ease of use for web application developers. Oracle ADF provides a common development framework that is built on Java standards and open source technologies, such as Java EE and service-oriented architecture (SOA). With Oracle ADF, you can build reusable business services and expose them through a variety of rich user interfaces (such as web, mobile, and Microsoft Office). The focus of the framework is on providing a visual and declarative development environment for building enterprise applications. It simplifies your development efforts by taking care of common infrastructure requirements, such as object-relational mappings (ORM) and security. Because the framework is metadata-driven, objects are constructed at run time using metadata that is stored in XML files. This design makes it easier for developers to use 4GL tools (such as JDeveloper) to modify the metadata declaratively at design time and thereby alter the behavior of the framework at run time. Oracle ADF also implements Java EE best practices and design patterns, including model-view-controller (MVC) and service facades.

How Oracle ADF Implements MVC



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle ADF achieves a clean separation of business logic, application flow, and user interface by adhering to a model-view-controller (MVC) architecture. In an MVC architecture:

1. The **view** represents the user interface
2. When a user interacts with the view, the **controller** handles page navigation
3. The view then reads data from the **model** to render the appropriate user interface

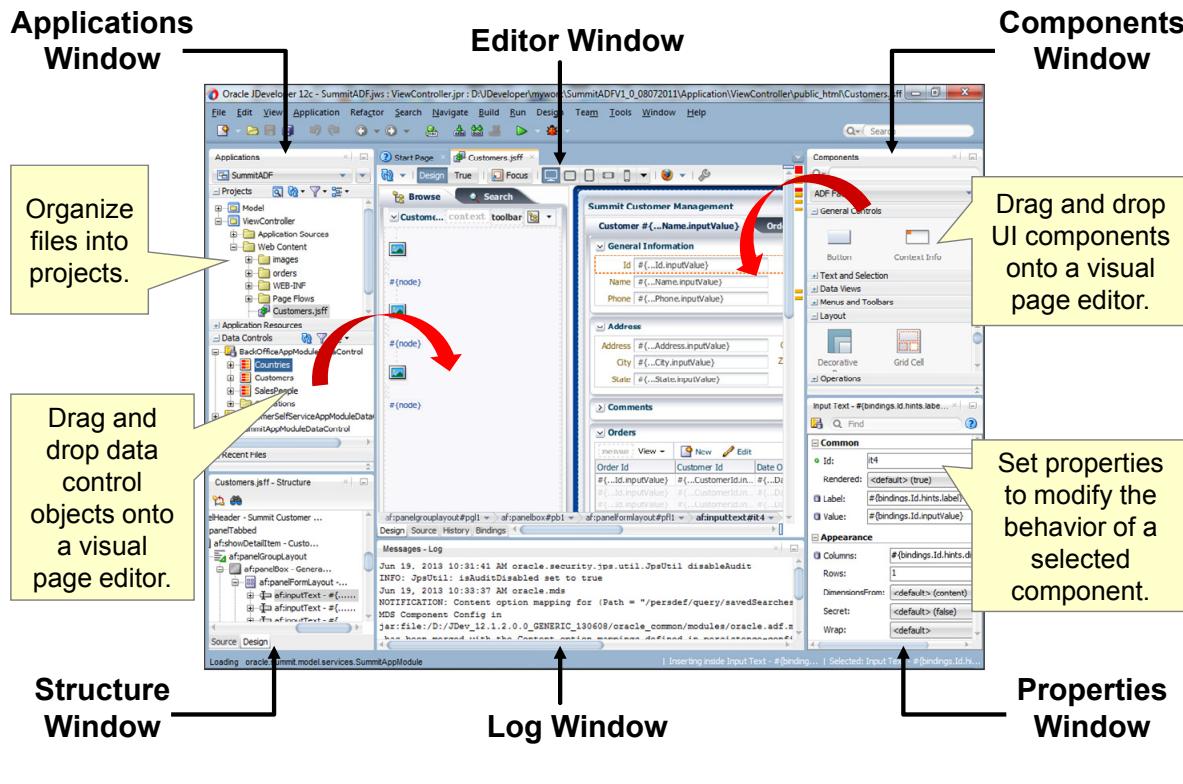
The separation of user interface components from the underlying application and business logic makes the code more flexible and easier to maintain.

The Oracle ADF implementation of MVC includes the model, view, and controller layers, plus an additional layer that handles data access and encapsulates business logic:

- The **view** layer contains the UI pages used to view or modify data. With ADF, you can build a variety of UIs, including mobile interfaces, browser-based interfaces that use JavaServer Faces (JSF) and ADF Faces for a rich client, and desktop applications that use Microsoft Excel as the front end.

- The **controller** layer determines page navigation. For an application that uses JSF and ADF Faces, you can use either JSF Controller or ADF Controller. ADF Controller extends the standard JSF Controller by providing additional functionality. Typically, ADF Controller is the best choice for ADF Faces-based web applications because it enables you to break up an application's page flow into reusable flows, and it provides support for application tasks such as validation, state management, security, and declarative transaction control.
- The **model** layer represents the data values bound to the current page. The ADF Model layer uses metadata to bind the user interface to the business services without the need to write Java code. This metadata, called ADF bindings, provides a layer of abstraction on top of the business services to enable the view and controller layers to work with different implementations of business services. So, regardless of how the business services are implemented (ADF business components, a Java class, a web service, or some other technology), the UI developer sees them as data controls that can simply be dragged and dropped onto a page to build the UI.
- The **business services** layer handles data access and encapsulates business logic. This layer is responsible for managing the object-relational mapping between Java objects and relational data. The business services can be implemented using a variety of technologies, including Java classes, Enterprise JavaBeans (EJBs), ADF Business Components (ADF BC), web services, Business Activity Monitoring (BAM) services, Business Process Execution Language (BPEL) services, and Business Intelligence (BI) services.

Introduction to Oracle JDeveloper



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An End-to-End Application Development Environment for Java EE

Oracle JDeveloper is an integrated development environment (IDE) for building Java-based enterprise applications. JDeveloper supports the full development life cycle from initial design and analysis, to coding and testing, through to deployment. JDeveloper provides a number of features that simplify development and increase developer productivity.

Visual and Declarative Environment

JDeveloper provides a combination of visual editors, property inspectors, wizards, and editing dialogs that minimize the need to write code. You can make changes to the application code declaratively (by changing property settings or using visual editors), or you can make changes directly to the code. For example, you can define a user interface visually by dragging and dropping data control objects, such as collections, attributes, or methods, onto the visual editor for a page, and JDeveloper automatically creates the bindings between the UI components on the page and your business services. Alternatively, you can click the Source tab in any editor to edit the source code directly. All areas of the IDE are synchronized so that changes in one panel are automatically reflected in other panels simultaneously.

Built-In Development Framework

JDeveloper is the development environment for Oracle ADF. This means that best practices and standards for building ADF applications are built into the tools that you use to develop the layers of your application, including data access, business services development, page navigation, rich client interfaces, data bindings, and security.

Integration with Oracle Fusion Middleware

Oracle JDeveloper is also the development environment for Oracle SOA Suite and Oracle WebCenter Suite, making integration with those technologies simpler. For example, from within JDeveloper, you can use diagrams to build composite processes with Business Process Execution Language (BPEL), and you can quickly generate rich web interfaces for Business Activity Monitoring (BAM) and Human Workflow.

Launching JDeveloper on Windows

Launch JDeveloper from the Start menu or command line:

- Select Start > All Programs > *OracleHome* > Oracle JDeveloper Studio > Oracle JDeveloper Studio
- Run from the command line:

```
<Jdev Home>\jdev\bin\jdev.exe
```

- Use optional command-line flags:
 - To specify the user directory:

```
<Jdev Home>\jdev\bin\jdev.exe -J-Dide.user.dir=<path>
```

- To specify that editor windows do not reopen:

```
<Jdev Home>\jdev\bin\jdev.exe -noreopen
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Options for Launching JDeveloper on Windows

You can launch JDeveloper in one of the following ways:

- By selecting Start > All Programs > Oracle Home > Oracle JDeveloper Studio > Oracle JDeveloper Studio (or a different location as specified in the Installer)
- By double-clicking the JDeveloper executable, `jdeveloper.exe`, in the home directory of the JDeveloper installation
- From the command line or a shortcut by running one of the following commands:
 - For 32-bit Windows:
`<JDev Home>\jdev\bin\jdevW.exe`
`<JDev Home>\jdev\bin\jdev.exe` (to display a console window for internal diagnostic information)
 - For 64-bit Windows:
`<JDev Home>\jdev\bin\jdev64W.exe`
`<JDev Home>\jdev\bin\jdev64.exe` (to display a console window for internal diagnostic information)

For information about starting JDeveloper on non-Windows operating systems, refer to *Oracle Fusion Middleware Installing Oracle JDeveloper*.

Command-Line Flags

When you launch JDeveloper from the command line, you can specify flags that provide additional control over how JDeveloper is launched.

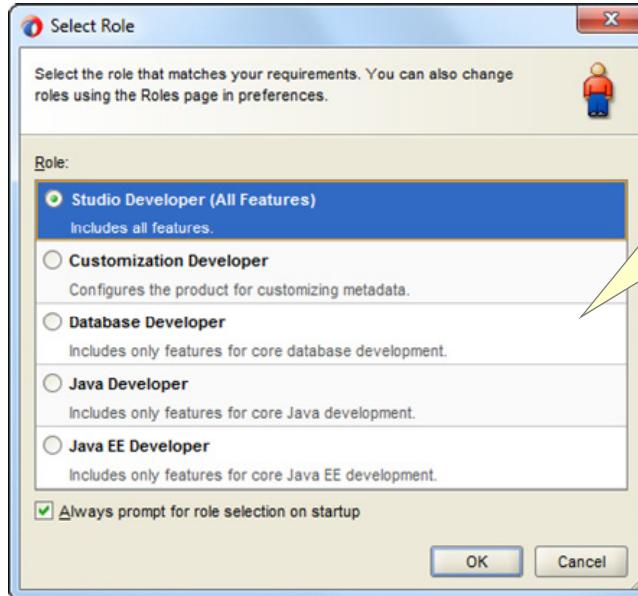
To view the available list of command line arguments, start JDeveloper with the `-help` flag:
`<jdev_home>\jdeveloper\jdeveloper -help`.

By default, JDeveloper stores user preferences, such as information about the appearance of the IDE, in a directory called `system` under the `<Jdev Home>` directory (for example, `C:\Oracle\Middleware\jdeveloper\system`). If you want to store user information in a different directory, run JDeveloper with the `-J-Dide.user.dir` flag specified. 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. Setting this flag can improve JDeveloper startup time.

Selecting a Role



Select the appropriate role to constrain the IDE to features that are applicable to your user role.

ORACLE

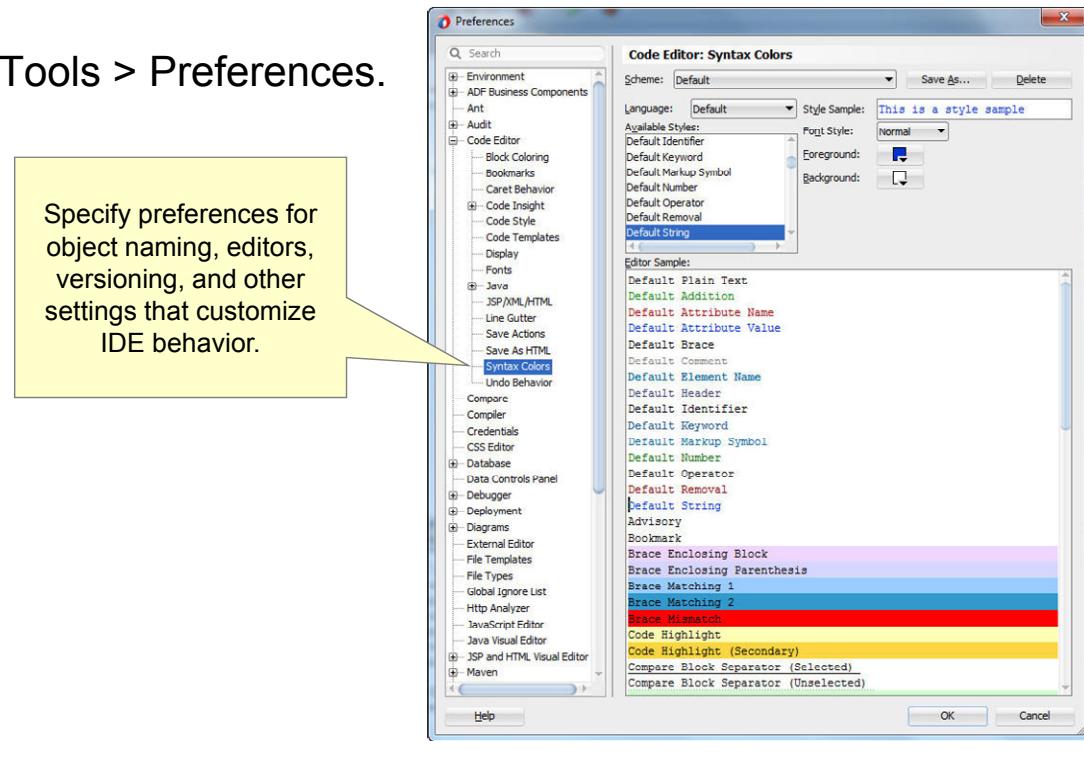
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

By default, when you launch JDeveloper, you must select a developer role. Selecting a role customizes the development environment based on the needs of specific developer roles by removing items that are not needed from the menus, preferences, new gallery, and even individual fields in dialog boxes. For example, if you select the Database Developer role, the JDeveloper IDE will include only the features that are required for core database development. For ADF application development, you select the Studio Developer role.

You can change the role later by selecting Tools > Switch Roles from the JDeveloper menu.

Setting Preferences

Select Tools > Preferences.



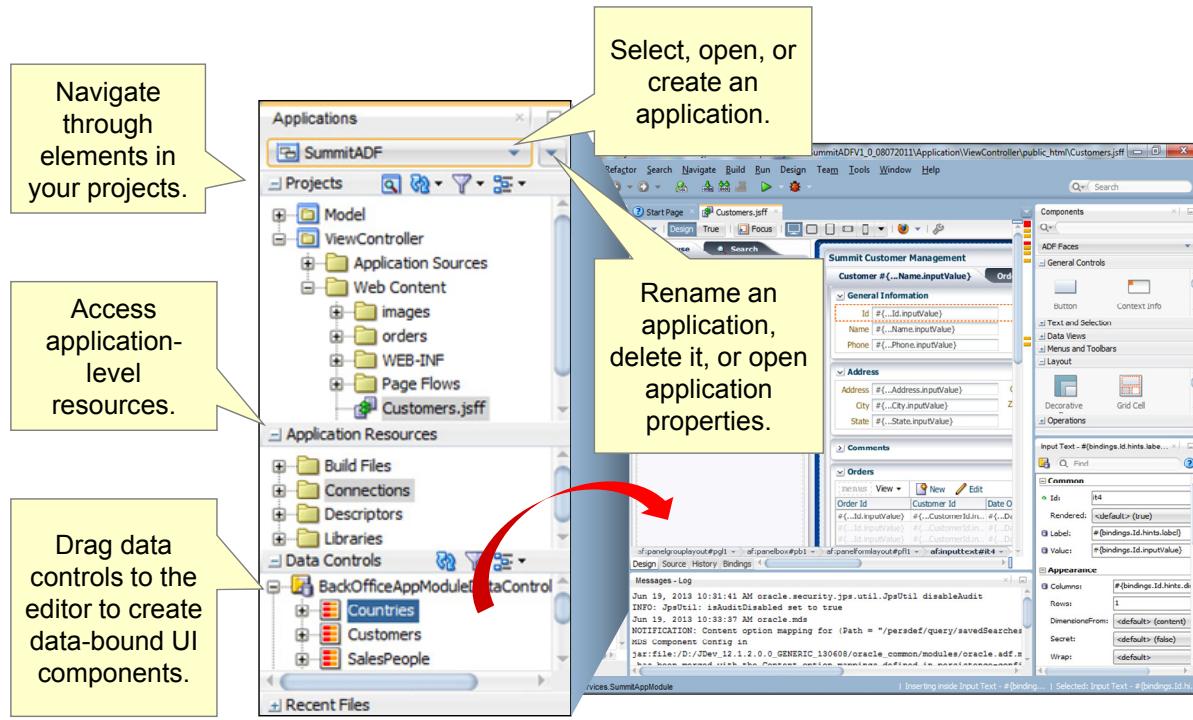
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

JDeveloper allows you to customize the IDE by setting preferences that alter the appearance and behavior of a wide variety of JDeveloper features. You can change the look and feel of the IDE, customize the general environment, and customize windowing behavior. Some examples of preferences that you can set include:

- Object naming for ADF business components. You specify the suffix that is appended by default to object names. You can override this value when you create the objects.
- Code editor look and feel, including syntax coloration, formatting, and many other options
- Web proxy settings
- Local history settings for viewing changes
- Versioning system configuration
- External editors for opening specific file types. You can set preferences to open files by default in the source code view of an editor.
- Source control settings
- Many others!

Applications Window



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

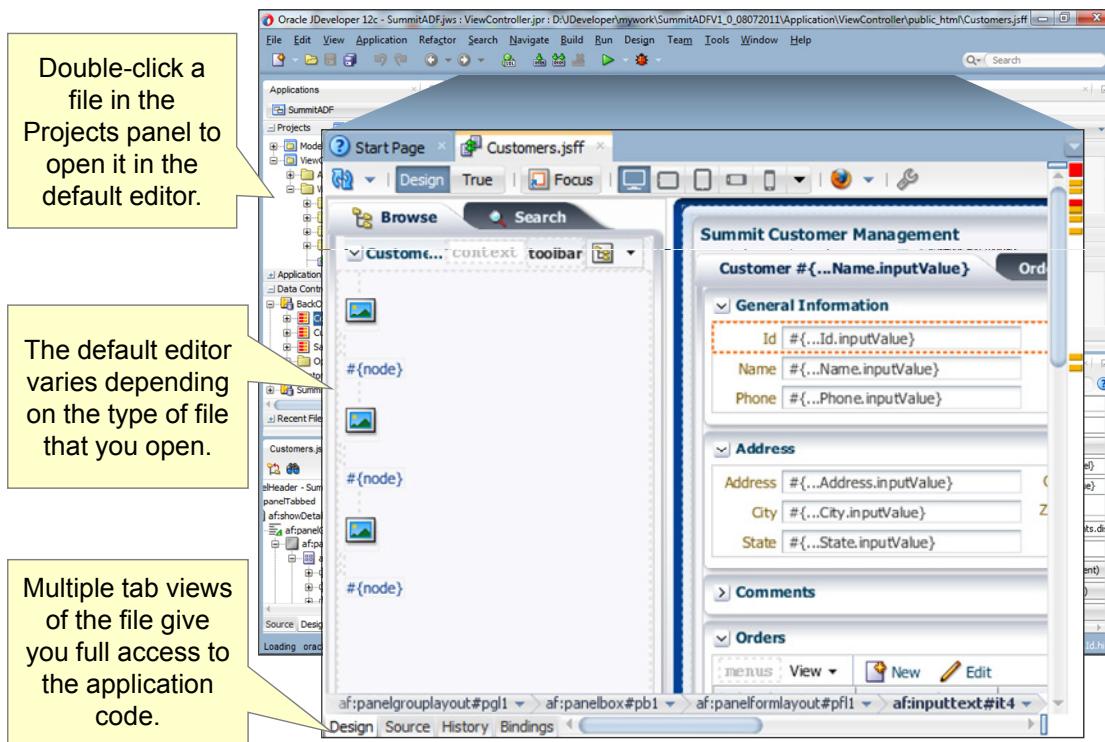
The Applications window allows you to manage the contents and associated resources of an application. To open an application or create a new one, you select an option from the drop-down list at the top of the Applications window. Click the arrow button next to the drop-down list to display the Application menu. From the Application menu, you can select options for working with the current application, including renaming the application, deploying it, deleting it, or opening application properties.

The Application Navigator contains four panels, which are expandable and collapsible.

- **Projects panel:** Shows all the projects that are part of the current application, along with a grouped, hierarchical view of the project contents. *Projects* contain the source files for your application and any configuration files that are required by the type of application that you are building. The Projects panel contains a toolbar that enables you to display project properties, refresh the view, filter, or rearrange the display. To edit content, you double-click an item in the Projects panel, and the editor for the item opens by default in the center of the main JDeveloper window.
- **Application Resources panel:** Shows application-level resources such as descriptors and connections.

- **Data Controls panel:** Provides a view of all the data objects, data collections, methods, and operations that are available for binding. From this view, you can drag and drop data controls to a page to bind back-end data services to user interface components. This panel also contains a toolbar that allows you to refresh and filter the display.
- **Recently Opened Files panel:** Provides quick access to files in the current application that have recently been opened.

JDeveloper Editors

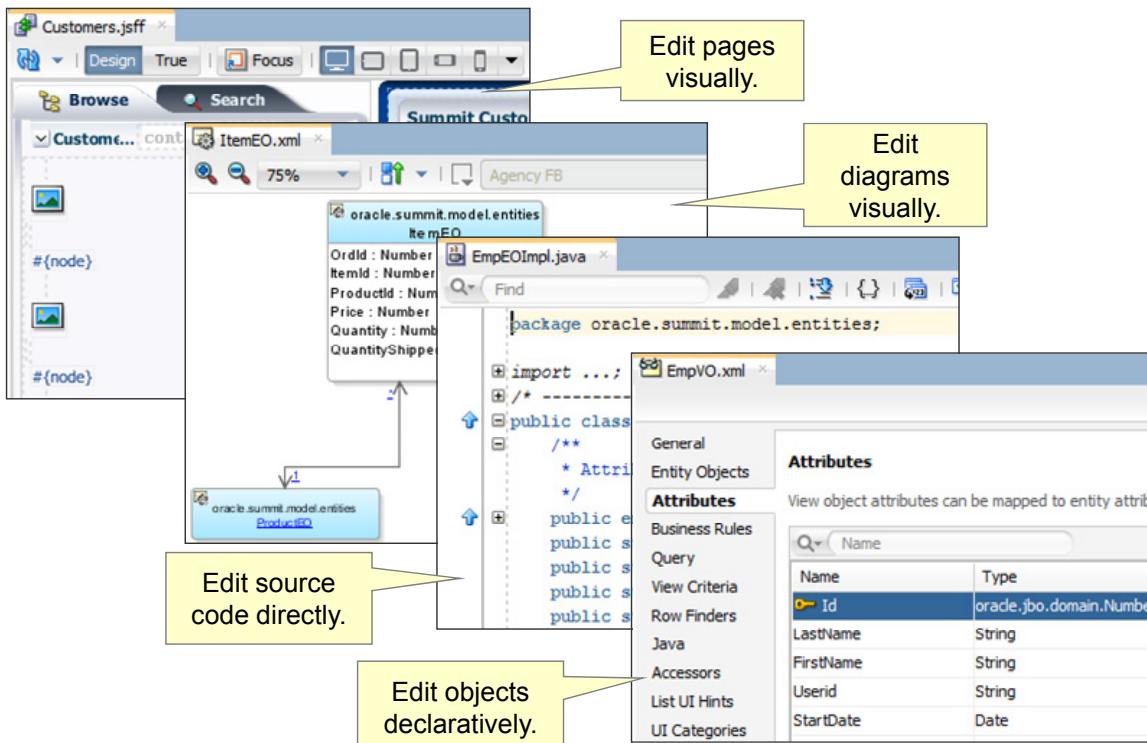


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you open a file to edit it, the file opens, by default, in the center of the JDeveloper window. The editor that appears varies depending on the type of file you open. Tabs at the bottom of the editor provide different views of the file to give you full access to modify the application code, including data bindings. For example, you can edit a file in the Design view visually, or you can click the Source tab to modify the code directly. You can click the History tab to view the history of a file and view the differences between two versions of the file.

Types of Editors



ORACLE

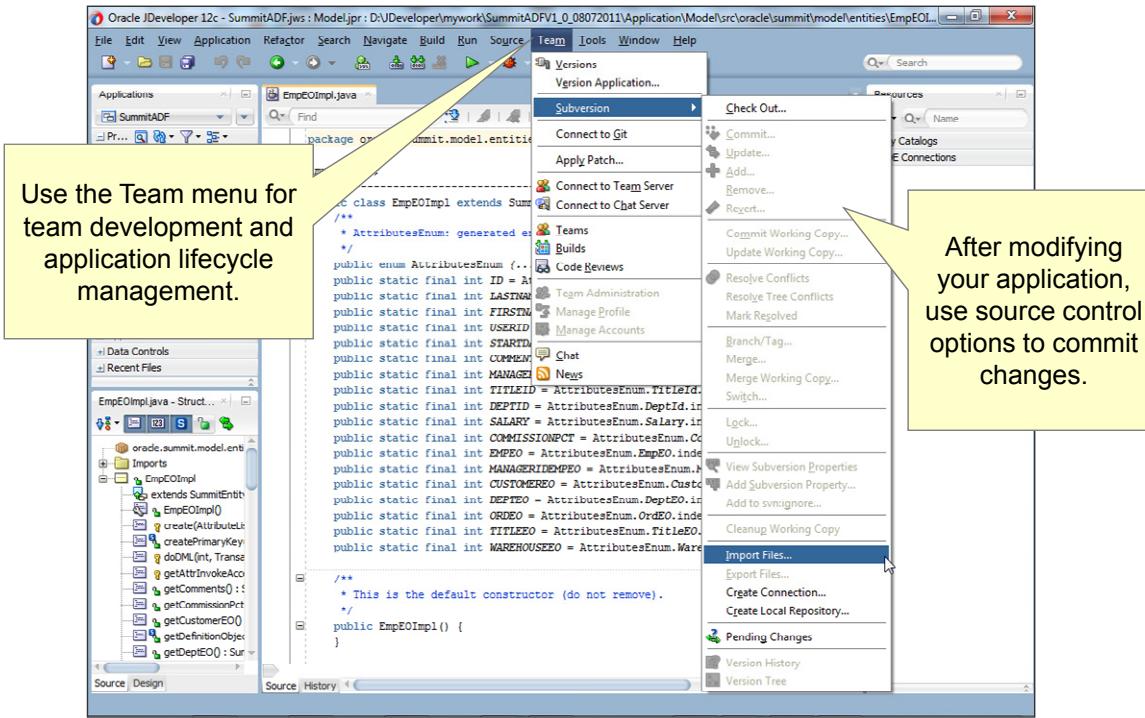
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Types of Editors in JDeveloper

JDeveloper provides a variety of editors:

- Visual editors:** Enable you to design and edit pages (such as facelets) and diagrams (such as task flows and data models) visually. The visual editors are integrated with other panels in the IDE to support the assembly of data-bound webpages by using simple drag-and-drop operations. Changes made in visual editors are automatically updated in the source code.
- Source editors:** Enable you to edit source code manually. By default, files open in a visual editor. To edit the source code, you click the Source tab. Changes that you make in the source editor are immediately reflected in the visual editor. The source editors include features like syntax coloration, code highlighting, code insight, and Quick Javadoc to make coding easier.
- Overview editors:** Enable you to view and modify features of various objects. For example, you can open data model objects and change properties of the object, such as adding validation or creating a list of values for the object.

Team Menu for Source Control Integration



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Team Development and Application Lifecycle Management (ALM) in JDeveloper

After modifying source files in your application, you can use source control integration in JDeveloper to commit your changes. JDeveloper integrates the popular version control system, Subversion, into its available feature set. You can access a number of Subversion commands directly from the JDeveloper interface, through the Team menu or through the Versioning Navigator. For example, you can view pending changes, compare versions, see code history, and branch or merge code. You can view version history in the History tab of the editors.

For teams that use other versioning systems, JDeveloper provides downloadable extensions that give you access to the following systems:

- Concurrent Version System (CVS)
- Git
- Perforce
- Rational ClearCase
- Microsoft Team System

To download these extensions, select Help > Check for Updates.

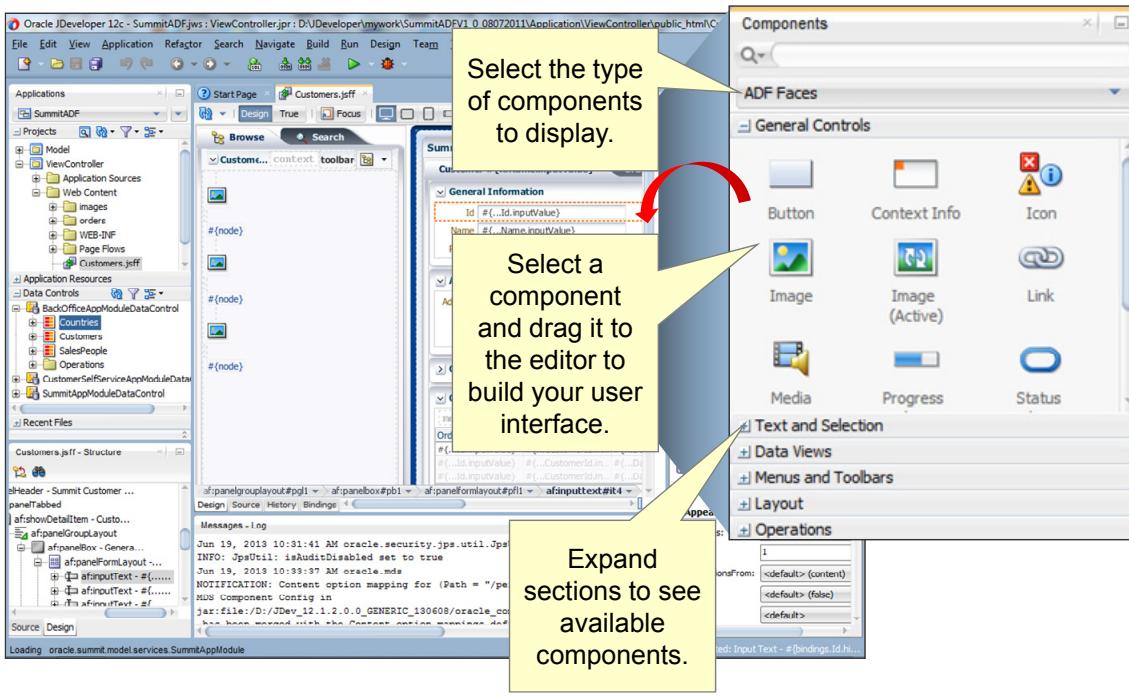
Team Productivity Center

The Team menu also provides access to Oracle Team Productivity Center (TPC) options. Oracle TPC is an Application Lifecycle Management tool that enables software development teams to collaborate and work productively together when developing applications using JDeveloper. TPC provides tools for team collaboration, management, code review, chat, and integration with other ALM repositories.

Some of the key features of TPC include:

- Versioning
- Requirements tracking
- Integration with defect tracking systems like Bugzilla

Components Window



ORACLE

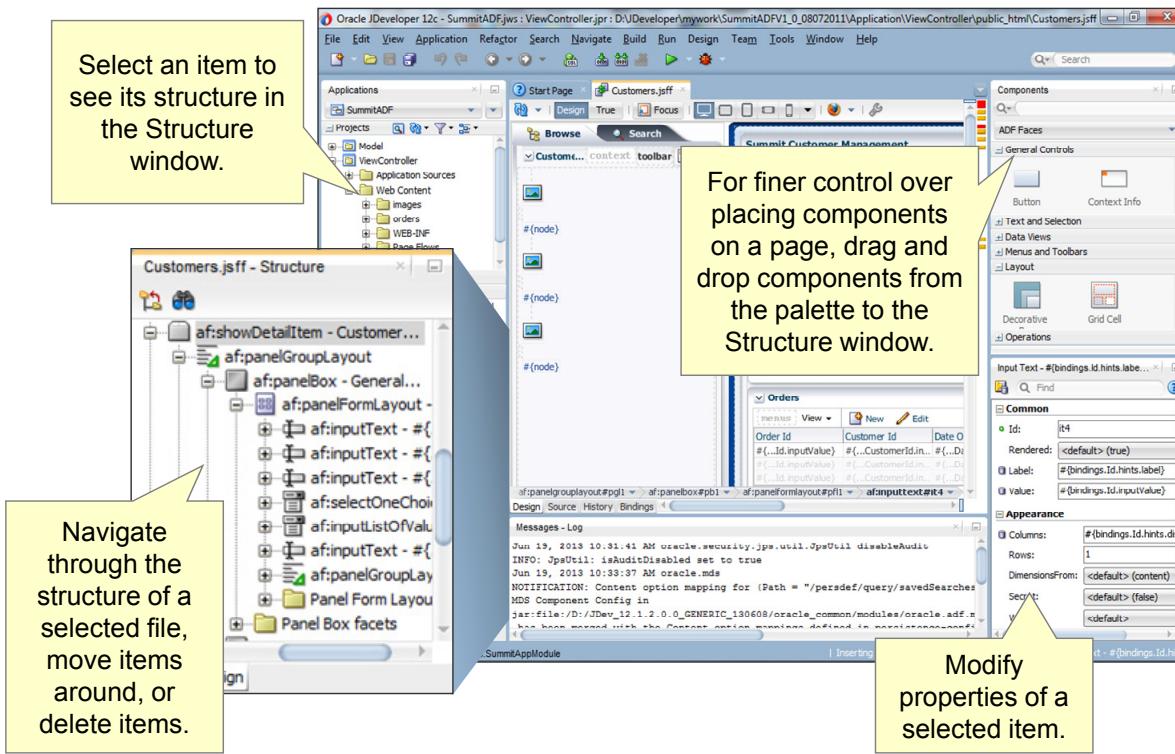
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Components window displays the components that are available in JDeveloper for building a user interface.

As you learned earlier, you can create data-bound user interface components visually by dragging and dropping data control objects, such as collections, attributes, or methods, onto the visual editor for a page. When you use this approach, JDeveloper creates the UI components and data bindings simultaneously.

In contrast, you can use the Components window to create non-data-bound UI components, such as layout components or components that you are not yet ready to bind to data. The components that are available in the window vary depending on the type of file that you have open in the editor. For example, if you are editing a JSF page, you see a list of ADF Faces components for building UI elements. You can access other types of components by selecting from the drop-down list at the top of the panel. You can also search for specific components by name.

Structure Window



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

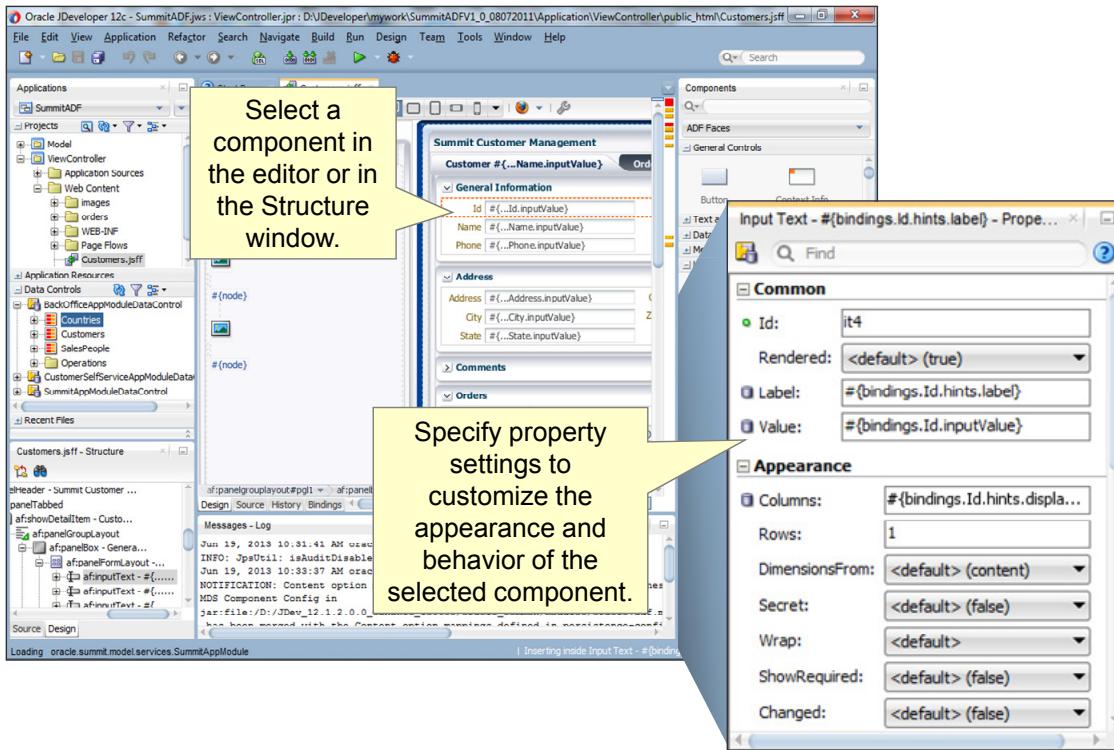
ORACLE

When you select a file in the Applications window or in an editor, the Structure window displays the elements of the selected document in a tree format, with the selected item highlighted. For example, when you use the visual editor to design a user interface, the Structure window displays the components of the UI in a hierarchical tree format. You can select a component from the Components window or the Data Controls panel, and drag it to the Structure window to place the component in exactly the correct location. You can also move components around in the tree structure by dragging and dropping them to a new location, and you can cut, copy, paste, and delete components.

The Applications window, editors, Structure window, and Properties window are fully synchronized. Selecting an item in the Structure window highlights it in the editor (and vice versa). The properties of the selected item display in the Properties window.

You can right-click the Structure window to open clones. Each clone tracks to the active view unless you right-click the window and freeze the content. Freezing the content allows you to have a separate Structure window open for each file that you edit. The content of each clone is synchronized with the view to which it is frozen. You can freeze and unfreeze the Structure window as required.

Properties Window



ORACLE

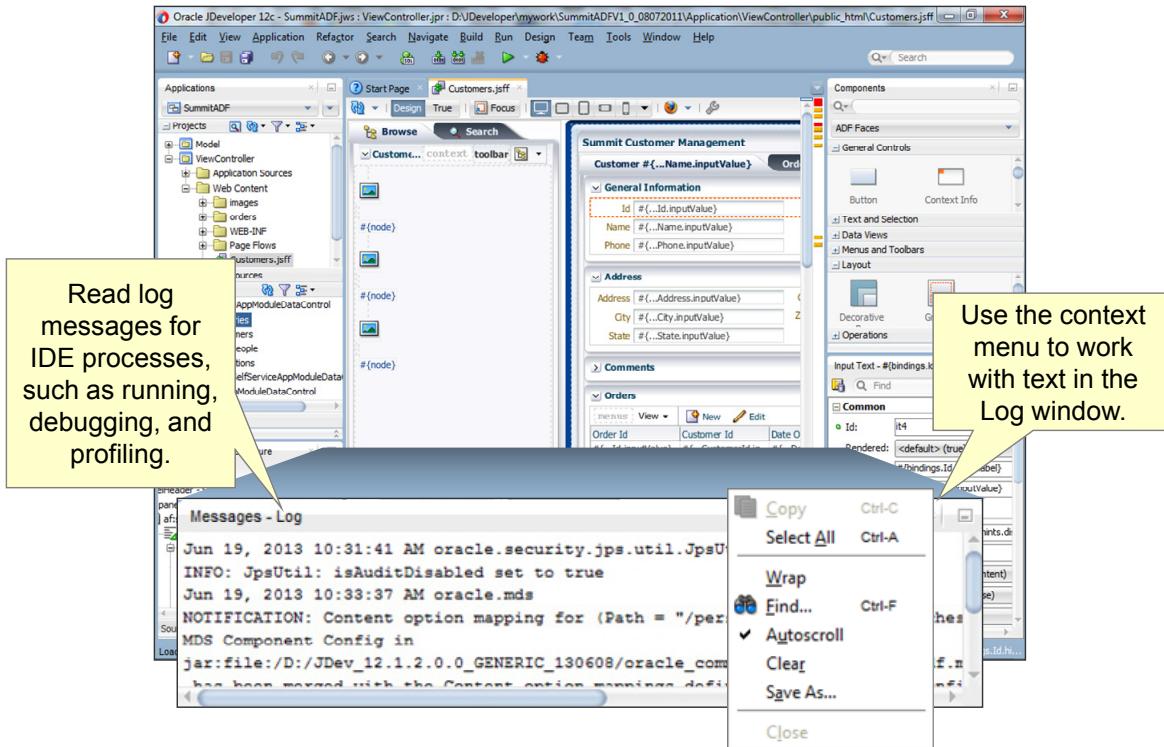
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Properties window displays all properties that are settable for a selected component. Property settings allow you to customize many aspects of the component, including its appearance and behavior. For example, if you are editing a JSF page, and select a table component, you can specify properties that control the appearance of the table (such as whether or not a horizontal grid is visible, or whether every other row is highlighted), as well as properties that control the behavior of the table (such as whether row selection, sorting, and editing are enabled). If an editor exists for a property, you can click the Property Menu icon next to the property to invoke the editor. Property help is also available by clicking the Property Menu icon.

Changes to property values are validated and applied when you tab out of a field. A green dot indicates that a property has changed from its default value.

If you are editing a component that contains dependent properties or multiple components, such as a table, you can click the Edit (pencil) icon to edit the component definition. For example, for a table component, you can click the Edit icon to add or remove columns from the table or enable sorting.

Log Window



ORACLE

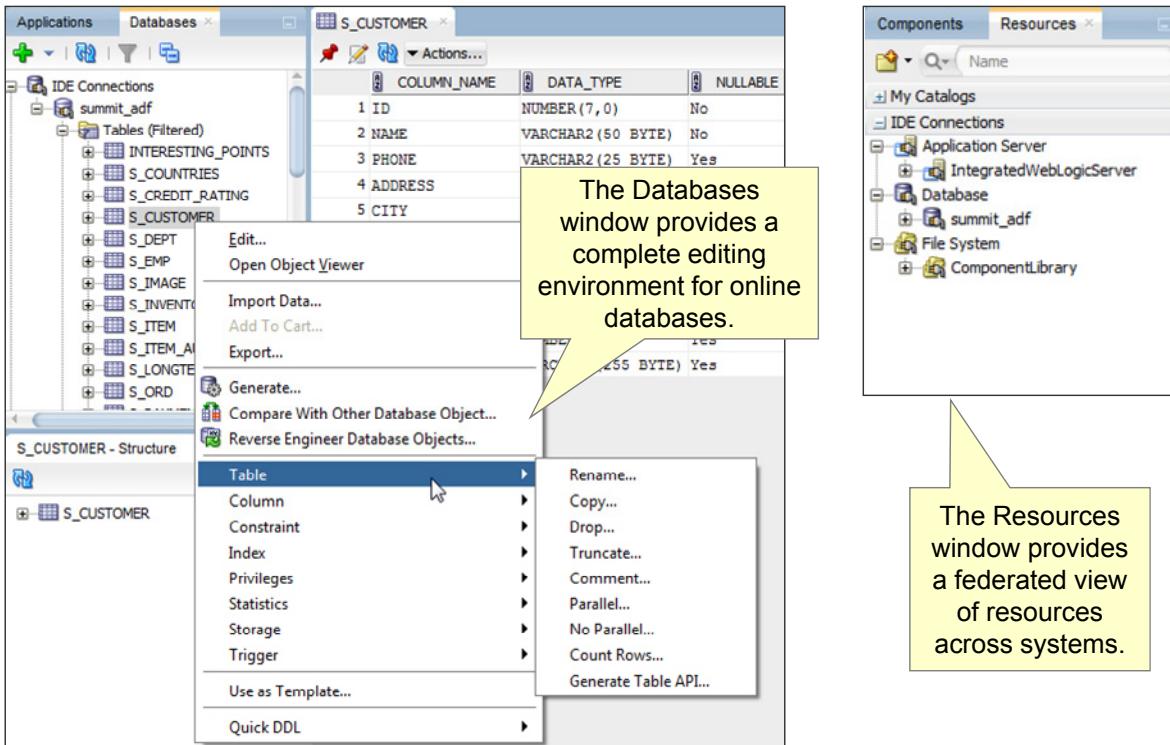
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Log window displays output from JDeveloper processes such as running, debugging, and profiling. The Log window consists of a Messages tab and, optionally, additional tabs for displaying messages related to the following categories:

- Compiler
- Apache Ant
- Debugger
- Audit
- Profiler
- Running

You can use the context menu in the Log window to select, copy, or save text, clear the text, wrap entries, find a specified string in the text, or close a tab. Other context menu options are available within the tabbed windows generated by other processes.

Other Useful Windows



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

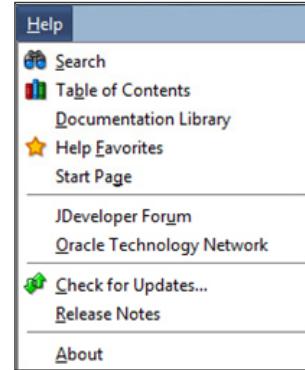
JDeveloper includes other windows that are not described in detail in this course. You can access all JDeveloper windows from the Window menu. While learning JDeveloper, you might want to explore the following windows.

- **Databases window:** The Databases window provides a complete editing environment for online databases. You can create, update, and delete database objects by using this window. The Databases window is integrated with other database-related capabilities, including SQL Worksheet, the Database Object Viewer, and the Database Modeler. You can also use the Databases window to create, import, and export database connections. To access this window, select Window > Database > Databases.
- **Resources window:** The Resources window provides a federated view of the contents of one or more otherwise unrelated repositories in a unified search-and-browse user interface. 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. Many types of connections, such as to databases, file systems, or application servers, can be created and displayed in the Resources window for use in any application. To access this window, select Window > Resources.

Obtaining Help

Help is available from within JDeveloper:

- From the Help menu:
 - Search the online help or view the table of contents
 - Access the Documentation Library
 - Go to OTN and view the JDeveloper discussion forum
- In the IDE, press F1 or click a question mark icon to access help.
- Within source editors, right-click and select Quick Javadoc, Quick TagDoc, or go to the Javadoc for a selected class.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To obtain help from within JDeveloper, click the Help menu and select one of the available options to:

- Search the online help
- View the online help table of contents
- Access the Oracle Fusion Middleware Documentation Library
- Read the JDeveloper and ADF discussion forum

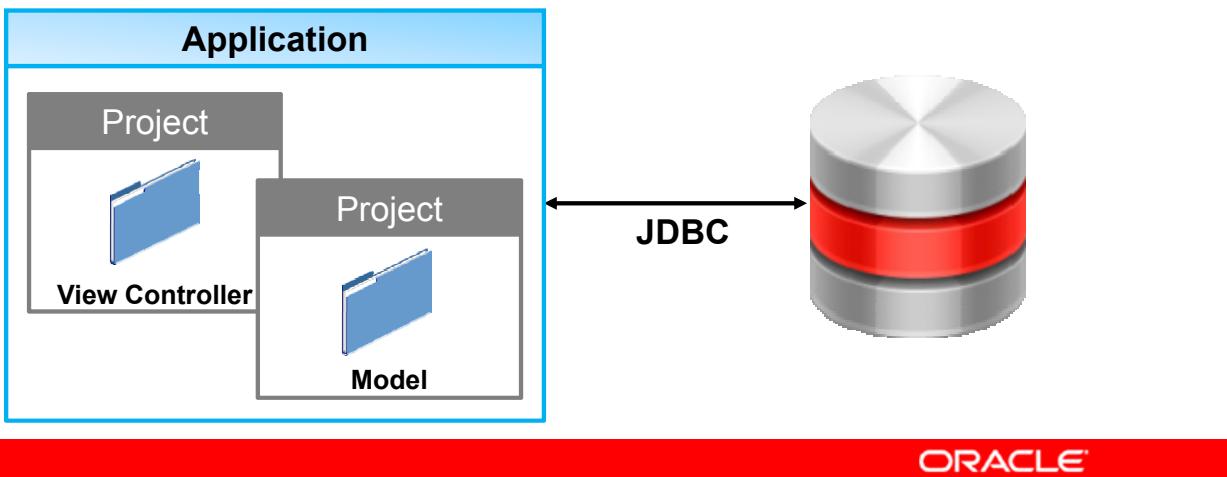
You can also access context-sensitive help directly from the IDE by pressing F1 or clicking the question mark icon.

Within the source editors, you can place your cursor within a file, right-click, and access Quick Javadoc or Quick TagDoc. You can also right-click a class name in a Java file, and go directly to the Javadoc for that class.

Getting Started in JDeveloper

To get started in JDeveloper, you must create:

- An application
- One or more projects
- A database connection (if your application requires database access)



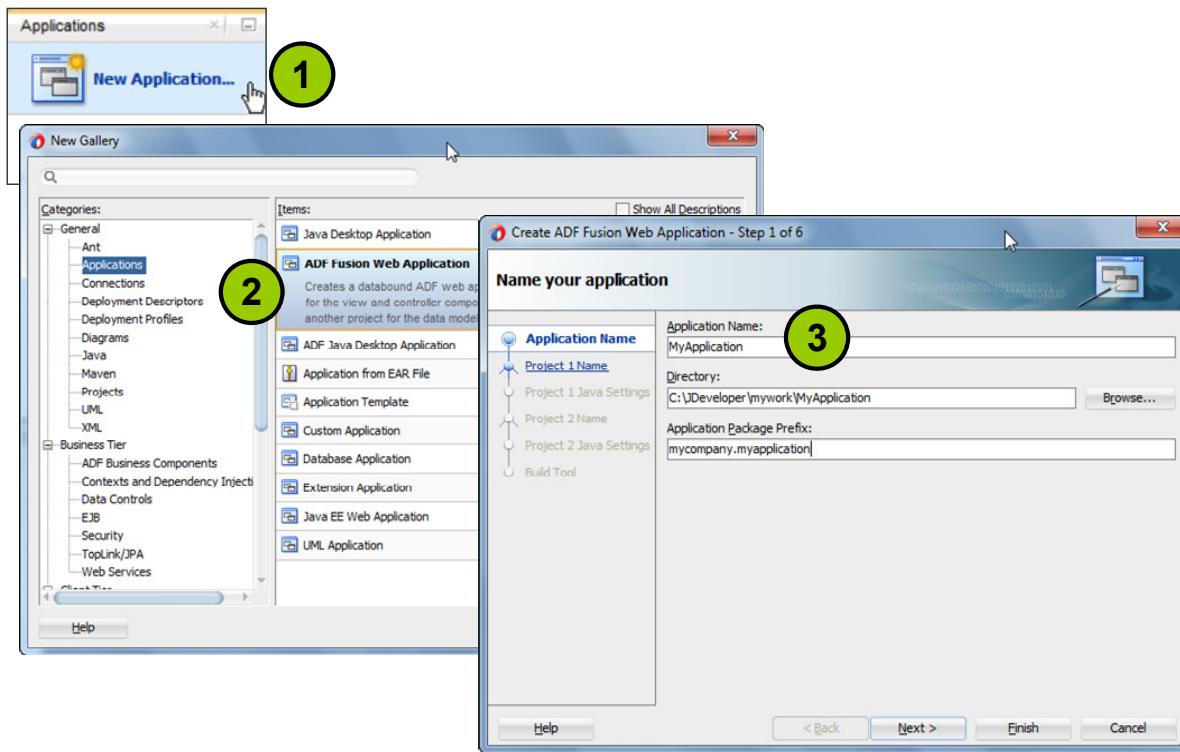
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To get started building an application in JDeveloper, you must define an application workspace and one or more projects.

JDeveloper provides application templates that you can use to organize your workspace into the correctly configured set of projects that you need. For example, if you create an application that is based on the ADF Fusion Web Application template, your application is organized into two projects: a ViewController project and a Model project. The ViewController project will contain the source and configuration files required for the view layer (the ADF Faces user interface) and the ADF controller layer (the layer that handles page navigation). The Model project will contain the business components and configuration files required for the model layer of your application.

If your application requires database access, you must also create a database connection that uses one of the supported connection methods (for example, JDBC).

Creating an Application in JDeveloper



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

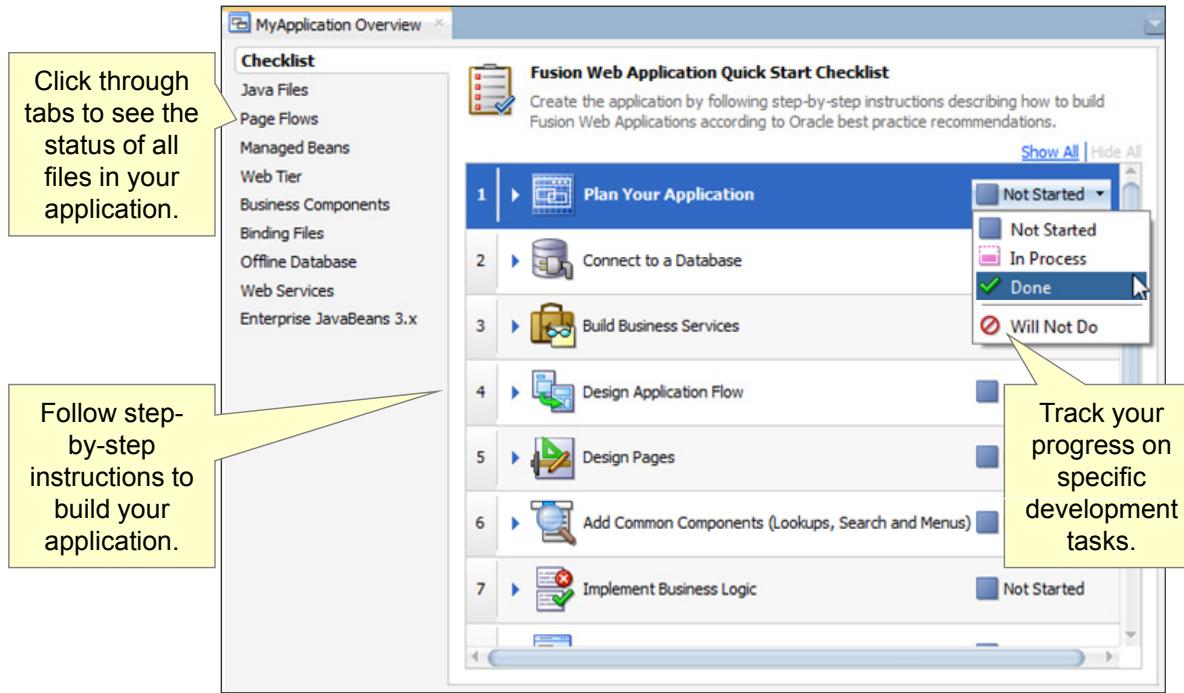
You can use the Create Application wizard in JDeveloper to specify the name and location of the application and its projects. To create an application, perform the following steps:

1. Launch the Create Application wizard. If you are building your first application or all other applications are closed, click New Application in the Applications window. Otherwise, click the drop-down list at the top of the window, and select New Application.
2. In the New Gallery, select the type of application that you are building. You can either select a template to create an application with a specific feature scope, or create a custom application that contains a single project that you can later customize to include any features. You should select an application template when you want to create a project structure that follows best practices and has the appropriate combination of features already specified. To create a web browser-based application that uses ADF for the model, view, and controller layers, select ADF Fusion Web Application. Click OK.

3. In the remaining wizard pages, specify the name, location, package naming, build tools, and other settings for the application and its projects. To ensure that your component names do not clash with reusable components from other organizations, choose package names that begin with your organization's name or web domain name. Make sure that package names also include a prefix that represents your application. If you follow this convention, components for your applications will reside in packages with names like `com.yourcompany.yourapp` (or `yourcompany.yourapp`) and sub-packages of these. For example, the Summit ADF sample application uses the package names that begin with `oracle.summit`. For more information about naming conventions, see the ADF coding standards defined by the ADF Enterprise Methodology Group (EMG) at <https://sites.google.com/site/oracleemg/adf/files>.

When you create an ADF Fusion Web Application, the wizard walks you through specifying the name, location, and Java settings for the Model and ViewController projects. The wizard also describes the technology features that will be available for each project. For example, when you specify settings for the ViewController project, you learn that your project can use the following technology features: ADF Faces, ADF Page Flow, Java, JavaServer Faces (JSF), JSP and servlets, Trinidad, and XML. The list of available features is based on the template that you select. You can add or remove features from the project later by editing the project properties.

Application Overview and Checklist



ORACLE®

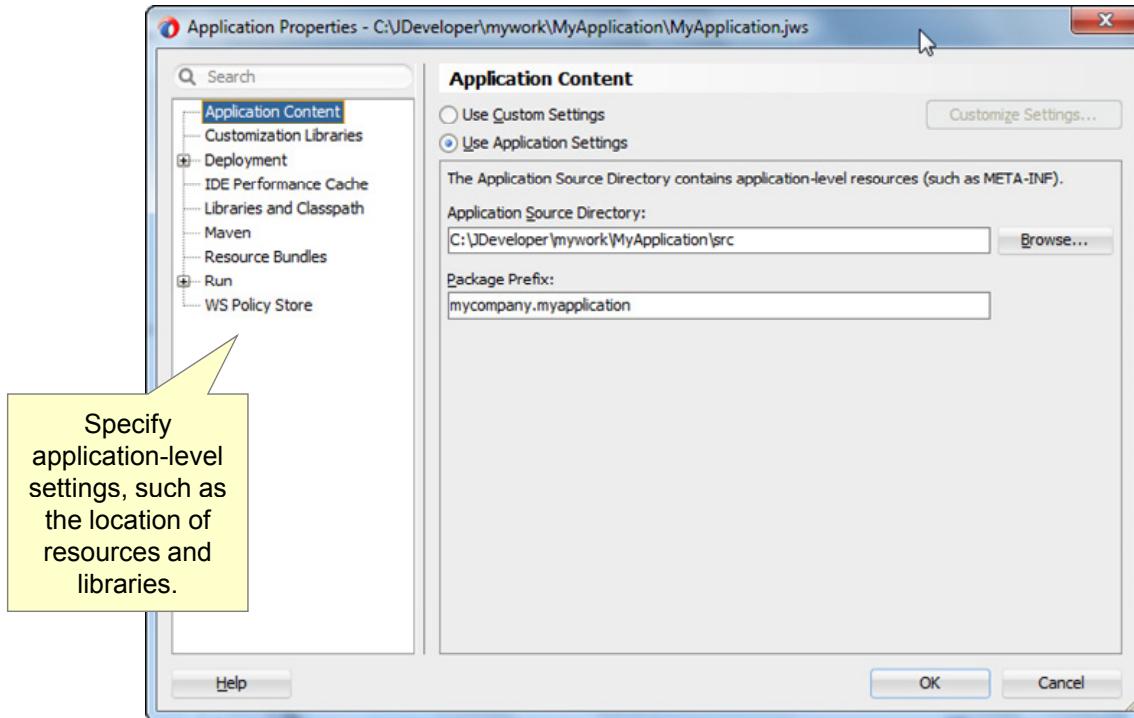
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you first create an application, the application overview page appears in the editor area of JDeveloper. For Fusion web applications, the application overview page includes a checklist that leads you through the steps required to build the application. Using the checklist is optional, but by using the checklist, you can more easily build an application that follows Oracle best practices. As you work through the checklist, you can mark your progress by selecting In Process, Done, or Will Not Do.

From the application overview page, you can also select tabs to view the status of all the files in your application. Status icons indicate whether a file has an error or warning, is incomplete, requires further inspection, or has not been checked. You can open files directly from the application overview page, and edit them in the default editor. You can also create or delete files. For example, in the Java Files panel, you can click the New button and create a Java class or Java interface.

In the Application Navigator, select Application > Show Overview to display the application overview page.

Editing Application Properties



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

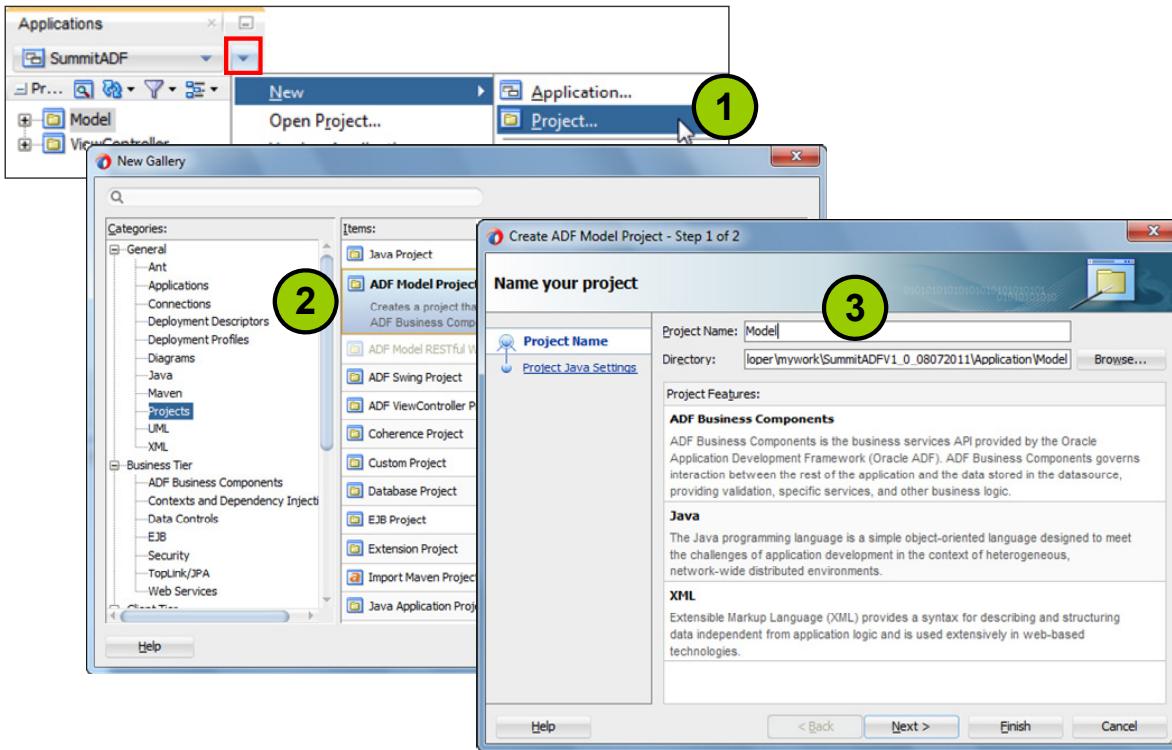
You can edit application properties by selecting Application > Application Properties from the main JDeveloper menu (or by selecting Application Properties from the Application menu). Application properties include settings that are valid at the application level, such as:

- The location of application-level resources and libraries
- Deployment properties
- Registered resource bundles

The application properties also include runtime properties, such as:

- The application name and server binding
- The location for storing runtime customizations and metadata
- The default hot deploy policy

Creating a Project in JDeveloper



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You have already learned how to create an application that includes projects. You know that projects are used to organize the files in your application, and that you can use templates to create applications that have the correct project structure and technology features for the type of application that you are building.

After defining an application, you can create new projects and add them to the application. For example, you might want to create a new project to hold unit tests, or to hold another functional area of the application, such as SOA composites.

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.

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

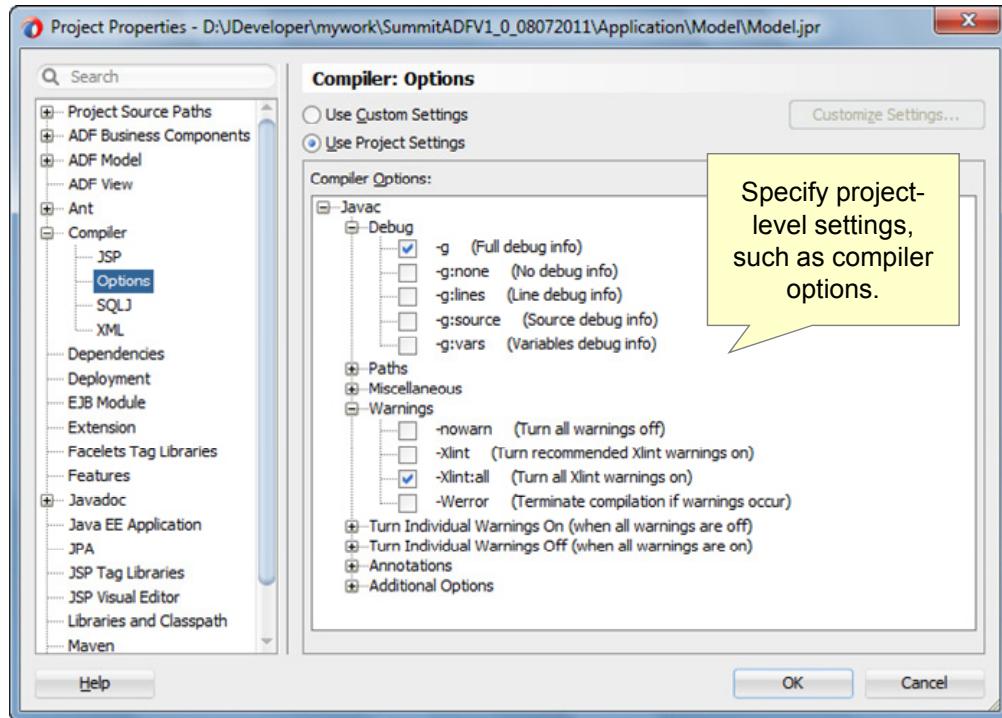
To create a project, perform the following steps:

1. With the application open, click the Application Menu icon, and select New > Project (or select File > New > Project from the main menu).
2. In the New Gallery, select the type of project to create, and then click OK.
3. In the Create Project wizard, enter a project name and specify where you want to store the project files. Depending on the type of project, you might also need to select the technologies to use, specify Java or EJB settings, or specify other types of project properties. For example, when you select an ADF Model project, the project automatically includes the technology features that are required for the project: ADF Business Components, Java, and XML. However, when you select Custom Project, you are required to select from a list of available technology features.

Setting Default Project Properties

You can set default project properties that are automatically applied to any new projects that you create. For example, you can set defaults for project source paths. To set default properties for new projects, from the JDeveloper menu, select Application > Default Project Properties, and specify settings. You can override these settings by editing the project properties.

Editing Project Properties



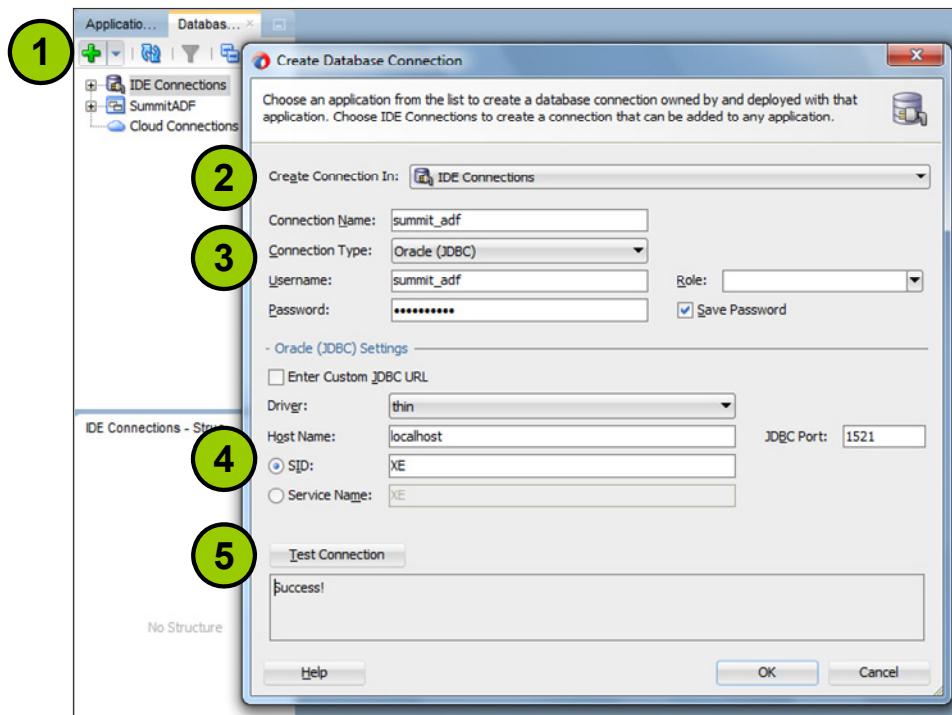
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can edit properties for an existing project by double-clicking the project in the Applications window (or by right-clicking the project and selecting Project Properties). Project properties specify a variety of project-specific settings, such as:

- The location of source files
- The behavior of tools like Ant and various compilers
- Project dependencies
- The database connection to use for development
- Deployment properties
- The list of features that the project is configured to use
- The classpath for libraries
- Resource bundle options
- Run configurations for the project

Creating a Database Connection in JDeveloper



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can create a database connection that is available to all the projects in a specific application, or you can create an IDE connection that is available to all applications and projects. IDE connections can be added to a resource catalog and shared with other developers on the team.

To add a database connection, perform the following steps:

1. From the Databases window (Window > Database > Databases), click the New Connection icon to launch the Create Database Connection wizard.
2. In the Create Connection In field, either select an application that will own the connection, or select IDE Connections to create a connection that can be added to any application.
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.
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. 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 Databases window.

Underlying Files

IDE connections and application-specific connections are stored in separate files:

- All IDE connections are stored in the `connections.xml` file in the `system12.1.2.0\o.jdeveloper.rescat2.model\connections` subdirectory of the JDeveloper user directory.
- Application-specific connections are stored in the `connections.xml` file in the `.adf\META-INF` subdirectory of the application itself. By default, applications are stored in `\mywork\<Application Name>` under the root directory in which JDeveloper is installed, although you can choose to save them elsewhere.

Summary

In this lesson, you should have learned how to:

- Describe the Oracle Fusion Middleware architecture
- Explain how ADF fits into the Oracle Fusion Middleware architecture
- Describe the ADF technology stack and how it implements the MVC design pattern
- List benefits that JDeveloper provides for application development
- Describe the main windows and editors available in the JDeveloper IDE
- Create applications, projects, and connections in JDeveloper



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 2 Overview: Using JDeveloper

This practice covers the following topics:

- Creating a JDeveloper application, project, and database connection
- Loading the course application and running it
- Looking at projects, business components, pages, and task flows, and using the “Search in Files” feature



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

Building a Business Model with ADF Business Components

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Identify the types of service components and describe how they are used in a web application
- Explain how entity objects relate to database tables and maintain persistence in an application
- Create entity objects and associations from database tables
- Create view objects and view links
- Explain the role of application modules
- Create an application module and test the application
- Refactor business components

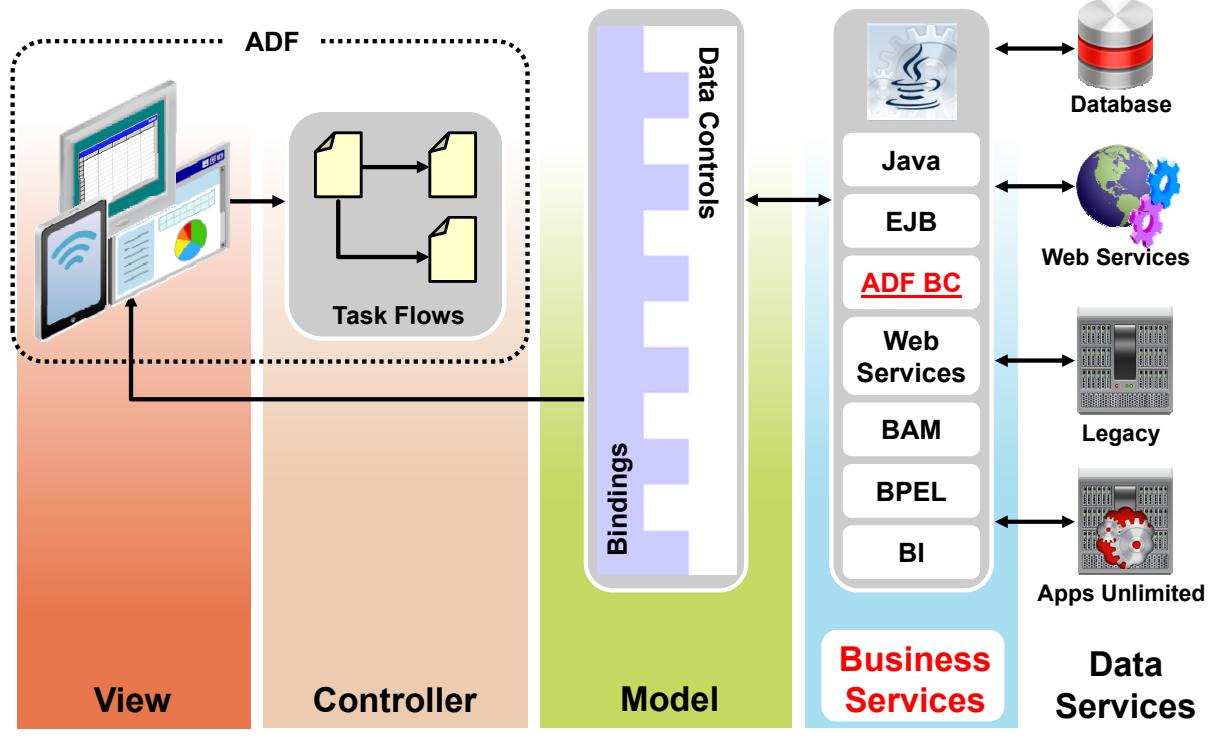


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Lesson Aim

This lesson provides an introduction to ADF Business Components (ADF BC) and building the model/business services side of your application. 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 view objects based on them. You then learn how to expose ADF BC data services to an application through an application module. You create ADF business components and test them with the Oracle ADF Model Tester.

Building the Business Services Layer



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the introductory lesson, you learned that the ADF architecture includes a business services layer that handles data access, encapsulates business logic, and manages the object-relational mapping between Java objects and relational data. You also learned that business services can be implemented by using a variety of technologies, including Java classes, Enterprise JavaBeans (EJBs), ADF Business Components, web services, and many other types of services.

In this lesson, you learn how to create a business services layer that uses ADF Business Components, and you learn how to generate data controls that you can use later to expose data in a user interface.

ADF Business Components

The ADF Business Components (ADF BC) framework:

- Handles data access and business logic execution:
 - Manages object relational mapping between application objects and the database
 - Manages validation of data and business logic
- Includes built-in, customizable operations for manipulating data, including select, insert, update, delete, commit, and rollback
- Enables 4GL-like development
 - Wizard-based, visual development
 - Driven by metadata
- Enables declarative business rule development



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

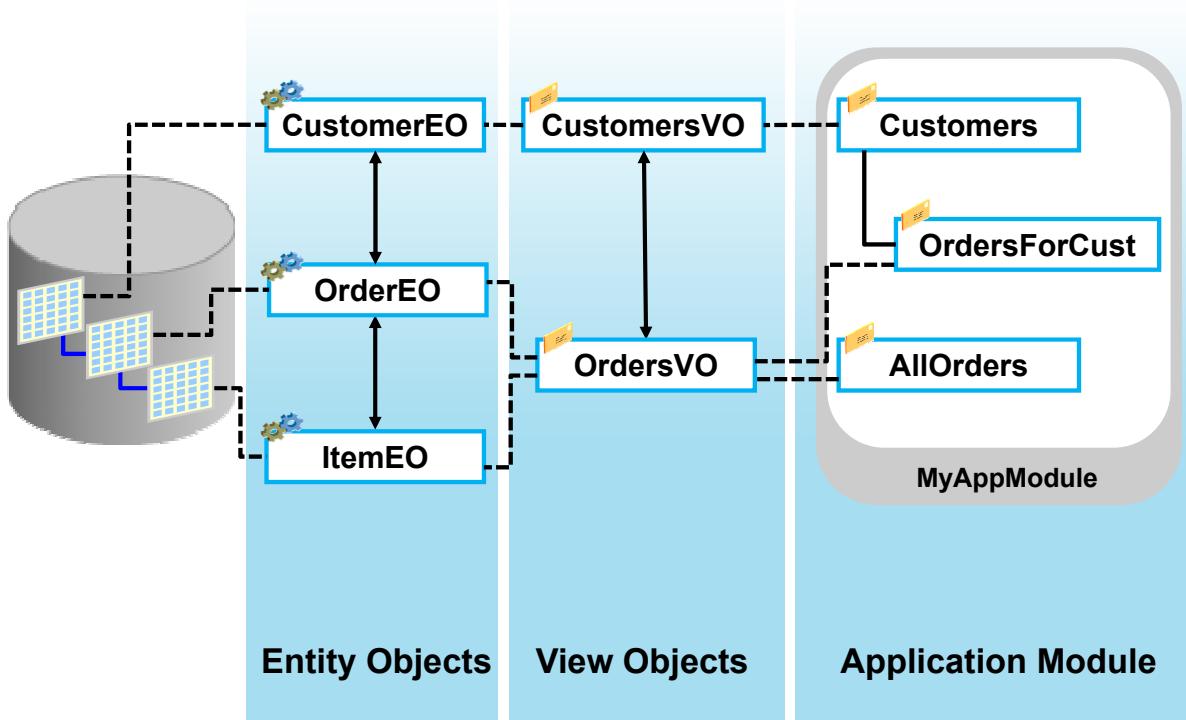
The Oracle ADF Business Components (ADF BC) framework provides building blocks for creating the business services layer of your application. ADF BC handles all interaction between your business service and the data stored in the data source. ADF BC provides the object-relational mapping between your business objects and the database, it manages validation of data and business logic, and it uses SQL-based data views to interact with the database.

With ADF BC, you can:

- Write and enforce business application logic in a centralized way
- Design and test business logic in components that automatically integrate with databases
- Reuse business logic in multiple applications and application tasks
- Maintain and modify business functionality in layers, without requiring modification of the delivered application

The core functionality of ADF BC is implemented in Java. However, the specifics of how the framework behaves for a specific application task are defined in XML. The various wizards and property pages in JDeveloper allow you to easily manage the XML metadata files that drive the behavior of the components at run time. You can optionally expose framework Java code to add functionality to business components.

Types of ADF Business Components



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you use ADF BC to build the business services layer of your application, you work with three main types of components: entity objects, view objects, and application modules. These components provide a layer of abstraction between the application's view of the data and the physical structure of the data source.

- **Entity objects:** Represent objects in the data source (usually tables, views, or synonyms in a database). Entity objects are the mechanism for persisting data to the database. Entity objects serve as an application cache for the data, they encapsulate business logic to enforce business rules, and they provide the object-relational mapping between the application and the database. At run time, an entity object represents a single row of data. **Associations** (indicated by black arrows in the Entity Objects section of the diagram) represent relationships between the objects (for example, foreign key relationships).
- **View objects:** Define an application-specific view of the data (usually based on SQL queries). View objects are the mechanism for reading data from the database. When end users modify data in the user interface, the view objects collaborate with entity objects to consistently validate and save the changes. At run time, a view object represents a collection of rows. **View links** (indicated by black arrows in the View Objects section of the diagram) define relationships (such as master-detail) between view object result sets.

- **Application modules:** Are containers for the view object instances and links related to a specific business task. An application module defines the data model and procedures and functions (called service methods) that UI clients use to work with application data.

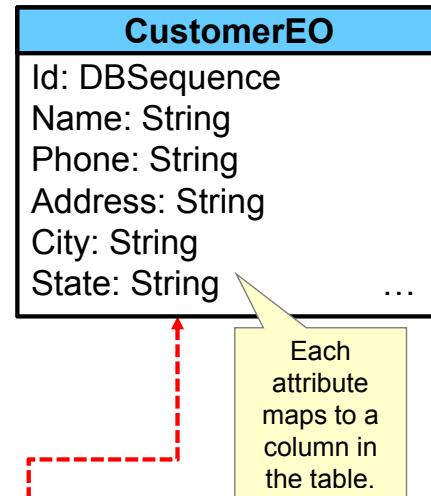
In the example in the slide, the database contains the following tables: Customer, Ord, and Item. Within the business services layer, the entity objects, CustomerEO, OrderEO, and ItemEO, represent each of the tables. The view objects, CustomersVO and OrdersVO, provide a view of the data that is based on the requirements of the user interface (with any extraneous columns removed). Notice that the OrdersVO view object is based on two different entity objects. Because view objects define an application-specific view of the data, a view object is often based on more than one entity object. In fact, you can use the full power of the SQL language to create view objects that join, filter, sort, and aggregate data into exactly the shape required by the end user.

The application module, AppModule, exposes only the view object instances that are required by the specific use case: managing customers and orders through a back-office application.

Next, let's learn more about the individual components.

Entity Objects (EOs)

- Handle data persistence
- Map to a single row in a table definition or other data source
- Contain attributes that represent columns
- Encapsulate validation logic and can include custom methods
- Are based on a physical representation of the data



ID	NAME	PHONE	ADDRESS	CITY	STATE	...
---	---	-----	-----	-----	-----	---
201	Unisports	55-2066101	72 Via Bahia	Sao Paolo	-	...
202	Simms Athletics	81-20101	Takashi	Osaka	-	...
203	Delhi Sports	91-10351	Chanaky	New Delhi	-	...

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An entity object is an ADF business component that represents a data source (usually a database table or view). Entity objects simplify modifying data by handling all data manipulation language (DML) operations for you. Entity objects handle database caching, so that changes to data are cached in the entity object before being committed to the database.

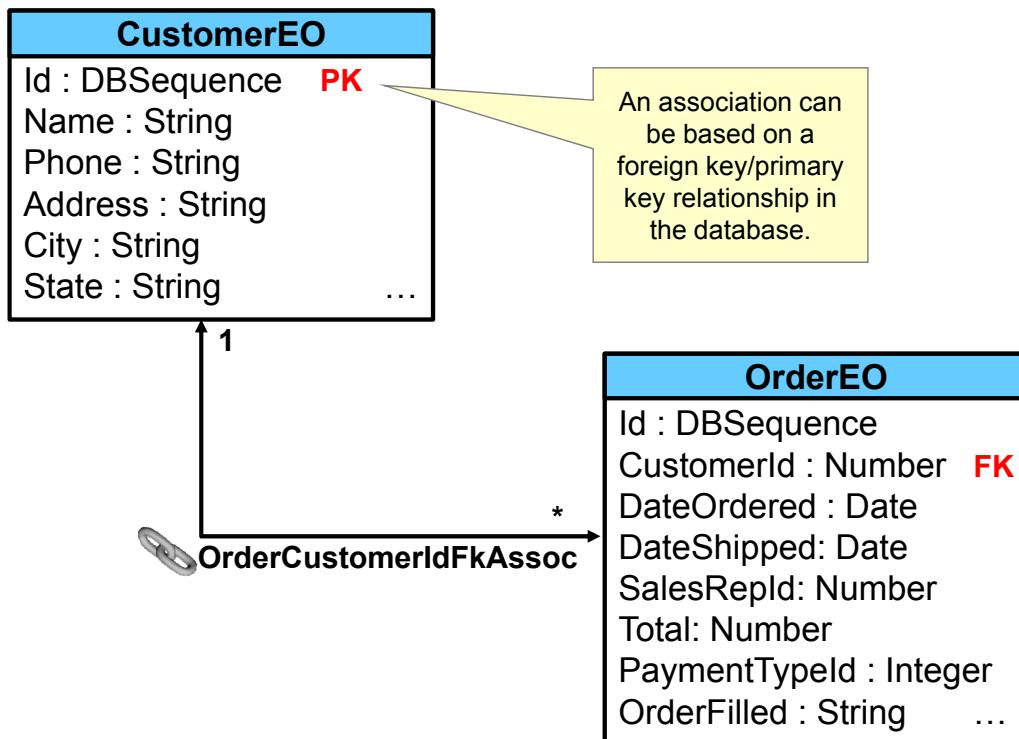
Entity objects contain attributes that map to each column in a table definition (or other data source). The attribute definitions have properties that are derived from the column definitions, such as the data type, column constraints, precision, and scale. Entity objects also handle business rules and validation and can contain custom business methods. By default, entity objects are updateable, which means that you can use them to update the database. However, you can choose to make them non-updateable.

In the example, CustomerEO is an entity object that represents the CUSTOMER table in the database. CustomerEO has attributes, such as Id, Name, Phone, and Address that map to the corresponding columns of the CUSTOMER table.

Because entity objects represent the business domain layer of your application, the design of entity objects should be based on the physical representation of your data, not on how the data will be presented in your application.

Entity object definitions are stored as XML files.

Associations



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Associations are ADF business components that represent relationships between entity objects. Associations are created automatically when you create entity objects that are based on database tables that have existing foreign-key relationships. Alternatively, you can create associations manually if such relationships do not explicitly exist in the database.

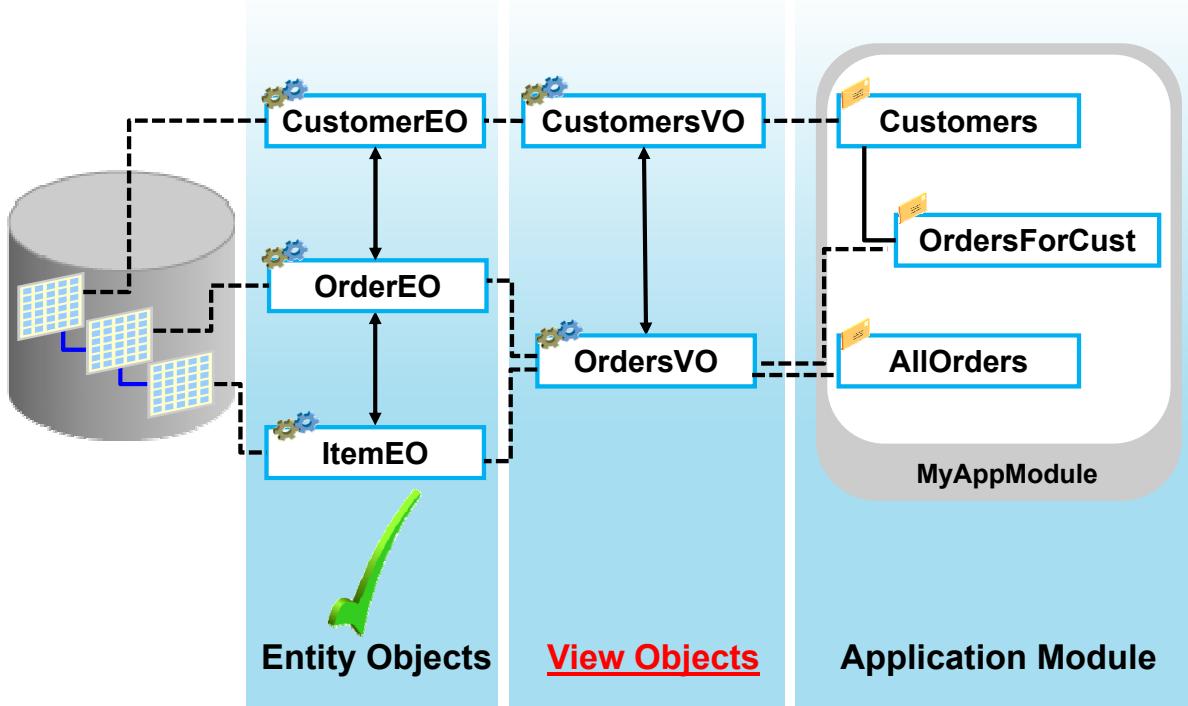
Associations allow you to preserve master-detail relationships and cardinality between source and destination objects. Associations can represent relationships that range from simple one-to-many relationships based on foreign keys to complex many-to-many relationships.

Associations help you create a more comprehensive representation of your business objects. For example, associations allow you to “walk” between entity objects, using the `findByPrimaryKey` 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 can also perform automatic lookups of reference information when foreign key values change.

The example in the slide shows an association, called `OrderCustomerIdFkAssoc`, that is defined between the `CustomerEO` and `OrderEO` entity objects. JDeveloper created this association automatically based on the foreign key/primary key relationship defined between the source tables (`Customer` and `Ord`) in the database. The association preserves the one-to-many relationship between the `Customer` and `Ord` tables.

JDeveloper generates one XML file for each association.

Types of ADF Business Components: Entity Objects Summary



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Now you understand what entity objects are and how they are related through associations. You know that entity objects are responsible for handling database persistence (issuing insert, update, and delete statements) and caching. Next, let's learn about view objects.

View Objects (VOs)

- Provide a view of the data that is shaped for the user interface
- Are used for joining, filtering, sorting, and aggregating
- Can access the entity object cache to apply changes to the data
- Can be constructed from an entity object, a SQL query, a static list, or programmatically

CustomersVO
Id : DBSequence
Name : String
Phone : String
Address : String
City : String
State : String
...

The structure of the view object is determined by the design of the UI.

Contact Information

* Name: Unisports

Phone: 55-2066101

Address: 72 Via Bahia

City: Sao Paulo

State:

ORACLE

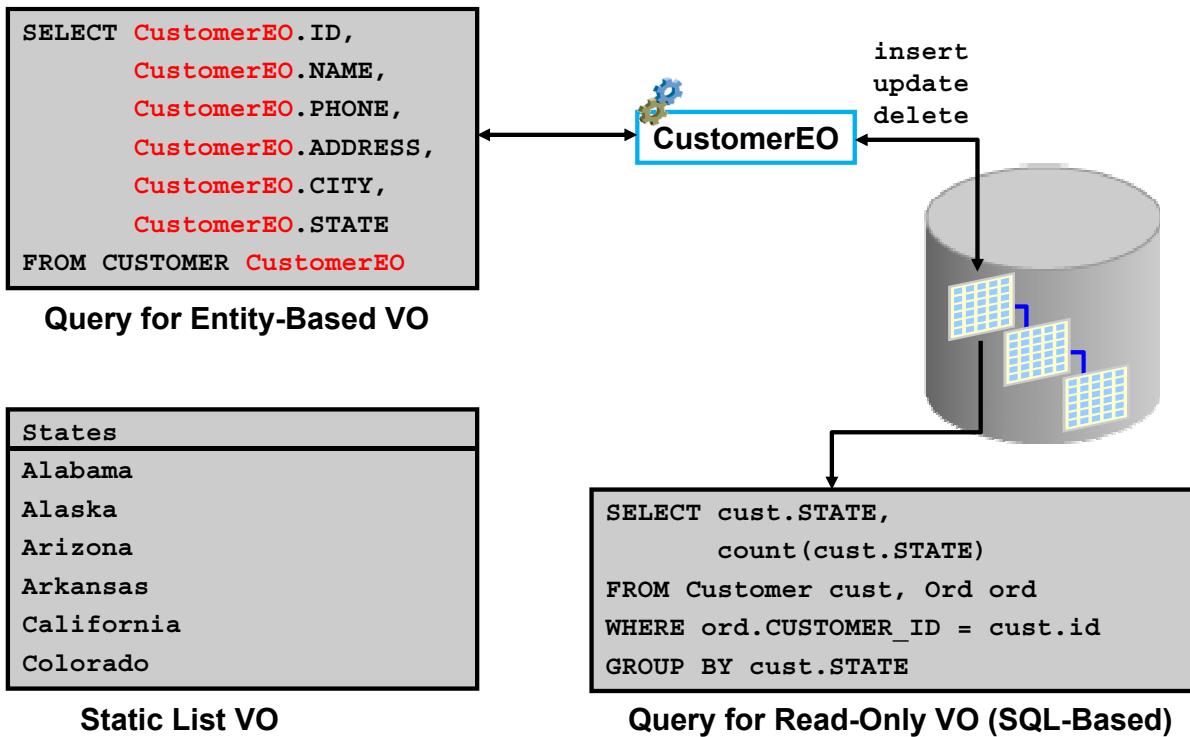
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A view object is an ADF business component that provides the ability to query your data source. Like entity objects, view objects contain attributes that represent columns or fields of data. However, the shape of the data in a view object is based on the requirements of your client interface, not the underlying data source. A view object typically encapsulates a SQL query that executes at run time to join, filter, sort, and aggregate data into exactly the shape required by the end user. The SQL query produces a row set through which you can iterate.

When end users modify data in the user interface, view objects can collaborate with entity objects to save changes and synchronize the data with other view object instances.

You can construct view objects that are based on entity objects, SQL queries, or static lists. You can also construct view objects programmatically.

Types of View Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use JDeveloper to create the following types of view objects:

- **Entity-based view objects:** These view objects contain entity object references that allow view objects to update data, use validation and other business rules from the entity object definition, and synchronize data with other view object instances. Entity-based view objects can only update data if the underlying entity object is updateable.
- **Read-only view objects:** These view objects provide a non-updateable view of the data. A read-only view object can be based on a non-updateable entity object, or an “expert-mode” SQL query (defined by you). Entity-based view objects provide performance benefits over SQL-based view objects. For example, entity-based view objects can read from the local entity object cache, which avoids re-executing the database query and ensures that the data is synchronized with other view object instances that are based on that entity object.

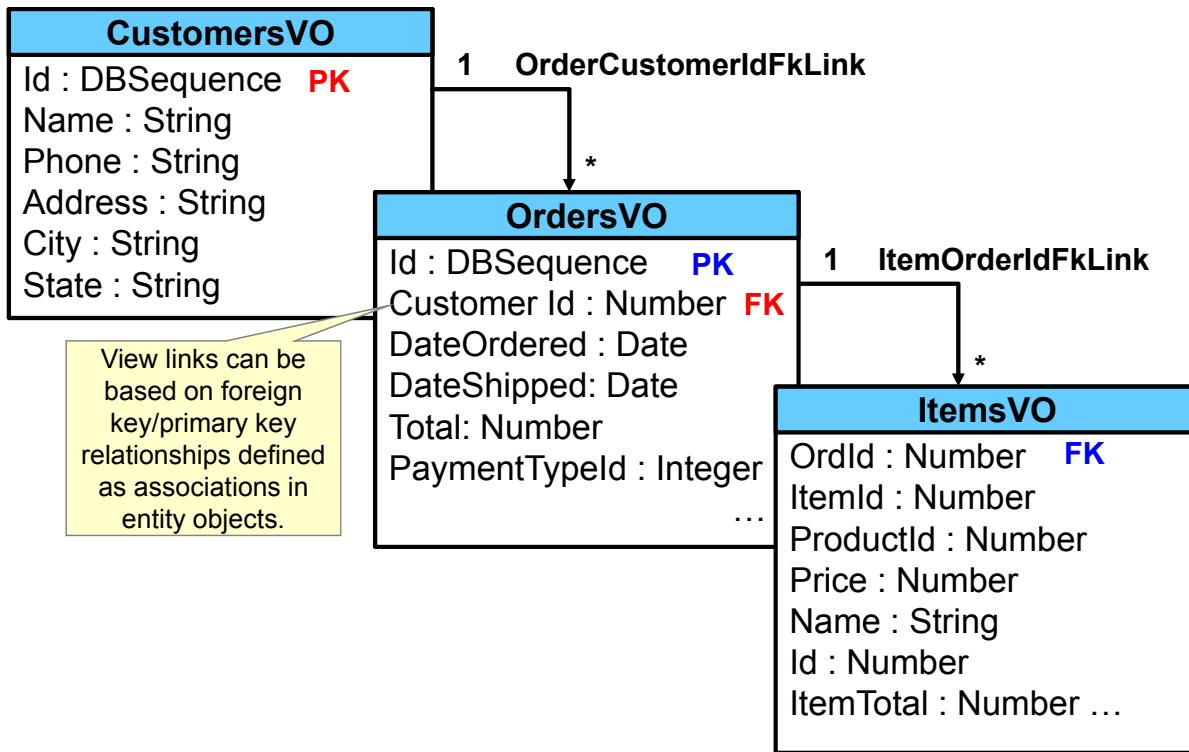
You should only create a SQL-based read-only view object when you need to execute a query that you cannot execute from an entity-based view object, or when you want to leverage the database to calculate aggregated results (for example, to perform data visualization). You must create SQL-based view objects when a query uses a GROUP BY or HAVING clause, or when it uses a set operator (MINUS, INTERSECT, UNION, and UNION ALL). The example in the slide shows a SQL-based view object that you could use to display a thematic map highlighting sales for particular states.

- **Static list view objects:** These view objects are based on static lists of values that you enter directly into the definition for the view objects or import from a comma-separate values (CSV) file. Use static list view objects when you have a small amount of data that will be updated infrequently. For example, you might create a static list view object when you have a couple of static text options (such as Yes/No, or Male/Female) that you want to provide as a drop-down selection list in the user interface.
- **Programmatic view objects:** These view objects are created by overriding methods in the custom Java class for a view object. You can use programmatic view objects to retrieve data from alternative data sources, such as a REF CURSOR, an in-memory array, or a Java *.properties file. Creating programmatic view objects is an advanced subject that is not covered in this course.

The slide shows example queries for a few different types of view objects:

- The read-only view object is based on a SQL query. The query executes every time the view of the data needs to be refreshed.
- The entity-based view object has references to the entity object, CustomerEO, and therefore can collaborate with the entity object to validate and save changes and to synchronize the data with other view object instances.
- The static list view object is constructed from a static list of data values. In this example, the view object has only a single attribute, Country, with static data values that represent each country.

View Links



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

View links are ADF business components that represent relationships between view objects. For entity-based view objects, you can create view links automatically based on associations between the underlying entity objects. You can also create view links between any two view objects manually. View links allow you to combine multiple view objects into a master-detail hierarchy. Different types of master-detail relationships are supported, including a master to single detail, a master to multiple details, or a cascading master-detail relationship of any depth. When you create your user interface, you can expose these master-detail relationships to show the user a set of master rows, and for each master row, a set of coordinated detail rows. The view links define how the master and detail view objects relate.

The example in the slide shows two view links. One view link, called OrdCustomerIdFKLink, is defined between the CustomersVO and OrdersVO view objects. JDeveloper created this link automatically based on an association that exists between the entity objects on which the view objects are based. The association is based on a foreign key/primary key relationship defined between the CUSTOMER and ORD tables. Another view link, called ItemOrderIdFkLink, is defined between the OrdersVO and ItemsVO view objects. This link is also based on an association that exists between the two entity objects on which the view objects are based. Together, the links establish a cascading master-detail relationship between the view objects. You can expose this master-detail relationship in your user interface.

Master-Detail Relationships in the UI

The screenshot displays a user interface for Customer Management. On the left, a search panel (1) shows fields for Name and City, with a 'Search' button. Below it, a table (2) lists customer records: Beisbol Sil, San Pedro de Macon's; Big John's Sports Emp, San Francisco; Delhi Sports, New Delhi; Futbol Sonora, Nogales; Hamada Sport, Alexandria; Kain's Sporting Goods, Hong Kong; Kuhn's Sports, Prague; Muench Sports, Stuttgart; Olybwray Retail, Buffalo; Simms Athletics, Osaka; Sparta Russia, Saint Petersburg; Sportique, Cannes; Unisports, Sao Paolo; Womansport, Seattle.

The main area is titled 'Customer Management' with tabs for 'Customer Muench Sports' and 'Orders Dashboard'. Under 'General Information', details for customer ID 208 are shown: Name (Muench Sports), Phone (+49-527454). Under 'Address', the address is listed as Marktplatz, Stuttgart, Germany. The 'Orders' tab (3) is selected, showing a list of orders for customer ID 208. Order details include Order Id (149-150), Customer Id (208), Date Ordered (04-03-2012-08-03-2012), Date Shipped (26-02-2012-21-02-2012), Order Total (\$1,234.35-\$32,430.00), Payment Method (CASH-CREDIT), and Order Filled (Yes-Yes). The 'ItemsVO Data' section (4) shows items for order 149, with columns Product Id, Price, Quantity, Product Name, and Item Total. The items listed are: Ace Ski Pole (\$1,075.55), Bumny Ski Pole (\$178.75), New Air Pump (\$300.00), World Cup Soccer Ball (\$1,792.00), and Junior Soccer Ball (\$154.00).

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

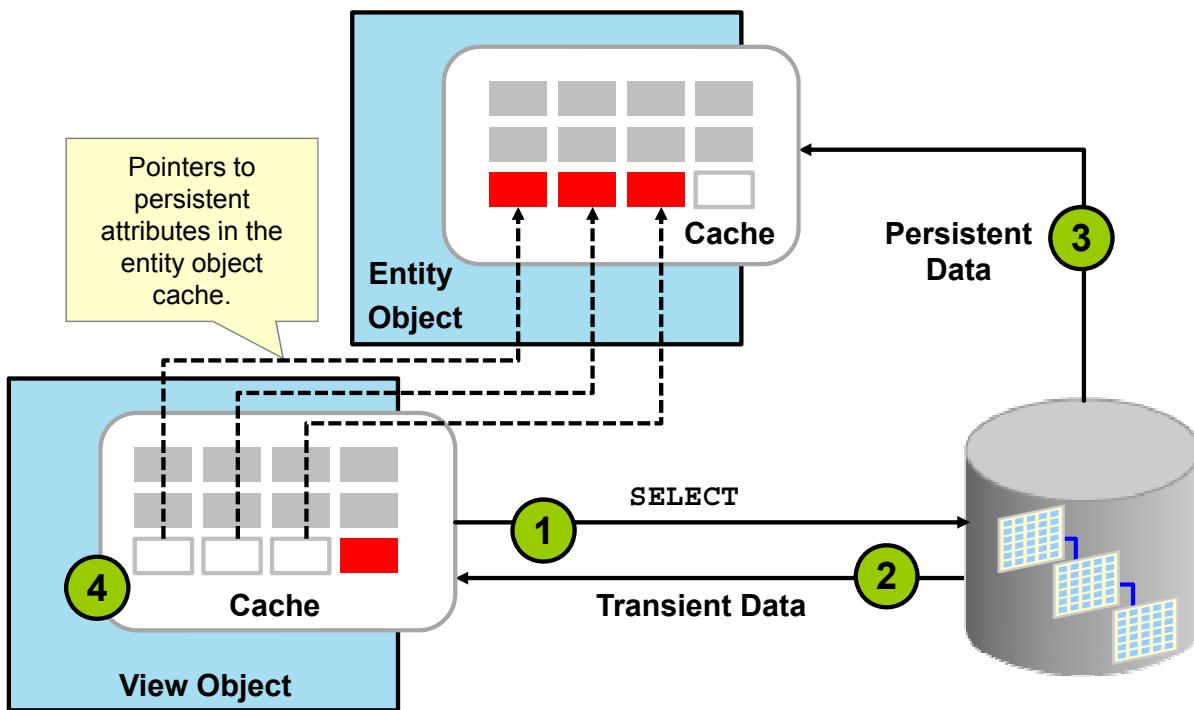


This slide shows an example of a user interface that exposes a cascading master-detail relationship between the CustomersVO, OrdersVO, and ItemsVO view objects.

1. The search tab contains a list of master customer records. The application accesses the customer data through the CustomersVO view object.
2. A user selects a master customer record to see the details about a customer.
3. A list of orders for the selected customer is displayed. The application accesses data about customer orders through the OrdersVO view object (represented in the application module by the OrdersForCustomer view object instance).
4. A user clicks a master order record to see details about an order.
5. Details about the order are displayed. The application accesses data about order details through the ItemsVO view object (represented in the application module by the ItemsForOrder view object instance).

As the previous slide explains, the view objects are related through a cascading master-detail relationship that is defined as view links between the three view objects. The view links enable the framework to synchronize the data across the three UI components.

Interaction Between View Objects and Entity Objects: Retrieving Data



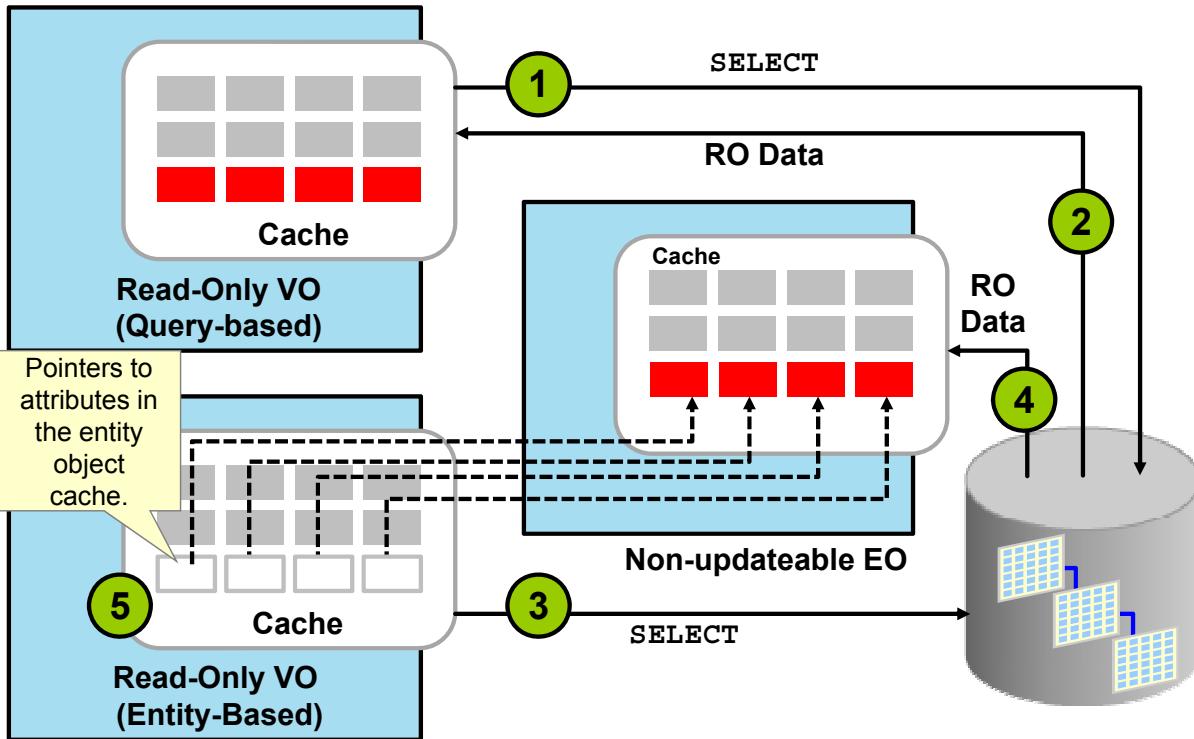
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As you learned earlier, a view object typically retrieves data by using a SQL query, and a view object is usually bound to an entity object, which is responsible for persisting changes to the database. The diagram in this slide shows how view objects and entity objects work together to cache data that is retrieved from the database. This slide discusses persistent and transient data. The caching of read-only data is covered in another slide in this lesson.

1. The view object queries the database, ensuring that its data is current.
2. Any transient data (represented by solid red blocks in the view object cache) is saved directly to the view object cache. Transient attributes are not mapped to attributes in an entity object and do not need to be persisted to the database. For example, you might have a calculated attribute called `Total_sal` that derives its value from the following SQL query: `select (12*monthly_sal)`. The `Total_sal` attribute is not mapped to an entity object and does not need to be persisted to the database. Therefore, its value is stored in the view object cache.
3. Any persistent data that is retrieved by the query (represented by solid red blocks in the entity object cache) is saved to the entity object cache. Persistent attributes represent columns in a database table.
4. The view object cache is populated with pointers to any persistent attributes that are stored in the entity object cache.

Interaction Between View Objects and Entity Objects: Retrieving Read-Only Data



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The caching process is similar for read-only view objects. However, read-only view objects can be based on expert-mode SQL queries or non-updateable entity objects. The diagram in this slide shows how data is cached for these two types of read-only view objects:

1. If the read-only view object is based on an expert-mode SQL query, the view object queries the database, ensuring that its data is current.
2. The data returned by the database is saved directly to the view object cache.
3. If the read-only view object is based on a non-updateable entity object, the view object queries the database, ensuring that its data is current.
4. The data returned by the database is saved to the entity object cache.
5. The view object cache is populated with pointers to the read-only data in the entity object cache.

In general, you should base view objects on entity objects unless you need to execute a query that you cannot execute from an entity-based view object, or when you want to leverage the database to calculate aggregated results (for example, to perform data visualization). You must create SQL-based view objects when a query uses a GROUP BY or HAVING clause, or when it uses a set operator (MINUS, INTERSECT, UNION, and UNION ALL). When you base view objects on entity objects, you can take advantage of entity object caching, which is more powerful than the caching mechanism used for view objects.

Advantages of Basing a Read-Only Object on an Entity Object

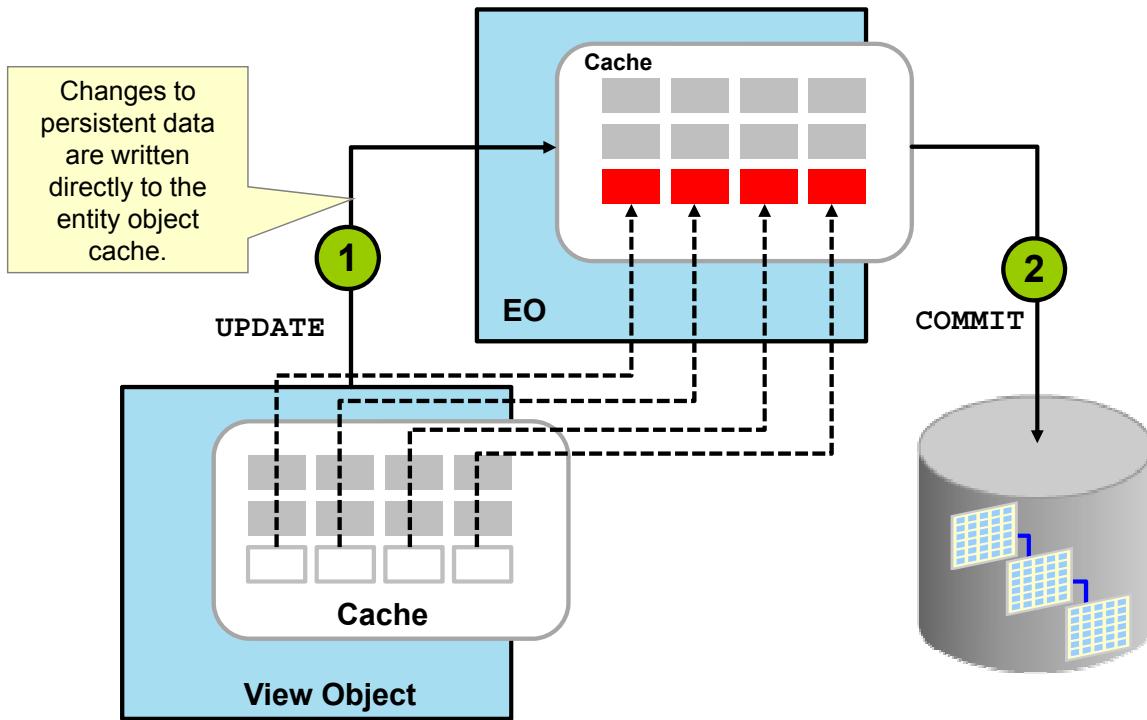
Basing a read-only view object on a non-updateable entity object enables you to use the entity object cache for storing data and provides the following advantages:

- Using the entity object cache saves space. Data is stored in one place, the entity object cache, and not duplicated in multiple view object instances.
- The entity object cache maintains a consistent view of the data. All view object instances that reference data in the entity object cache see the same view of the data. When you store data in the entity object cache, changes to data are visible to all views that reference the entity object. This means that read-only view objects based on non-updateable entity objects have access to the current transaction data that has not been committed.

When you store data in the view object cache, however, each view object instance has a separate copy of the data. If the value changes in one view object instance, other view object instances cannot see the change, and the data becomes inconsistent.

- Business logic defined in the entity object is reusable. If you define business logic in an entity object and map view objects to the entity object, you can use the same business logic in many different views of the data.

Interaction Between View Objects and Entity Objects: Persisting Data



ORACLE

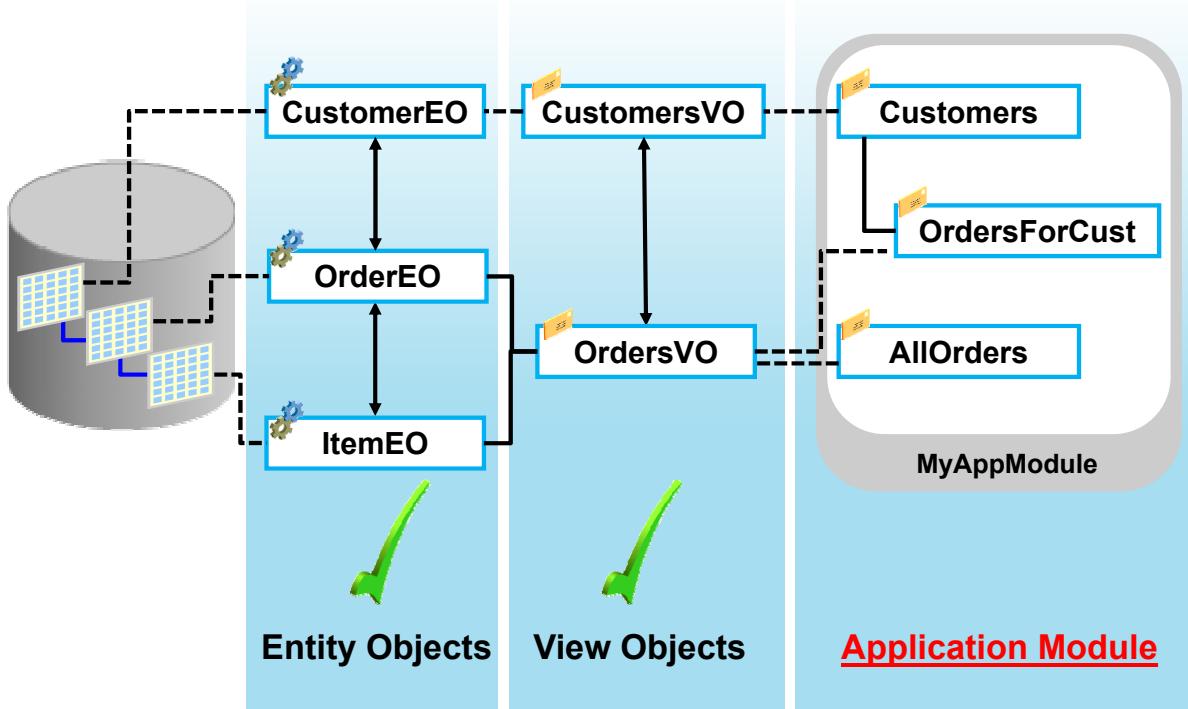
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When end users modify data in the user interface, entity objects are responsible for caching the data, performing data manipulations, and updating the database when transactions are committed by the application module. The diagram in the slide shows how a view object and entity object interact to persist changes:

1. The view object updates the entity object cache with changes. Because the data needs to be persisted, the data is written directly to the entity object cache. Remember that the view object cache contains pointers to persistent attributes in the entity object cache.
2. When the transaction is committed by the application module, the entity object updates the database.

The entity object guarantees the validity of the data that is committed to the database. If the data does not pass entity object validation, then the commit ceases. If the data passes entity object validation, but does not pass database validation, then the commit ceases as expected.

Types of ADF Business Components: View Objects Summary

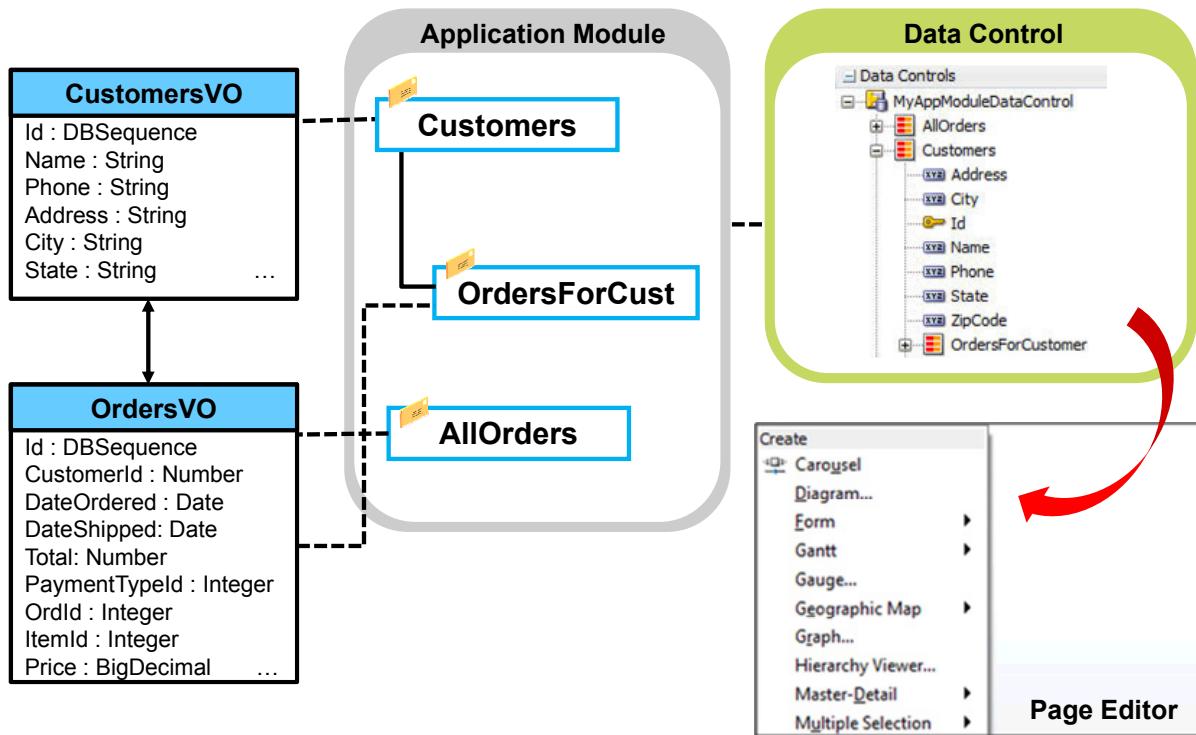


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Now you understand what view objects are and how they can be related to other view objects through view links. You know that a view object typically encapsulates a query (a SELECT statement) and that view objects collaborate with entity objects to read, cache, and write data to the database. Next, let's learn about application modules.

Application Modules



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Application modules are containers for the view object instances and links related to a specific business task. For example, the diagram shows an application module that is used to manage customer orders. The application module wraps the following view object instances, which are required for managing customer orders: Customers, OrdersForCust, and AllOrders.

- The Customers view object instance exposes the CustomersVO view object and related links.
- The OrdersForCust view object instance exposes OrdersVO. However, because OrdersVO is linked to CustomersVO through a view link, the OrdersForCust view object instance exposes orders for the current customer only.
- The AllOrders view object instance exposes all orders through OrdersVO.

In this way, the application module defines the updateable data model (and, as you learn later, service methods) that UI clients use to work with application data.

Another important responsibility of the application module is to provide a single connection to the database and handle all database transactions. The application module is a transactional container for a logical unit of work, which means that any updates, inserts, or deletes for all view objects in the application module are committed or rolled back at once. Application modules can also contain references to other application modules.

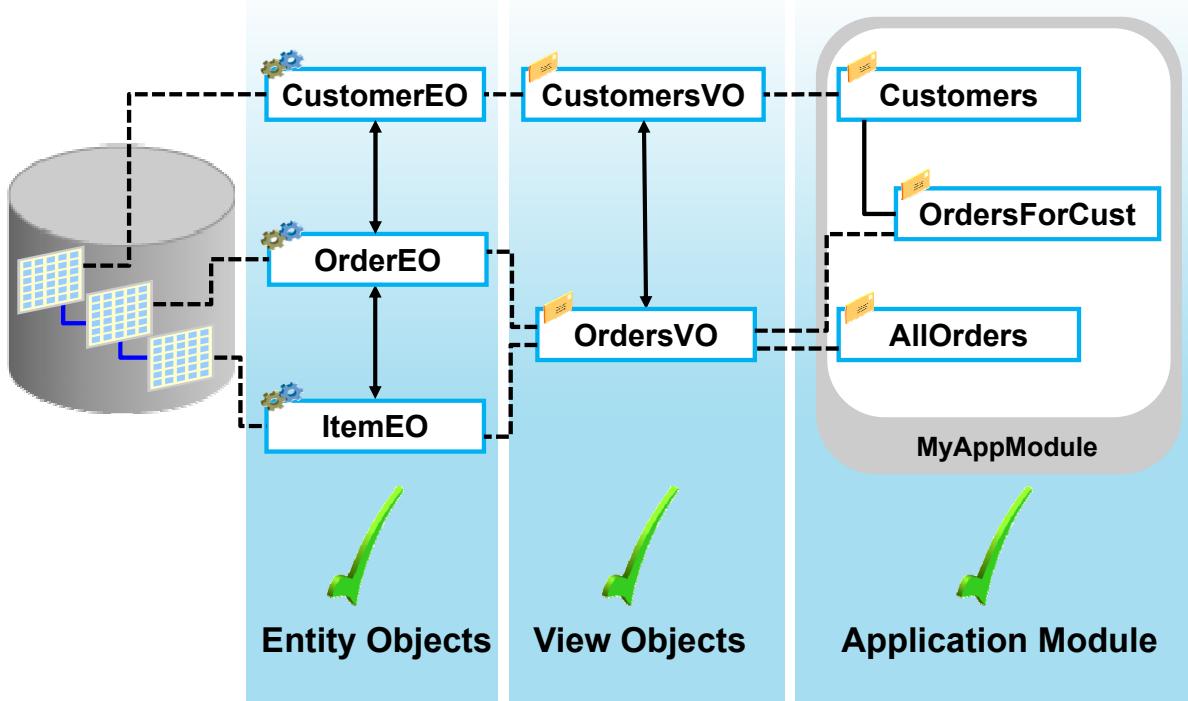
Like other ADF business components, application modules are reusable in other applications.

When you use ADF BC to create an application module, you select the view objects and view links that you want to expose in the data model, and those components effectively become part of a data control that is available for the application module.

The data control provides a hierarchical view of data collections, attributes, and operations that you can drag and drop to the page editor. (Note that the data control in the diagram shows a subset of the data objects that are actually available in the data control; the data control exposes other objects, such as built-in operations, that are not shown.) You can use the data controls to build data-bound UI components, such as text fields, selection lists, forms, tables, graphs, and many other types of components, simply by dragging and dropping data objects onto a page editor. When you drag and drop data objects to a page, a context menu displays a list of UI components that you can create for the selected object.

When you use business components that are not ADF BC, such as EJBs or web services, you must generate the data controls for those components manually.

Types of ADF Business Components: Summary



ORACLE

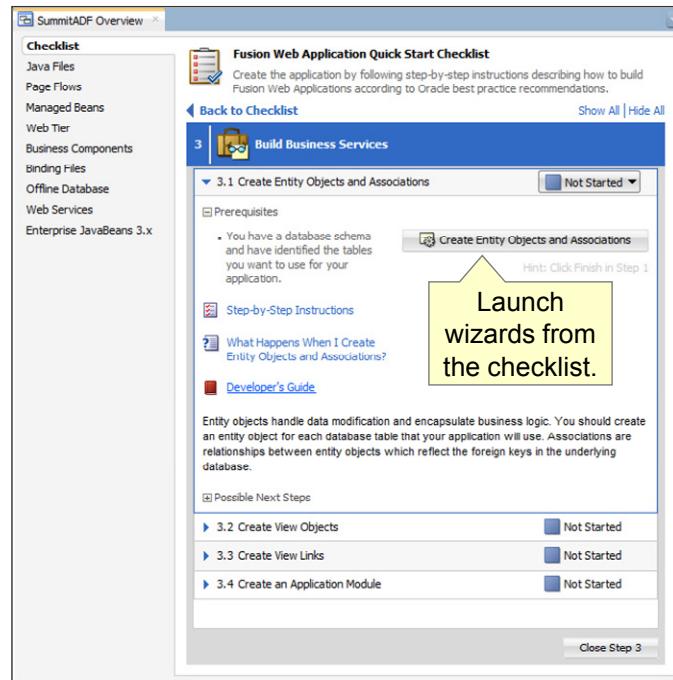
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Now you understand the basics of ADF business components. You understand that:

- Entity objects are responsible for handling database persistence (issuing insert, update, and delete statements) and caching. Entity objects can be related to other entity objects through associations.
- View objects typically encapsulate a query (a SELECT statement) and collaborate with entity objects to read, cache, and write data to the database. View objects can be related to other view objects through view links.
- Application modules define the data model and procedures and functions (called service methods) that UI clients use to work with application data.

Creating ADF Business Components

- Follow best practices by using the Quick Start Checklist (select Application > Show Overview).
- Create a first cut by running the Create Business Components from Tables wizard.
- Create BCs individually for finer control over specifying properties.
- Model BCs in a diagram.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Now that you have learned about the basic building blocks of ADF business components (entity objects, associations, view objects, view links, and application modules) and how they are related, let's learn how to create them.

JDeveloper provides several different ways to create ADF business components: You can use the Fusion Web Application Quick Start Checklist to walk through each step of building your business components, you can use the Create Business Components from Tables wizard to build a first cut of your business components quickly, you can invoke individual wizards from the New Gallery to define one component at a time, or you can create business components from within a business components diagram.

The Fusion Web Application Quick Start Checklist leads you through the steps required to build an application that follows Oracle best practices. You can return to the checklist after completing each step in the development process to verify that you are following the recommended sequence of steps. You usually create the components in the following order: entity objects, associations, updateable view objects, view links, read-only view objects, and one or more application modules.

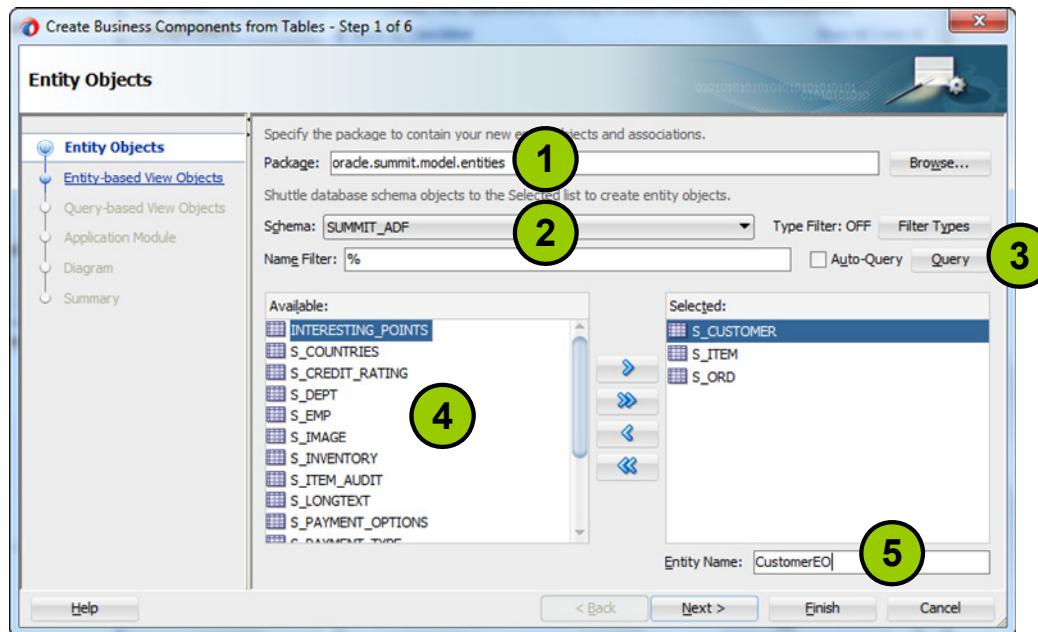
To create components, right-click your model project and select New > From Gallery from the context menu. Select ADF Business Components. The New Gallery enables you to create several components in a variety of ways by invoking either a wizard or modeler. Notice that you can also launch the wizards directly from the Quick Start checklist, or by right-clicking items in the Applications window.

To create a first cut of your business components quickly, you can use the Create Business Components from Tables wizard and create several types of components at once. Later, you can edit any default business components that you create, and you can add new business components. If you want finer control over specifying the properties of your components, you can use individual wizards, such as the Create Entity Object wizard, to create the components individually.

You can also create business components by creating a business components diagram. Creating a diagram enables you to visualize the relationships between the business components, and create and modify them visually. This course does not describe how to use the business components diagram, but you may want to experiment with it on your own.

Regardless of the approach that you follow, you can refine the properties of the objects later.

Create Business Components from Tables Wizard: Entity Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

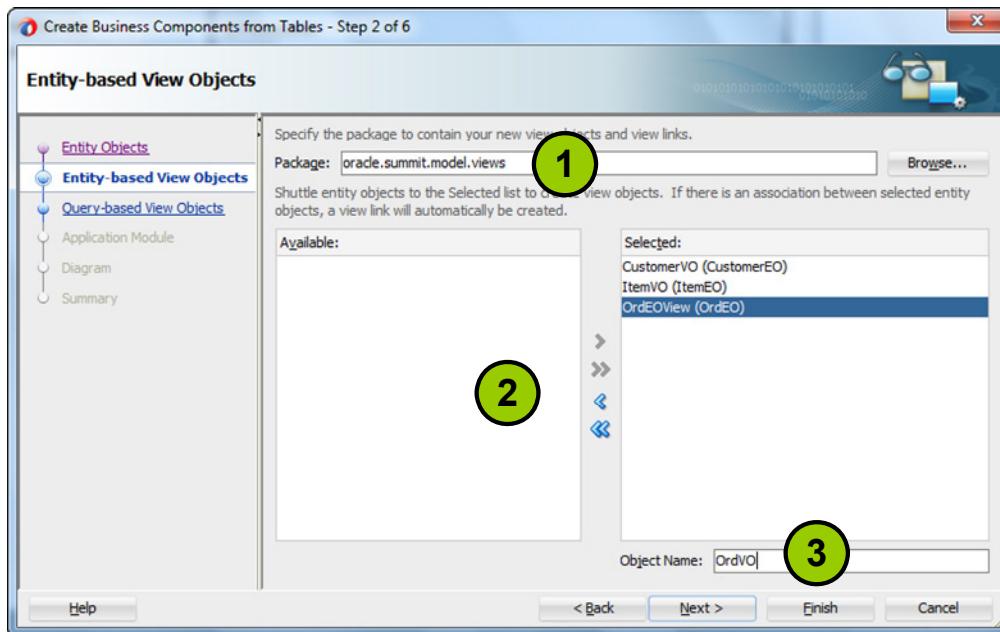
On the Entity Objects page of the Create Business Components from Tables wizard, you create default entity object definitions for the specified database objects. The wizard also creates associations based on all relevant foreign-key relationships.

To create entity objects:

1. Specify the name of a package to contain the entities. To make it easier to work with the business domain objects, store the entity objects in a separate package from other business components. In the example, the entities are stored in the package named `oracle.summit.model.entities`.
2. Select the database schema that you want to use.
3. Click Query to populate the Available list with the tables, views, and synonyms in that schema, or enter a name filter to display only certain ones.
4. In the Available list, select the database objects for which you are creating entity objects, and shuttle them into the Selected list.
5. In the Selected list, select each object and change the entity name to follow the naming conventions that you have established for your project. A common convention is to append EO to the end of entity object names and to use singular names.

Create Business Components from Tables

Wizard: Entity-Based View Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

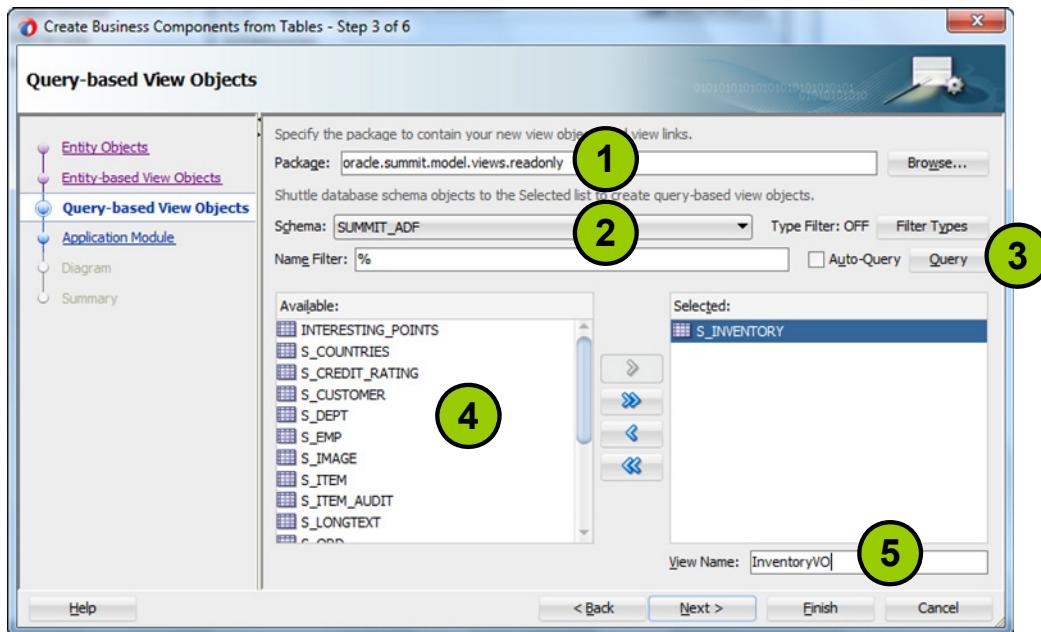
On the Entity-based View Objects page of the wizard, you create updatable view objects that are based on the specified entity objects. The wizard also creates view links for all relevant association definitions. The view object definitions that you create on this page expose all attributes in the entity object. You can remove attributes from the view objects later when you edit them.

To create entity-based view objects:

1. Specify the name of a package to contain the view objects. To make it easier to work with the view objects, store them in a separate package from other business components. In the example, the view objects are stored in the package named `oracle.summit.model.views`.
2. In the Available list, select the entity objects for which you are creating view objects, and shuttle them into the Selected list.
3. In the Selected list, select each object and change the object name to follow the naming conventions that you have established for your project. A common convention is to append VO to the end of view object names.

Create Business Components from Tables

Wizard: Query-Based View Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

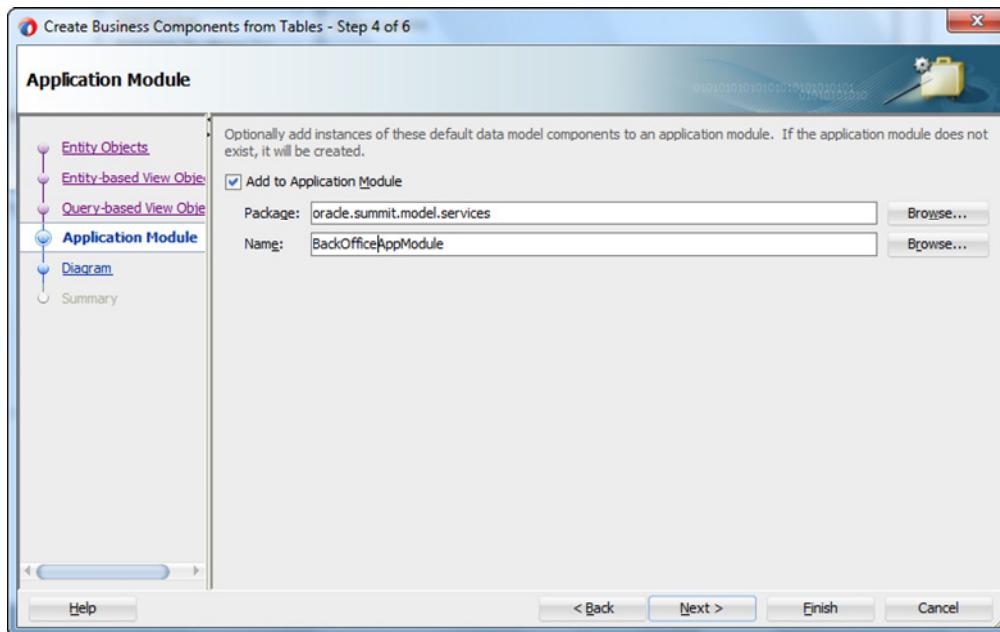
On the Query-based View Objects page of the wizard, you create read-only view objects and view links that are based on SQL queries instead of entity objects. As you learned previously, creating SQL-based view objects provides you with more control over the query. However, because the view objects do not use the entity object cache, you lose some of the performance benefits and data synchronization that is provided by the entity object cache. To make use of the entity object cache, do not use this page of the wizard, and instead create read-only view objects that are based on non-updateable entity objects.

To create a query-based view object:

1. Specify the name of a package to contain the view objects. To make it easier to work with the view objects, store read-only view objects in a separate package from other view objects. In the example, the view objects are stored in the package named `oracle.summit.model.views.readonly`.
2. Select the database schema that you want to use.
3. Click Query to populate the Available list with the Tables, Views, and Synonyms in that schema, or enter a name filter to display only certain ones.
4. In the Available list, select the database objects for which you are creating read-only view objects, and shuttle them into the Selected list.

5. In the Selected list, select each object and change the view name to follow the naming conventions that you have established for your project. A common convention is to append VO to the end of view object names, and for view object names to be plural because they typically represent a collection of rows.

Create Business Components from Tables Wizard: Application Module

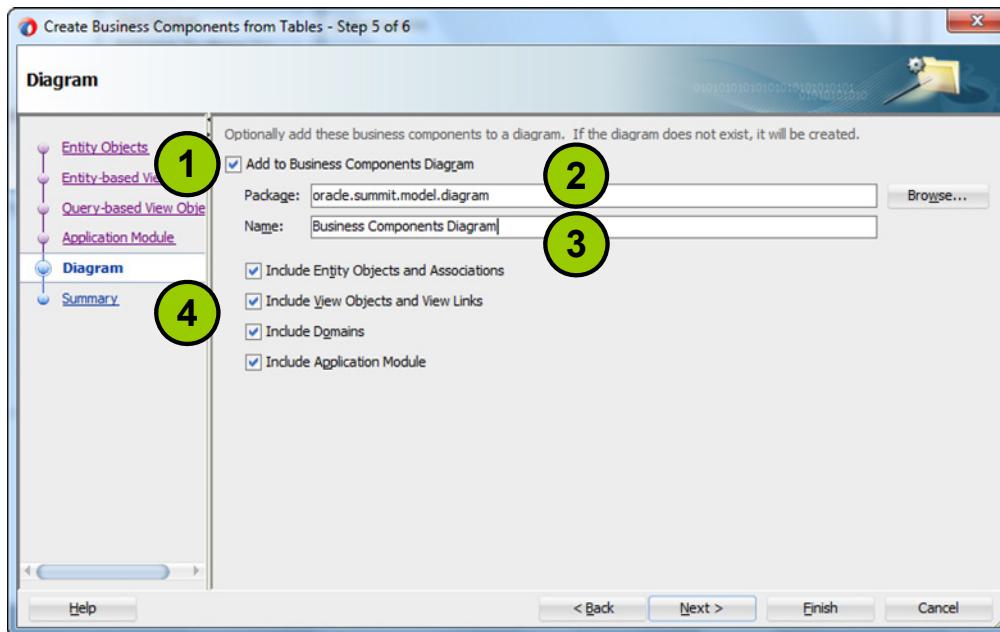


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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. You can specify an existing application module, or create a default application module that contains instances of all the view objects. The application module will contain instances of every view object the wizard creates, joined in master-detail relationships in every possible combination by instances of the view links that the wizard creates. As a best practice, you generally want to skip this step in the wizard and add your business components to an application module explicitly. This will give you full control over the view object instances and view object links that are added to the application module.

If you are creating a new application module, specify the name of a package to contain it, and give the application module a name. To make it easier to work with the application modules, store them in a separate package from other business components. In the example, the application module is stored in the package named `oracle.summit.model.services`.

Create Business Components from Tables Wizard: Diagram



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

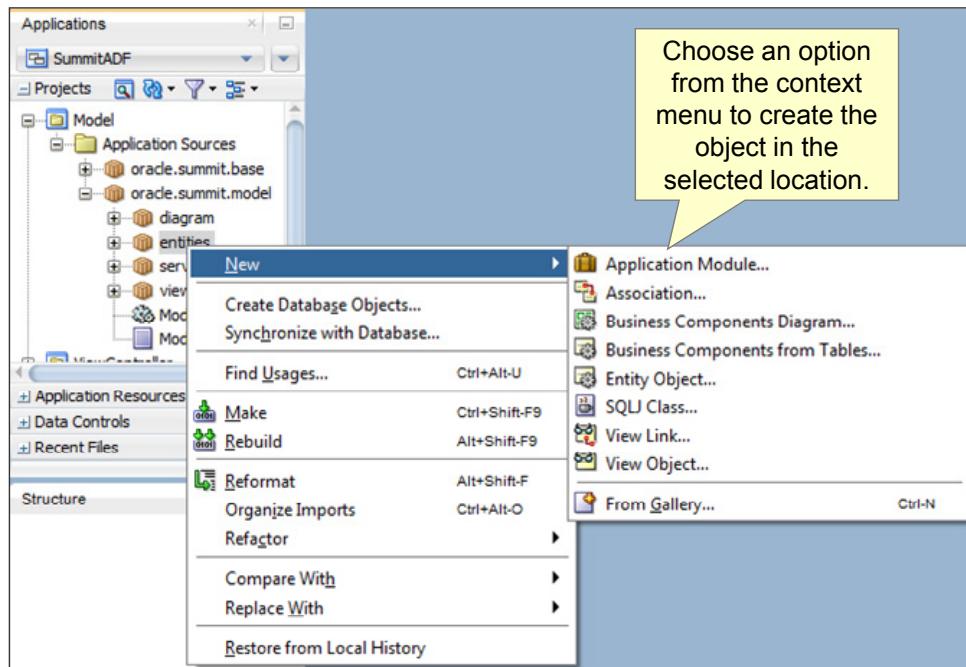
To add diagram elements to a diagram:

1. Select Add to Business Components Diagram.
2. Specify the name of a package to contain the diagram. To make it easier to work with the diagrams, store them in a separate package from your business components. In the example, the diagram is stored in the package named `oracle.summit.model.diagram`.
3. Specify the name of the diagram.
4. Select the 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 the objects that will be created by the wizard. Click Finish to create the business components.

Note: When you run the Create Business Components from Tables wizard, you see a subset of the wizard pages that are available when you use the individual wizards to build business components.

Launching Individual Wizards to Create Business Components

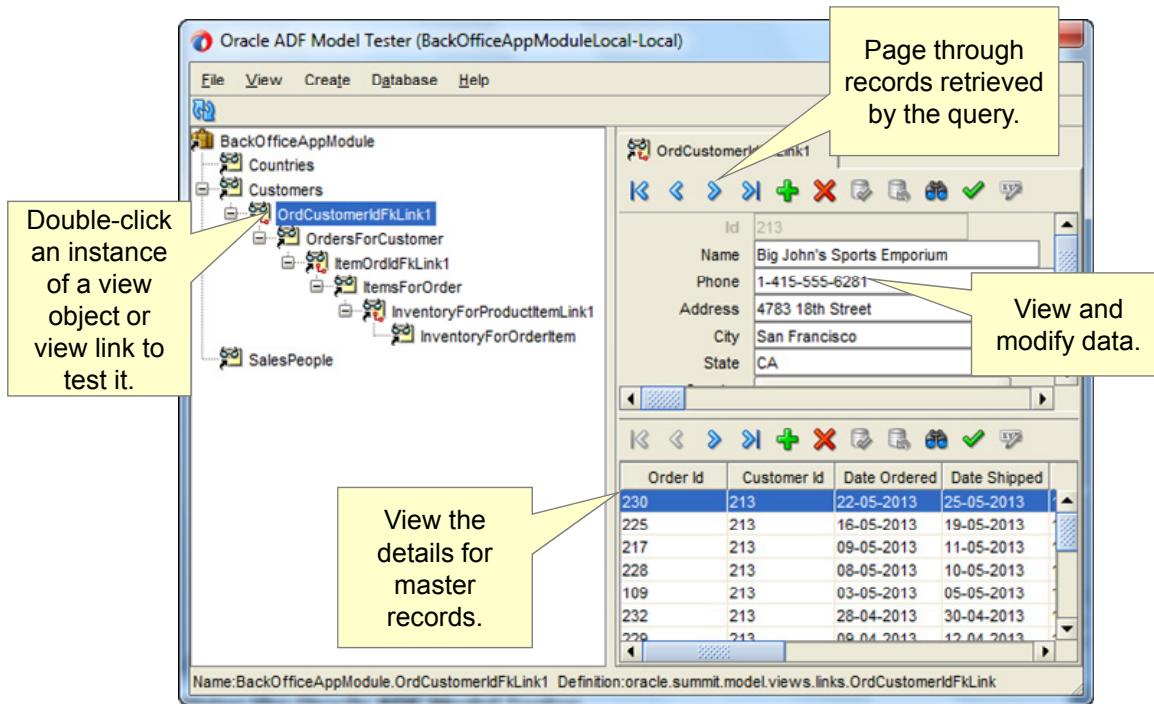


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You learned previously that you can use the Create Business Components from Tables wizard to create a first cut of your business components quickly. However, the wizard does not allow you to customize all aspects of the objects that you create. For example, you can specify the view objects and entity objects, but you cannot specify the attributes to include in those objects, and you cannot specify details of the attributes, such as the Java type, the default value, or whether the attribute is persistent. Of course, you can customize the objects later, but if you want finer control over specifying the details of your business objects from the beginning, you can use the individual wizards.

The most convenient way to launch the individual wizards is to right-click an item in the Applications window and select an option from the context menu. The options in the context menu vary according to the item that is selected. For example, if you right-click a package name, you see a list of all the types of components that you can create in the package. If you right-click an entity object, you see a list of objects that you can create based on the selected entity (for example, a new default view object for the entity). When you launch the wizards from the project tree, some of the fields in the wizard are automatically populated with information based on the item that you select. For example, if you select a package name, the wizard is automatically populated with the correct package name.

Testing the Data Model



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

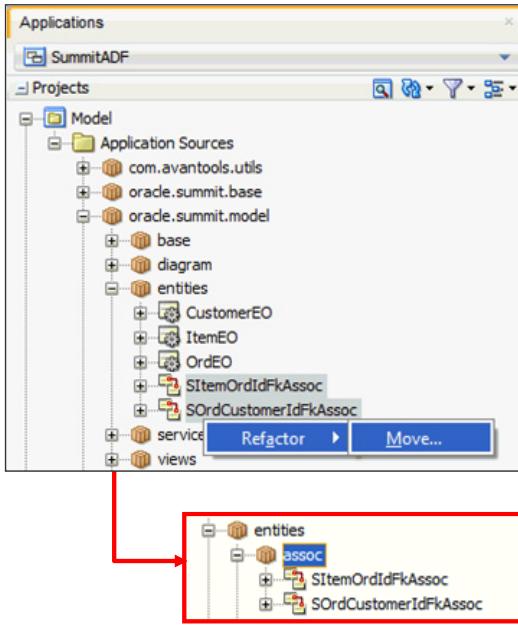
Using the Oracle ADF Model Tester

Early in the development process, you can use the Oracle ADF Model Tester to test instances of your view objects without having to develop a user interface or write a test client program. In fact, you can simulate an end user interacting with your application module data model before you have started to build any custom user interface of your own. Even after you have developed UI pages for your application, you can use the Oracle ADF Model Tester to diagnose problems and reproduce issues to determine whether the issue exists in the business services layer of the application.

Before running the Oracle ADF Model Tester, you need to define an application module and specify the view object instances and view link instances to include in the application module.

To launch the Oracle ADF Model Tester, right-click the application module in the Applications window, and select Run from the context menu. The Oracle ADF Model Tester opens and displays all the view object instances and view link instances in your application module. Double-click a view object instance or view link instance to display the first row of data. You can iterate through the rows of data, add a row, delete a row, commit changes, and test your validation rules. If you are testing a view link instance, you see the master record and corresponding details.

Refactoring Components



After Refactoring

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
- Preview and exclude individual usages before applying the change

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After creating business components, you might want to rename them or move them to a different package. To do this, you use refactoring:

- **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. You might want to rename them to use names that are more meaningful. To rename an object, you right-click it in the Applications window, and from the context menu, select Refactor > Rename. Optionally, you can preview usages to inspect and modify or exclude individual usages before you proceed with the renaming. The preview is displayed in the Log window.
- **Moving:** When you create an object, you are given the opportunity to specify which package to place it in. However, when you use a wizard to create entity objects and view objects, the associations and view links are created automatically in the same package as the entities and view objects. If you want to change the package name (for example, to place all view links in a single package), you can move the objects to a new location. To move an object, right-click the object and select Refactor > Move. If the package that you specify does not exist, you can choose to have the wizard create it.

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

Summary

In this lesson, you should have learned how to:

- Identify the types of service components and describe how they are used in a web application
- Explain how entity objects relate to database tables and maintain persistence in an application
- Create entity objects and associations from database tables
- Create view objects and view links
- Explain the role of application modules
- Create an application module and test the application
- Refactor business components



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 3 Overview: Creating and Testing ADF Business Components

This practice covers the following topics:

- Creating a first cut of ADF business components by using the Create Business Components from Tables wizard
- Testing the ADF business components in the Oracle ADF Model Tester
- Optional: Inspecting the data model in the Database Navigator



Copyright © 2014, 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 NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

Creating Data-Bound UI Components

4

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

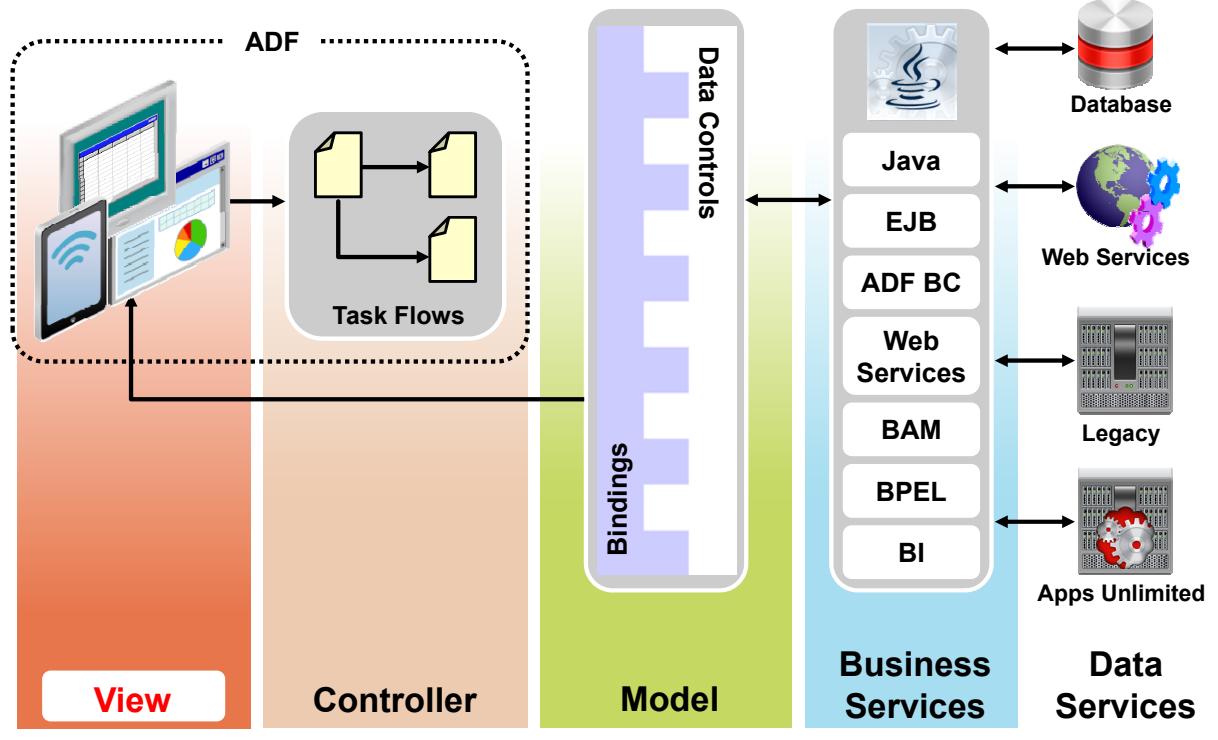
After completing this lesson, you should be able to:

- Define JavaServer Faces and the JSF component architecture
- Explain how JSF managed beans are used in an application
- List some JSF component types included in the standard implementation
- List some ADF Faces component types that extend the basic JSF component types
- Explain the purpose of a data control
- Decide between creating pages based on Facelets or JSP XML
- Create a JSF page that contains data-bound components



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Building the View Layer



ORACLE

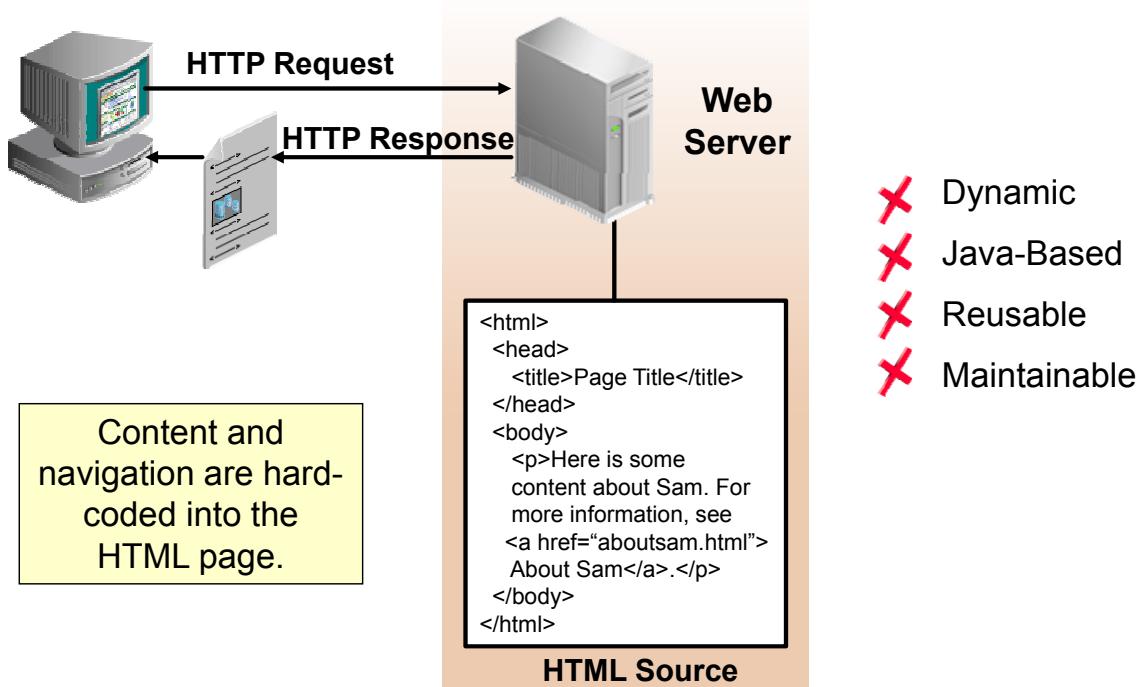
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the previous lesson, you learned how to build the business services layer of your application. You learned about the three main building blocks of ADF BC: entity objects, view objects, and application modules, and how they work together to query, cache, manipulate, and persist data from data sources. You also learned that applications that use ADF BC are automatically data-control aware.

In this lesson, you learn the fundamentals of building the view layer. First, you learn about the evolution of web technologies from the early design of dynamic webpages through to newer Java-based technologies. Then, you learn about ADF Faces and the underlying JavaServer Faces technology on which ADF Faces is built. You also learn about ADF Faces UI components and how to use data controls to build data-bound UI components.

This lesson shows you how to use data controls, but it does not explore in depth how data controls are bound to user-interface components. You learn more about data controls and bindings in a later lesson.

In the Beginning: Static HTML



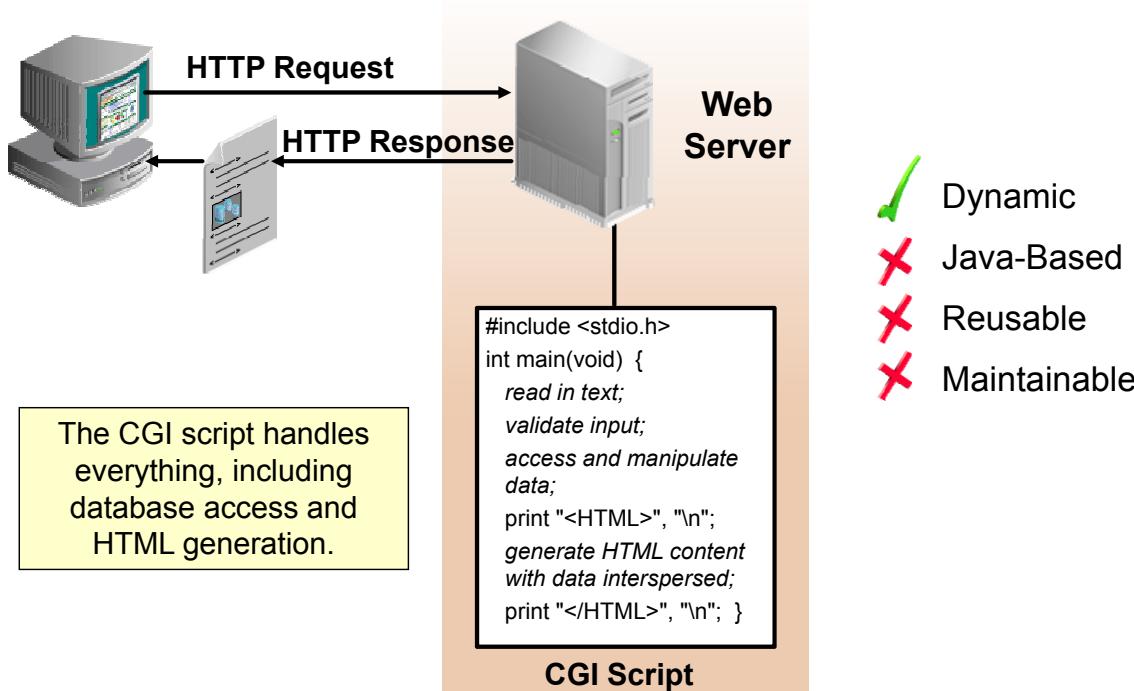
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the early days of the World Wide Web, pages were written in HTML and contained static data. HTML documents were transmitted to a web browser from a web server through HTTP. The browser sent an HTTP request to the web server, which returned a response in the form of the requested HTML document and other content, such as images and sound. Navigation to other pages was hard-coded as hyperlinks within pages.

This model worked well for static content. However, static webpages were difficult to maintain and could not be customized for individual users. When a page was added or removed, every page that linked to the new (or outdated page) had to be updated. Keeping large websites up-to-date became a maintenance nightmare.

Dynamic Webpage Technologies: CGI



ORACLE

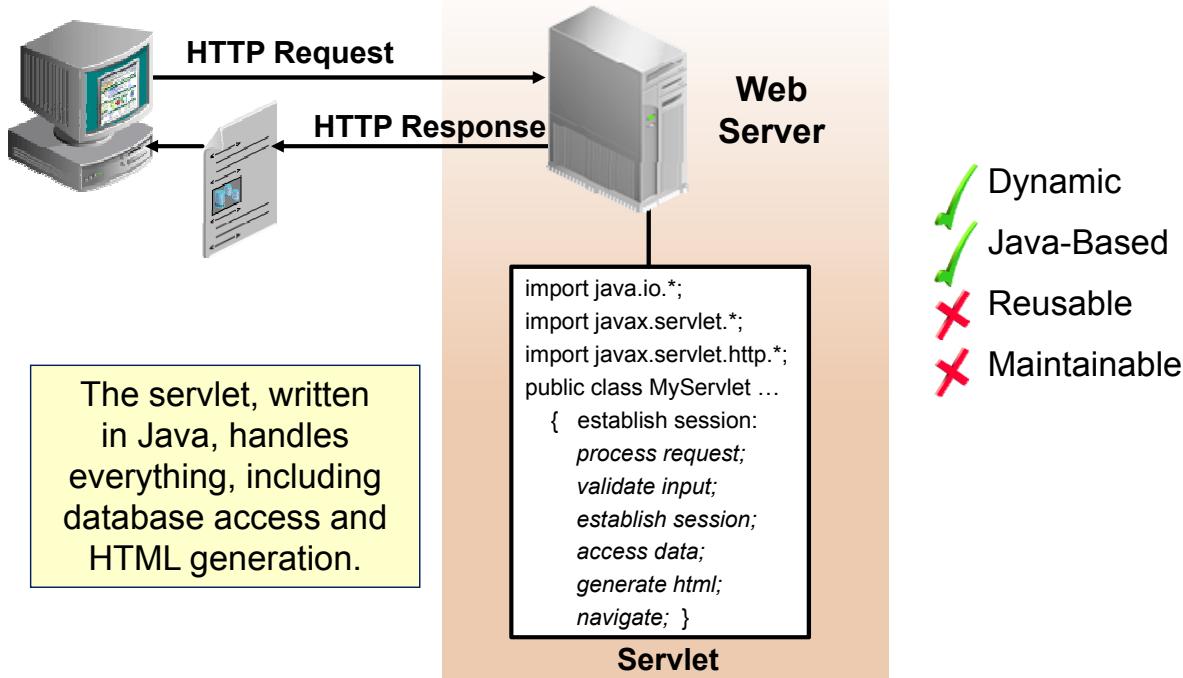
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Common Gateway Interface (CGI) was introduced to provide a mechanism for including dynamic data in webpages. The first CGI scripts were written as C programs.

Here is how CGI scripts work: When a browser sends a request to the web server to access a URL that references a CGI script, the web server delegates generation of the HTML response to the CGI script. The CGI script reads in the text from the request, performs any required data validation, handles back-end database access, performs calculations and data manipulations, and generates a stream of HTML interspersed with dynamic data. The web server then sends a stream of HTML back to the browser as an HTTP response, which the browser renders as an HTML page.

This approach worked well and enabled web developers to create webpages that contained dynamic content. However, CGI scripts were tedious to code, and they were difficult to debug and maintain. Invoking a new server process for each HTTP request resulted in additional overhead and consumed too much memory, which affected performance. Web developers required a better solution.

Dynamic Webpage Technologies: Servlets



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

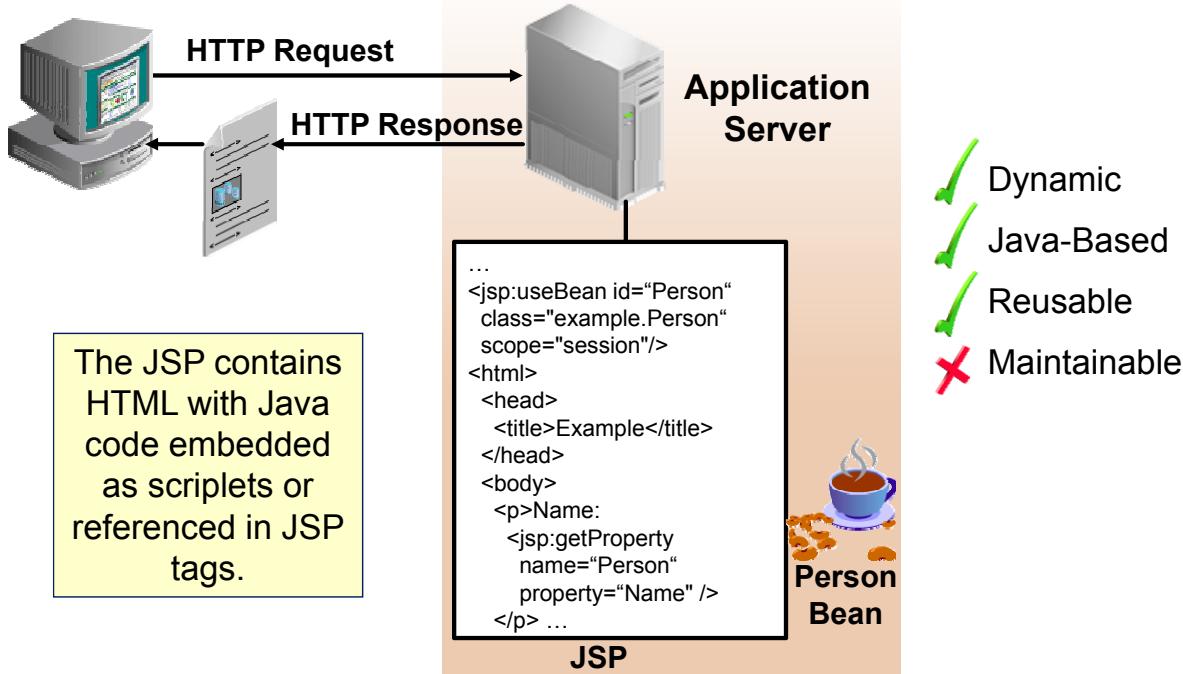
By the late 1990s, Java was quickly becoming a popular programming language. Java was object-oriented, could run on a variety of platforms, was multithreaded, and had built-in support for security and memory management. Sun Microsystems, Inc, developed the idea of using Java to generate dynamic webpages. They called this technology a servlet.

A servlet is a server-side Java class that extends the functionality of the web server by processing client requests and generating dynamic webpages. Much like a CGI script, a servlet reads data from the HTTP request stream sent by a browser, performs any required data validation, handles back-end database access, performs calculations and data manipulations, and generates a stream of HTML, interspersed with dynamic data, to send back in the HTTP response. The servlet also handles other tasks related to maintaining a session, such as setting up cookies. Page navigation is coded directly into the servlet.

Servlets offer developers several advantages over CGI scripts. Servlets provide faster performance than CGI scripts because servlets create a separate thread for each HTTP request (rather than spawning a separate process for each request). In addition, servlets require less memory than CGI scripts because a single instance of the servlet is created in memory to handle all HTTP requests, and the servlet stays in memory between requests.

This new model solved some problems, but introduced new ones. Java programmers had to write HTML that was embedded in Java code, but they were not proficient in HTML. The code itself was difficult to maintain and debug.

Dynamic Webpage Technologies: JavaServer Pages (JSP)



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Servlets provided the ability to generate dynamic HTML pages and they addressed some of the performance issues inherent in CGI scripts. However, servlets were hard to code and difficult to maintain. In addition, because the presentation logic (the logic for presenting a view of the data to the user) was mixed with business logic (the logic that interacted with the database and performed operations), changes to one often resulted in changes to the other.

The next step in the evolution of dynamic webpage technologies was to separate the HTML in the pages from the Java code so that the HTML developers could focus on designing webpages, and the Java developers could focus on the programming logic. To accomplish this goal, Sun Microsystems, Inc, developed the JavaServer Pages (JSP) technology.

A JSP is an HTML file, saved with a .jsp extension, that includes a combination of standard HTML tags and JSP tags. The JSP tags allow HTML developers to embed Java code within the HTML. When the browser sends the first HTTP request to access the JSP, the JSP is compiled into a servlet, which responds to the HTTP requests.

To reduce the amount of code in the HTML page even further, developers can write JavaBeans that encapsulate programming logic. JavaBeans are reusable software components that encapsulate Java code. For example, a JavaBean might contain the logic for retrieving and manipulating data from a database. JavaBeans include properties, along with getter and setter methods, for getting and setting property values. To call methods on the bean, page developers include simple JSP tags in the JSP file. JavaBeans remain in memory for a defined period of time.

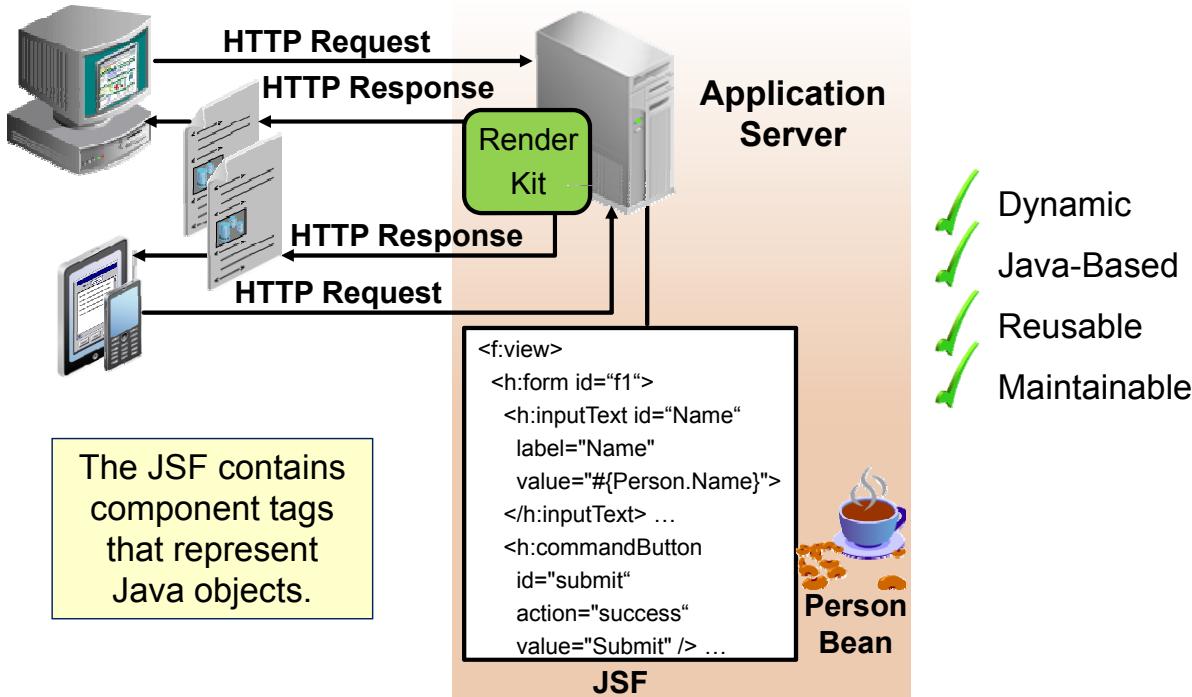
The JSP design provided a nice separation between the Java code and the HTML code. The HTML programmer knew just enough about JSP to use tags that called methods on the JavaBeans to access data dynamically, and the Java programmer had full control over the programming logic.

Over time, however, inline Java became more common in JSPs, which made the code harder to reuse and maintain. On the server side, developers did not like coding everything into a single servlet.

Eventually, frameworks (such as Struts) came along to address some of these problems and provide an additional controller servlet to handle navigation and other functions. JSPs were reserved for display purposes only, and servlets were used to handle other functions, like performing calculations and validation. Developers also began creating tag libraries that combined XML and Java to generate markup language for user interfaces that had a sophisticated look and feel. Each JSP could have a Java class associated with it that was used to set values on the page and make the UI work correctly by enabling and disabling fields and buttons, as required.

Because the JSP technology conformed to standards, vendors were able to develop builder tools, such as JDeveloper, to visually manipulate the source files.

Dynamic Webpage Technologies: JavaServer Faces (JSF) 1.1 and 1.2



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

JSP technology offered some advantages over servlets and CGI because it provided a clear separation between presentation logic and business logic. However, developers needed an easier way to develop robust UIs, and they had to resolve some of the lingering problems of the JSP technology. For example, JSP continued to use a request-and-response model, instead of an event-driven model, and JSP was still heavily tied to markup language.

Although JavaServer Faces (JSF) is based on JSP technology, it is a radical departure from servlets and traditional JSPs. JSF is a component-based and event-based architecture, not a markup language-based architecture. Typically, a JSF page includes no HTML. Instead, it contains tags for UI components, such as buttons, labels, layout managers, headers, footers, tables, links, and so on. Each tag in the JSF page represents a server-side Java object.

The component-based architecture simplifies web development because it insulates developers from the implementation of the UI. Developers of various skill levels can quickly develop web applications by adding reusable UI components to a page, connecting the components to a data source, and wiring client-generated events to event handlers on the server. Instead of writing HTML, developers use tags to add components to a page, and they specify property settings that customize the behavior of the components. For example, instead of writing markup to highlight every other row in a table, developers bind data to a JSF table component and then set properties on that component to highlight every other row.

The tag libraries for the page are associated with a render kit. The render kit contains renderers that generate the markup language required by the target environment. This design enables developers to define pages that are not tightly coupled to the target environment. Instead, the pages can be rendered differently for different target environments.

When developers need to perform UI-specific operations or customize the behavior of components even further, they can create managed beans. A managed bean is simply a Java class that is registered with the JSF runtime and contains UI-specific code. As the name “managed bean” implies, its life cycle is managed by JSF. Developers can call methods, including getters and setters, on the bean to perform UI-specific operations and set bean properties. (You learn more about managed beans later in this lesson.)

Unlike JSP, JSF provides an event-driven model that is based on lifecycle phases; you can listen for and respond to specific events that occur at specific phases in the life cycle.

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. The JSF technology manages state, handles events and input validation, and defines page navigation. In this way, JSF hides the complexities of UI management, and simplifies the process of building rich client UIs. You learn more about the JSF life cycle, event handling, and navigation in a later lesson.

Because JSF is standards-based, it can be used with visual design tools like JDeveloper, which makes JSF accessible to a wide spectrum of programmer types.

Dynamic Webpage Technologies: JSF 2.0 Enhancements

JSF 1.2 Limitations

- ✗ Extra overhead is incurred during page rendering because JSP is compiled into an intermediate servlet .
- ✗ AJAX is implemented differently by each JSF component provider.
- ✗ Bookmarking is sometimes difficult to achieve due to the use of postbacks.
- ✗ Debugging is difficult because runtime errors are sometimes hard to trace.

JSF 2.0 Enhancements

- ✓ Performance is improved because facelets (the default view type) build the JSF component tree directly.
- ✓ Native AJAX support is built into the framework with support for specifying component behavior declaratively.
- ✓ Bookmarking is easier to achieve because GET request handling is supported.
- ✓ Debugging is easier because facelets provide more-detailed information, including the line number of the failed metadata definition on the page.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The initial releases of JSF resolved some problems inherent in JSP: The component-based JSF architecture simplified web development by insulating developers from the implementation of the UI. Likewise, the event-driven model hid the complexities of UI management and simplified the process of building rich client UIs. However, because JSF support was somewhat retrofitted into the existing JSP architecture, developers continued to face some limitations. For example, before version 2.0 of JSF, JavaServer Pages (JSP) was used as the default way to produce the JSF component tree at run time. Unfortunately, the JSP request life cycle, a simple parse-compile-render process, is not integrated with the more complex JSF request life cycle. Because of this lifecycle mismatch, developers who used HTML and JSTL tags in the context of JSF views defined within JSP had to deal with unpredictable results in page rendering. In addition, the need to compile the document into an intermediate servlet at run time incurred additional overhead.

JavaServer Faces 2.0 (JSF), the standard web user-interface technology in Java Platform, Enterprise Edition (Java EE) 6, addresses many of these problems and standardizes some features that were previously only available in frameworks that extended the JSF specification.

With the introduction of JSF 2.0, JavaServer Pages (JSP) is no longer the default way to produce the JSF component tree at run time. Instead, Facelets are the default document type. Unlike JSP documents, Facelets do not impose the unnecessary overhead of compiling the document into an intermediate servlet at run time; instead, facelets build the JSF component tree directly. This leads to far better performance in the component tree creation and page rendering processes.

JSF 2.0 also introduces several other enhancements that improve the developer's experience. For example, JSF 2.0 provide features like native integration of Asynchronous JavaScript and XML (AJAX) into the JavaServer Faces request life cycle, enhanced navigation, improved bookmarking capabilities, and improved error handling.

Comparison of Facelets and JSP XML Pages

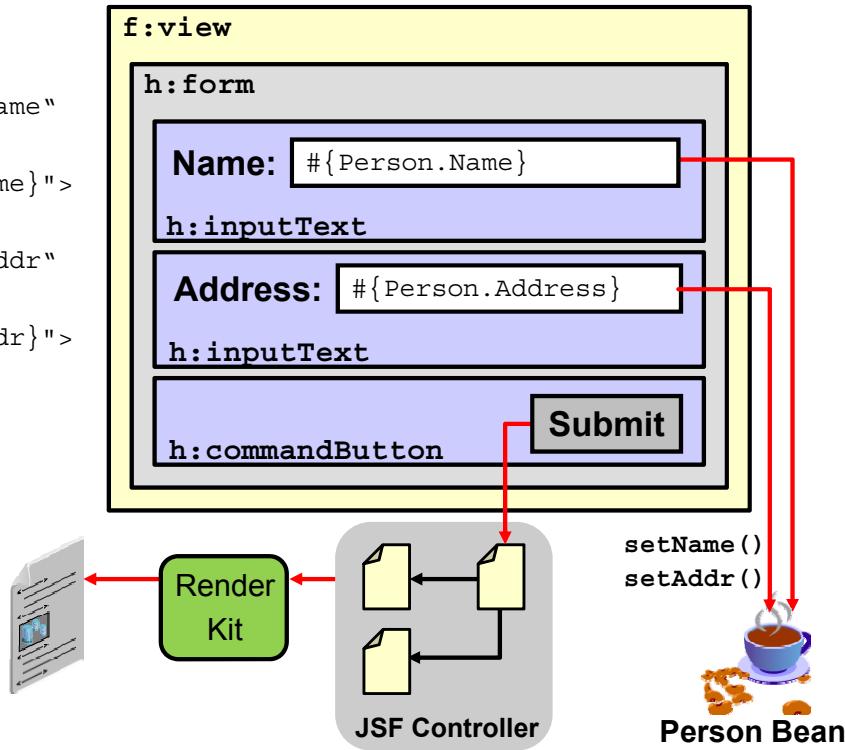
Facelets and JSP XML are both valid document types that you can use to build views in JSF 2.0. The key difference between the two technologies is that facelets handle JSF pages as XHTML documents that are partially processed by the server and partially processed by the browser. JSP documents handle JSF pages as XML metadata that is fully parsed and processed by the server-side JSP runtime engine.

Each page type offers various advantages:

- Facelets provide:
 - Cleaner integration with JSF. The facelets technology was designed from the ground up to target the JSF use case.
 - Reduced overhead and improved performance because no compilation is required. Facelets are parsed directly by the facelets engine, which avoids the need to translate the page definition into Java
 - Improved page templates. Facelets provide integrated page templating.
 - Improved error reporting. Runtime errors, such as EL evaluation exceptions, can be hard to trace because the JSP engine does not always provide clear guidance of where in the document the problem is located. Facelets provide more-detailed information, including the line number of the failed metadata definition on the page, making it easier to debug.
 - Better performance. Facelets do not need to be compiled into Java code at run time, which improves performance.
 - Support for JSF 2.0 features that are not available in JSP. Facelets are the recommended View Declaration Language for JSF.
- JSP documents provide:
 - An XML-based structure, which simplifies working with pages as well-formed trees of UI component tags
 - The ability to parse the page easily to create documentation or audit reports
 - Libraries that might not yet have an equivalent in facelets

The JSF Component Architecture: Overview

```
<f:view>
<h:form id="f1">
  <h:inputText id="Name"
    label="Name"
    value="#{Person.Name}">
  </h:inputText>
  <h:inputText id="Addr"
    label="Address"
    value="#{Person.Addr}">
  </h:inputText>
  <h:commandButton
    id="submit"
    action="success"
    value="Submit" />
</h:form>
</f:view>
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Let's examine the JSF component architecture in greater depth by looking at a simple example. The standard JSF model encompasses the following major concepts, which you examine in detail in the slides that follow: UI components, managed beans, expression language, JSF Controller, and renderers and render kits.

The JSF Component Architecture: UI Components

```
<f:view>
<h:form id="f1">
  <h:inputText id="Name"
    label="Name"
    value="#{Person.Name}">
  </h:inputText>
  <h:inputText id="Addr"
    label="Address"
    value="#{Person.Addr}">
  </h:inputText>
  <h:commandButton
    id="submit"
    action="success"
    value="Submit" />
</h:form>
</f:view>
```



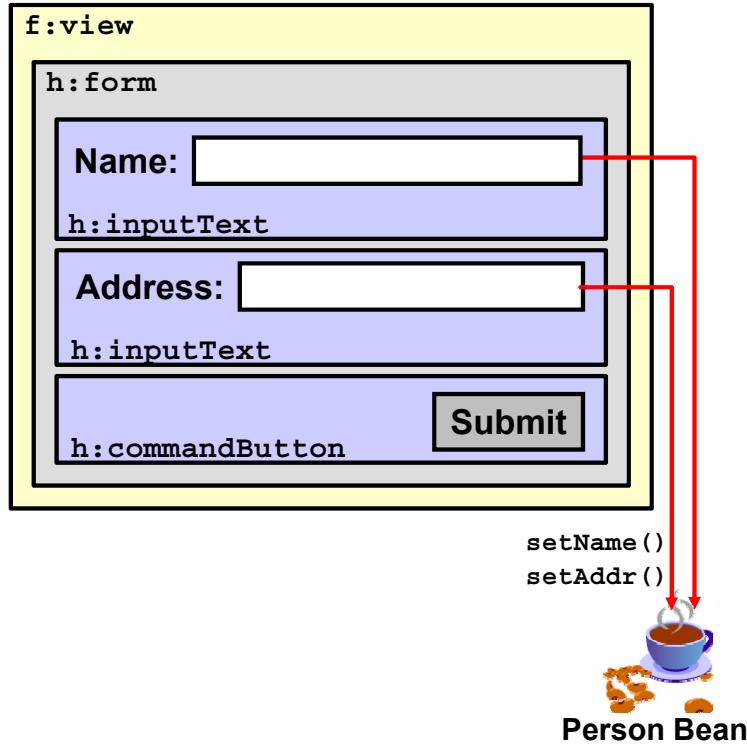
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

UI Components

You learned earlier that page developers use components, rather than mark-up tags, to build JSF views. The JSF technology provides tags for adding UI components (such as text boxes, labels, and check boxes) to the page, and core tags for using framework features like validation, data conversion, and event handling. The coding paradigm for UI components is based on the JavaBeans model. You set property values or attributes to define the component 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. The example shows a form that contains tags for the following components: two input text fields that include labels, and a Submit button. Each component has properties that further define the component. For example, each `h:inputText` tag includes a `label` attribute that specifies the text to display on the input text box. The tags are nested to reflect the structure of the page. You can easily add validation to a component by using one of the JSF standard validator tags (such as `f:validateLength`), or you can create a custom validator.

The JSF Component Architecture: Managed Beans

```
<f:view>
  <h:form id="f1">
    <h:inputText id="Name"
      label="Name"
      value="#{Person.Name}">
    </h:inputText>
    <h:inputText id="Addr"
      label="Address"
      value="#{Person.Addr}">
    </h:inputText>
    <h:commandButton
      id="submit"
      action="success"
      value="Submit" />
  </h:form>
</f:view>
```



ORACLE

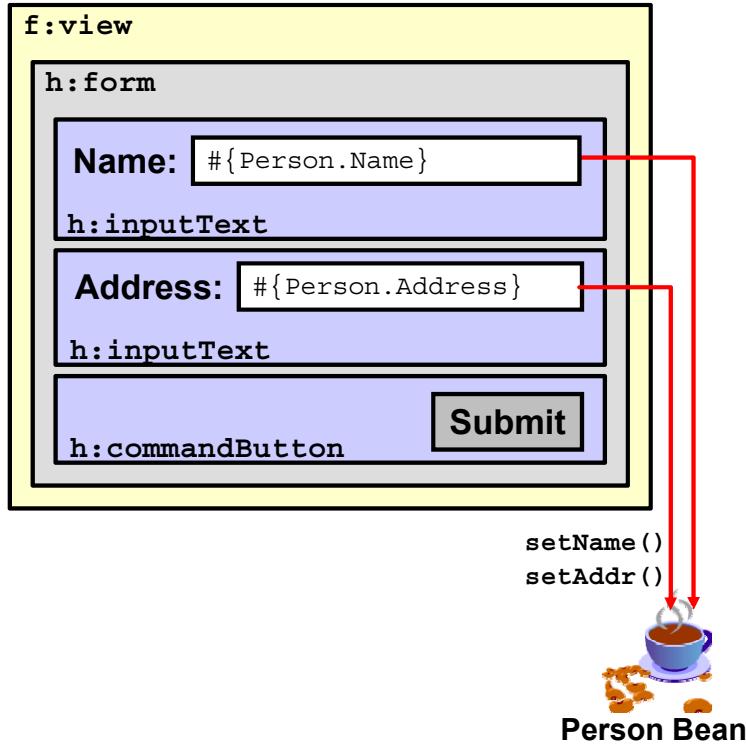
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Managed Beans and Backing Beans

Managed beans are Java objects that encapsulate UI-specific code and data. Managed beans are often used to validate data, handle events, store data between pages, and perform navigation. Managed beans are simply Plain Old Java Objects (POJOs) that are registered with the JSF container and include a no-argument constructor; managed beans do not inherit from a specific base class. For example, in the slide, the values of the Name and Address fields are stored in the managed bean, Person. Managed beans have a scope that determines how long they are held in memory. A backing bean is a special use of a managed bean that is associated with a specific page; the bean “backs” the page. For example, a backing bean might contain logic that disables or enables a field based on specific criteria.

The JSF Component Architecture: Expression Language

```
<f:view>
<h:form id="f1">
  <h:inputText id="Name"
    label="Name"
    value="#{Person.Name}">
  </h:inputText>
  <h:inputText id="Addr"
    label="Address"
    value="#{Person.Addr}">
  </h:inputText>
  <h:commandButton
    id="submit"
    action="success"
    value="Submit" />
</h:form>
</f:view>
```



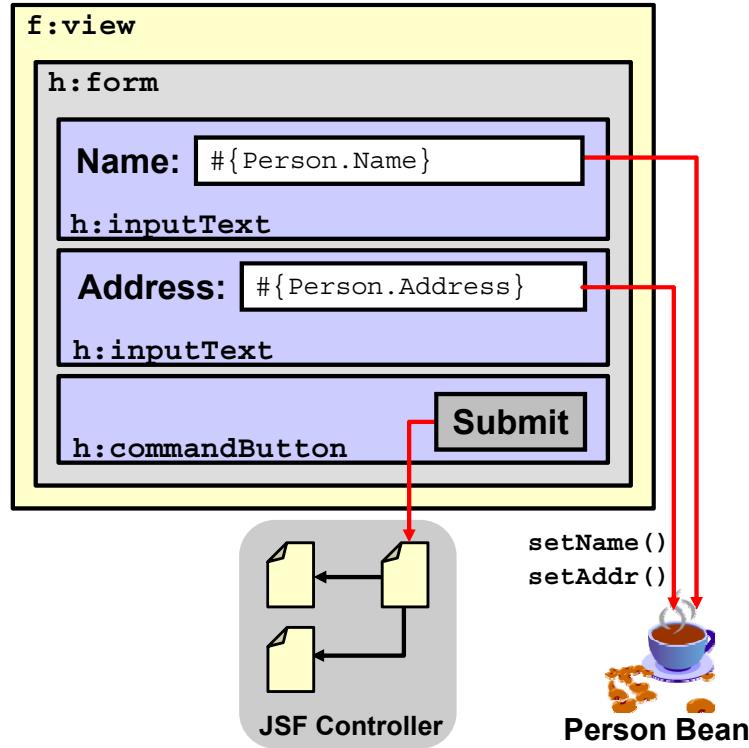
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Expression Language

Expression Language (EL) is a scripting language that enables the presentation layer (webpages) to communicate with application logic (such as managed beans) through a simple syntax. Expression language provides the ability to get and set property values, and to invoke methods on a bean. A typical EL expression uses the following syntax to reference a bean property or method: `# { BeanName.propertyName }` or `# { BeanName.methodName }`. For example, in the slide, the value attribute for the Name input text field contains the expression `# { Person.Name }`. This expression binds the value of the Name field to the Name property in the Person bean. When the page is rendered, the JSF runtime automatically calls the `getName()` method on the Person bean to display the value for the Name property. Likewise, when the user enters a value and submits the form, the JSF runtime automatically calls the `setName()` method on the Person bean to update the property.

The JSF Component Architecture: JSF Controller

```
<f:view>
<h:form id="f1">
  <h:inputText id="Name"
    label="Name"
    value="#{Person.Name}">
  </h:inputText>
  <h:inputText id="Addr"
    label="Address"
    value="#{Person.Addr}">
  </h:inputText>
  <h:commandButton
    id="submit"
    action="success"
    value="Submit" />
</h:form>
</f:view>
```



ORACLE

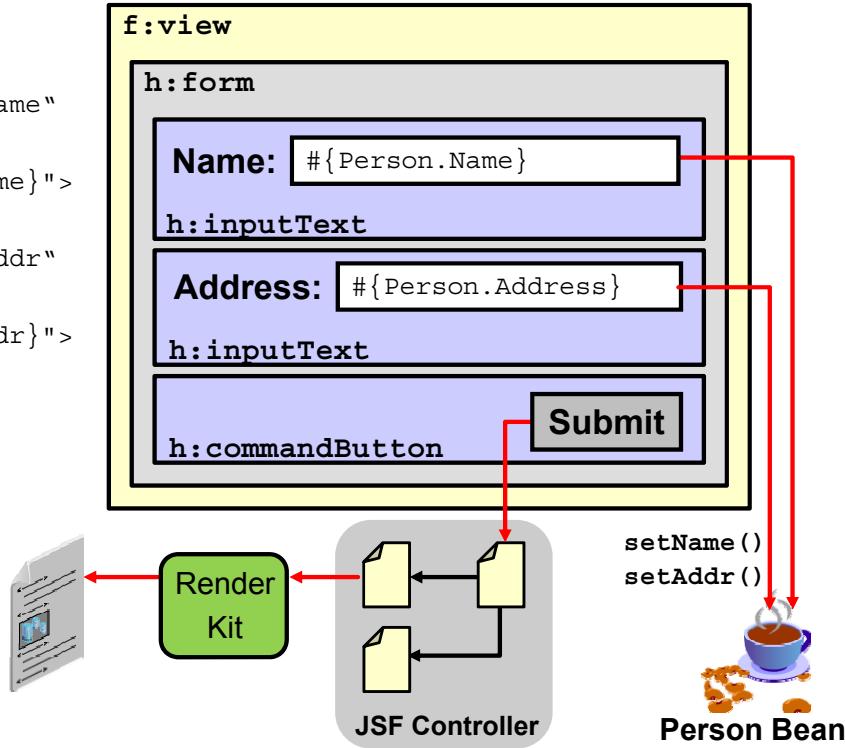
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

JSF Controller

The JSF Controller contains navigation rules that define how the user progresses through the application. When a user performs an action (such as clicking a link or submitting a form), an action handler executes and returns an outcome in the form of a string value. For example, in the example in the slide, when the user clicks the Submit button, the outcome is set to "success." The controller uses the outcome to determine which page to load next.

The JSF Component Architecture: Renderers and Render Kits

```
<f:view>
<h:form id="f1">
  <h:inputText id="Name"
    label="Name"
    value="#{Person.Name}">
  </h:inputText>
  <h:inputText id="Addr"
    label="Address"
    value="#{Person.Addr}">
  </h:inputText>
  <h:commandButton
    id="submit"
    action="success"
    value="Submit" />
</h:form>
</f:view>
```



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

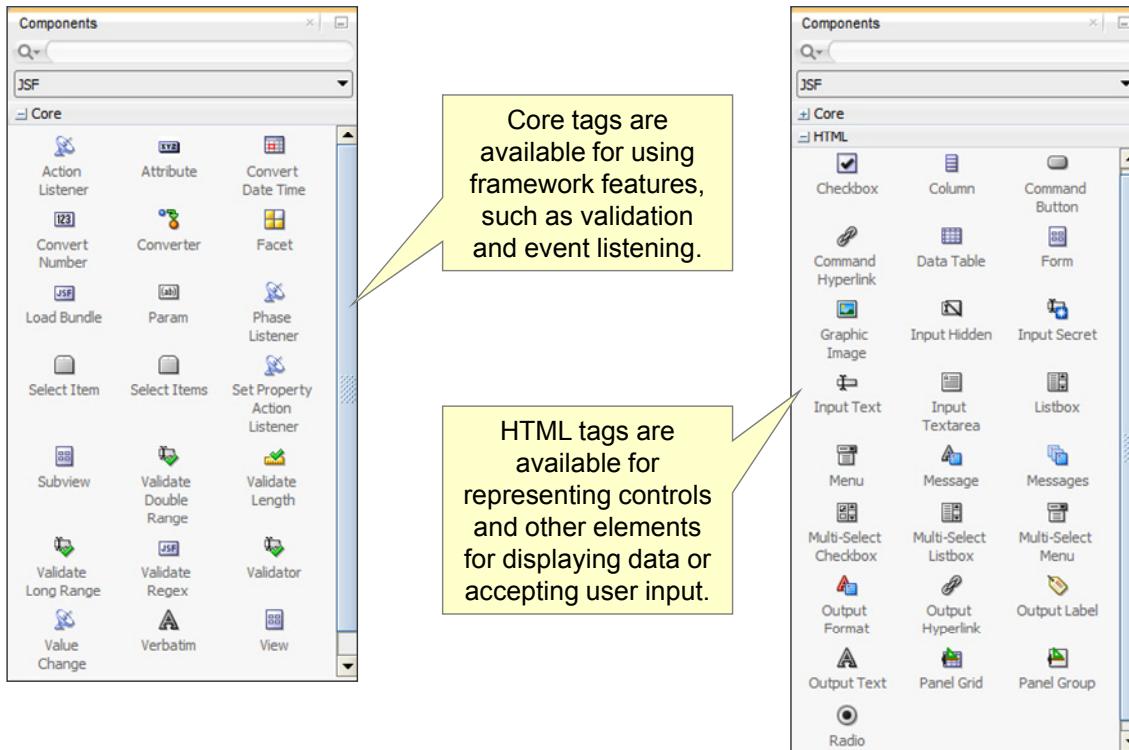
Renderers and Render Kits

The JSF component architecture is designed so that the behavior of the components is defined by UI component classes, and the component rendering is defined by separate classes, called renderers. JSF renderers reside on the server and are responsible for displaying the UI components in a graphical representation that is understood by the client. Renderer classes are packaged into libraries, called “render kits,” that enable you to develop a JSF application for different client devices without altering application code. A render kit for an HTML client is part of standard JSF.

Runtime Behavior of JSF Components

When a browser submits an HTTP request to access a page for the first time, the web server parses the URL and forwards the request to the JSF Controller servlet, which loads the page into memory. The servlet reads the page, instantiates (creates) a new object for each tag on the page, and builds a tree of all components on the page. The controller servlet calls the rendering engine to render the page. The rendering engine reads the component tree and generates a stream of HTML to send back to the browser. The empty form appears on the screen. When the user enters values and submits the form, the browser posts the page to the web server. The web server parses the URL and forwards the request to the controller servlet. The controller servlet pulls the data out of the request, validates the data, executes methods in the managed beans, reads navigation rules to determine the next page to load, and then rebuilds the component tree. The rendering engine generates markup for the new page and sends a stream of HTML back to the browser.

Standard JSF Components



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

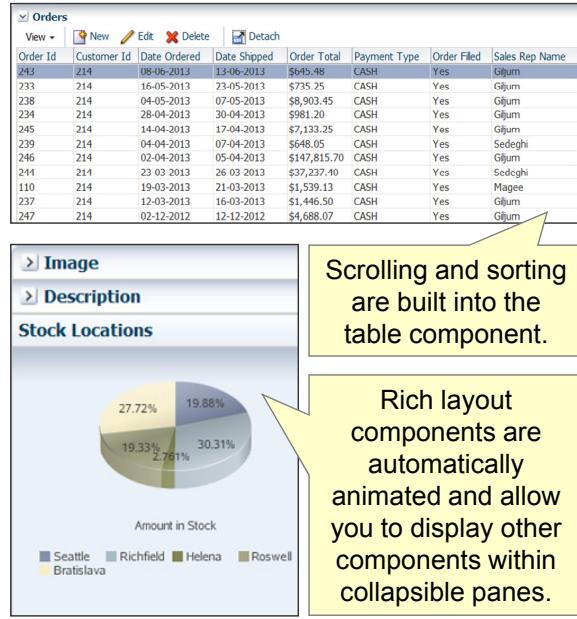
The standard JSF implementation includes the following main tag libraries:

- **JSF Core tag library:** Used with other components to perform core actions that do not depend on a particular render kit. The tags start with `f:`. The library includes such elements as listeners, converters, validators, and subcomponents, such as list items. It also includes the `f:view` tag, which must wrap all the child tags used on a page to ensure that the tags are part of the same view.
- **HTML tag library:** Used to represent form controls and other basic HTML elements that display data or accept user input. The tags start with `h:`. The library includes such elements as buttons, links, messages, tables, check boxes, graphics, input and output text, and menus.

The standard JSF tags provide the basic functionality that you need to build a web-based UI that displays dynamic data. However, the components do not provide the dynamic features that users have come to expect in a rich client UI. For example, you can use the standard JSF components to display data in a table format; however, if you want to provide a sortable, scrollable table, you must implement the logic required to cache and sort the data yourself, if you are using the standard JSF component set.

ADF Faces Rich Client Components

- Built on top of JSF APIs
- Over 150 AJAX-enabled components
- More advanced and interesting components
 - Partial-page rendering
 - Scrollable, sortable tables
- Rich feature set for customizing applications
- ADF Model support
- Runs on any JSF-compliant implementation



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle ADF Faces is a rich set of user-interface components that are built on top of standard JSF. Oracle ADF Faces uses the flexible JSF rendering architecture to combine the power of AJAX and JSF. AJAX, which stands for Asynchronous JavaScript and XML, is a group of related client-side technologies that enable webpages to send and receive data asynchronously without interfering with the display or behavior of the page. While AJAX allows rich client-like applications to run on standard Internet technologies, JSF provides server-side control, which reduces the dependency on an abundance of JavaScript often found in typical AJAX applications.

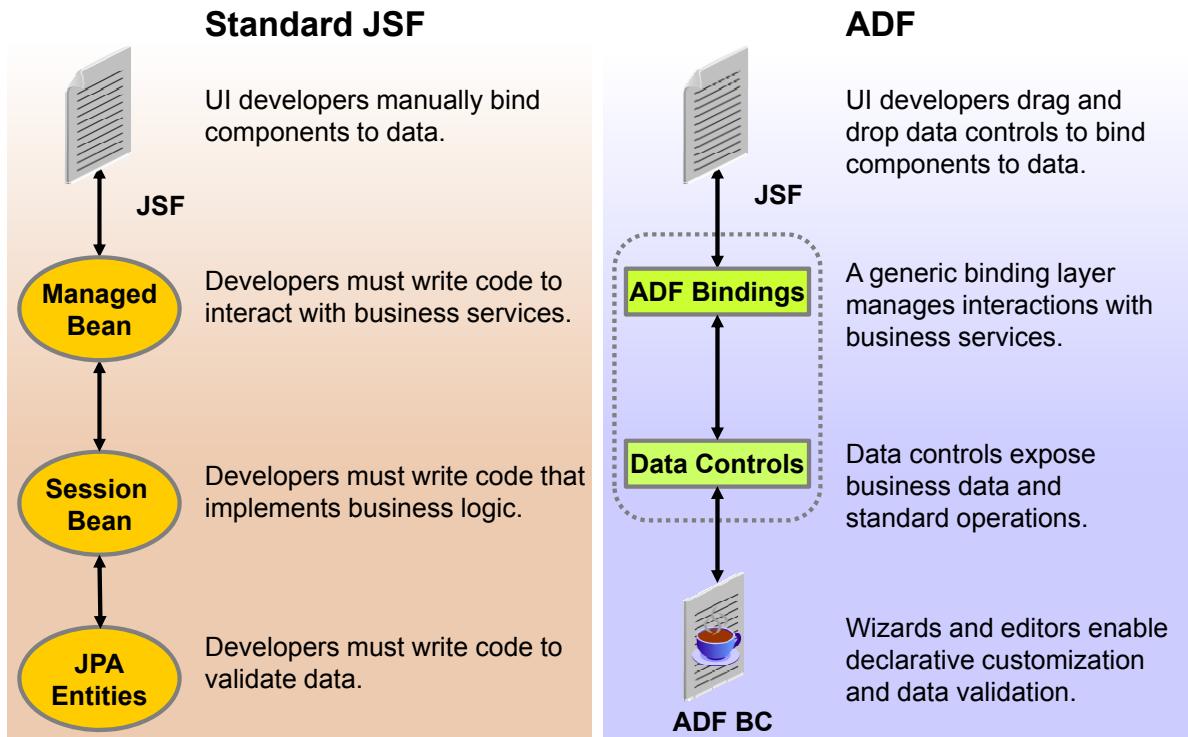
ADF Faces provides over 150 components with built-in functionality—such as data tables, hierarchical tables, and color and date pickers—that you can customize and reuse in your application.

For example, ADF Faces provides an enhanced table component with scrolling and sorting built in; you simply select the appropriate check boxes to enable these capabilities when you add the table component to a page in JDeveloper. In contrast, with the standard JSF table component, you are responsible for implementing the logic that caches and sorts the data.

Each ADF Faces component offers complete customization, skinning, and support for internationalization and accessibility. ADF Faces also provides a rich set of visualization components that are capable of rendering dynamic charts, graphs, gauges, and other graphics that provide real-time updates. The visualization components have AJAX functionality built in, which means that you do not have to code animations and interactivity into the component; you simply use ADF Model to bind the component to the data that it will display, and then you specify properties of the component to configure how the component behaves.

ADF Faces components can be used in any IDE that supports JSF.

Standard JSF and ADF Compared



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ADF Faces components are designed to work with the ADF Model. To understand the value and time savings that the ADF Model layer provides, let's compare a JSF application that accesses business services through a managed bean, and an ADF application that accesses business services through the ADF Model.

The left half of the slide shows an application that uses standard JSF. The developer binds UI components to properties or methods that are implemented in a managed bean. (Remember that a managed bean is simply a Java class that is registered with the JSF runtime and contains UI-specific code.) The managed bean is responsible for managing interactions with the business services layer by calling business methods that are defined in a session bean. (A session bean is an enterprise bean that encapsulates business logic that clients invoke programmatically.) For example, a managed bean might call business methods in the session bean to get and set property values, iterate through result sets, define lists of values (LOVs), and handle exceptions. The session bean implements business logic, such as calling queries to retrieve result sets, performing calculations, persisting changes, and committing or rolling back transactions. To persist changes and execute database queries, methods in the session bean run named queries that are defined in JPA entities (or another persistence technology). The JPA entities provide getters and setters (accessors) for modifying values, and they provide an object-relational mapping to database tables.

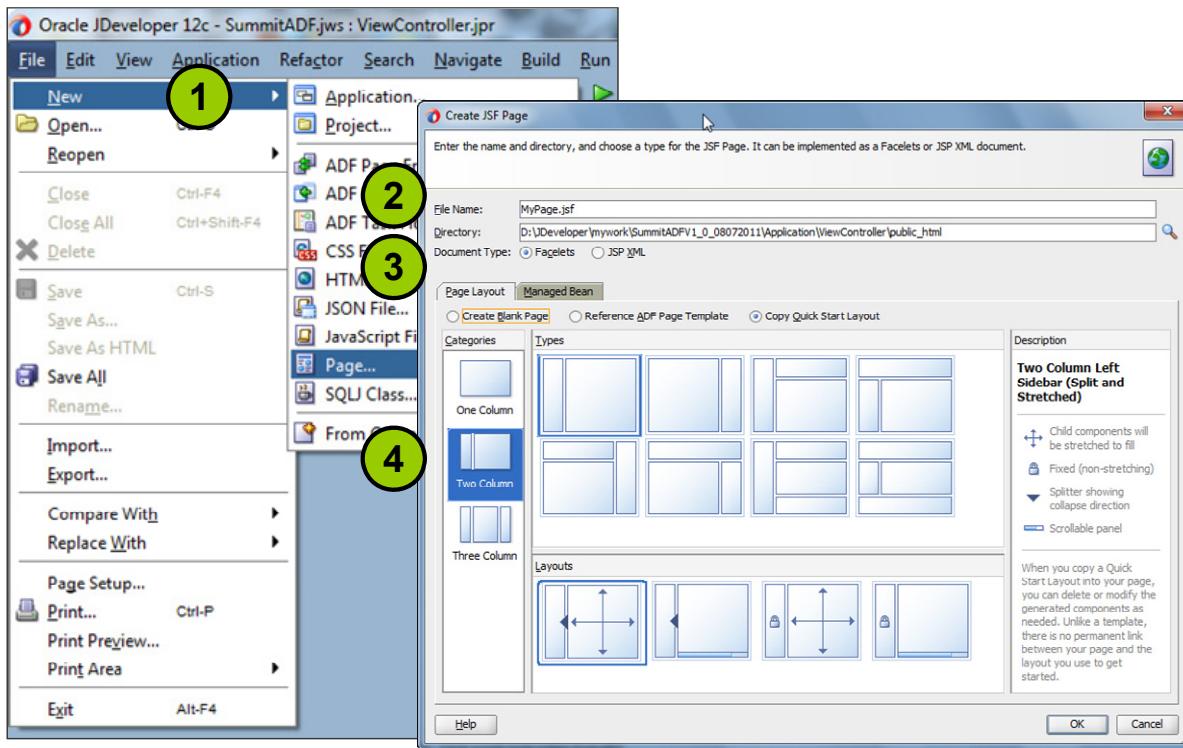
The right half of the slide shows an application that uses the ADF Model. At design time, page developers drag and drop data controls onto a page. The data controls provide a standard metadata interface that exposes the business service's data collections, attributes, and methods, as well as built-in operations that are generic to all data collections. When a data control is dropped onto a page to create a UI component, JDeveloper automatically generates ADF bindings for the component. The ADF bindings manage the interaction between the UI component and the business service. This design further decouples the user interface from the data portions of the application.

You learn more about ADF bindings and data controls in a later lesson. For now, you just need to understand that the ADF Model implements much of the functionality required to interact with the business services layer, and it provides drag and drop controls for wiring UI components to business services by writing little or no Java code.

In the example, the business services layer uses ADF Business Components (ADF BC). As you learned earlier, ADF BC handles all interaction between your application and the data stored in the data source. When you use ADF BC, instead of writing code to interact with the data source and perform validation, you set properties and define logic declaratively by using wizards and editors.

The business services layer can also contain non-ADF business components, such as EJBs. In this situation, data controls provide an additional function: they contain a layer of metadata that supports the ability to customize attributes. For example, you can edit the data control definition and specify UI control hints, define validation, and create LOVs in the same way that you would for an ADF business component.

Creating a JSF Page in JDeveloper



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Now that you understand the technologies that are available for building the view layer, let's learn how to build a simple page.

To create a new page in JDeveloper:

1. Select the ViewController project, and select File > New > Page. (You can also create a new page from the New Gallery.)
2. Specify a name and location for the file (or accept the default directory). The JSF page must exist in the ViewController project of your application.
3. For the document type, select either Facelets or JSP XML. Each page type offers some advantages (as described earlier in this lesson).
4. Select a page layout. You can create a blank page, specify a page template to use (you learn about templates later), or use a quick-start layout. Quick-start layouts provide a quick and easy way to build the layout correctly. You can choose from one, two, or three column layouts, and then determine how you want the columns to behave. For example, you might want to lock the width of one column, while another column stretches to fill available browser space.

Exposing UI Components in a Managed Bean

On the Managed Bean tab, you can choose to automatically expose UI components in a new or existing managed bean. If you select one of these options, JDeveloper creates a backing bean (or uses an existing one). When you drop a component on the page, JDeveloper inserts a bean property for each component, and uses the binding attribute to bind component instances to those properties, allowing the bean to accept and return component instances.

Example: Two Column Layout

The screenshot shows the Oracle JDeveloper interface. On the left, there are three tabs: 'Wizard' (highlighted in red), 'Visual Editor' (highlighted in green), and 'Source View' (highlighted in blue). In the 'Visual Editor', a 'Layouts' palette is open at the top, showing four layout options. The first option, 'Two Column Left Sidebar (Split and Stretched)', is selected and highlighted with a red border. A red arrow points from this selection down to the visual editor area. The visual editor shows a horizontal splitter bar with two facets: 'first' on the left and 'second' on the right. In the 'Source View' tab, the generated XML code for this layout is displayed:

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html>
<f:view xmlns:f="http://java.sun.com/jsf/core" xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
    <af:document title="untitled1.jsf" id="d1">
        <af:form id="f1">
            <af:panelSplitter orientation="horizontal" splitterPosition="100" id="ps1">
                <f:facet name="first"/>
                <f:facet name="second"/>
            </af:panelSplitter>
        </af:form>
    </af:document>
</f:view>

```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide shows an example of the visual editor and source code view for a JSF page that uses the quick layout called “Two Column Left Sidebar (Split and Stretched)”:

- The `f:view` tag is a container for all the tags that are used in the view.
- The `af:document` tag contains all other components that make up the page. The `af:document` component is used at run time to create the root elements for the client page. For example, for HTML output, the `af:document` component creates each of the standard root elements of an HTML page: `<html>`, `<head>`, and `<body>`.
- The `af:form` tag creates an HTML `<form>` element. Because dynamic webpages typically contain form elements, JDeveloper adds this tag for you.

Because you used a quick layout, the page also includes layout components. Layout components are designed to help you achieve the desired page layout.

- The `af:panelSplitter` tag divides the page into two parts with a repositionable divider.
- The `af:panelSplitter` tag contains two `af:facet` tags: the first facet, which holds the contents of the repositionable column, and the second facet, which holds the contents of the stretchable column. A facet is simply a named section within a container component. You nest additional layout components within facets to build user interfaces according to your specifications.

You learn more about layout components in a later lesson. For now, you need to understand that there is a difference between layout components, which arrange other components on the page, and data-bound UI components, which handle the dynamic display and persistence of data.

Next, you learn how to add data-bound UI components to a page.

Adding UI Components to the Page

You can add UI components to a page by using a variety of methods.

- Dragging a data element from the Data Controls panel to the visual editor, source editor, or Structure window
- Dragging a component from the Components window to the visual editor, source editor, or Structure window
- Using the context menu in the editor or Structure window
- Typing code directly into the source editor



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Several methods are available for adding UI components to the page. The method that you use depends on your personal preferences for working with the IDE and whether you are creating layout components or data-bound components.

You can add UI components to a page by:

- Dragging a data element from the Data Controls panel to the visual editor, source editor, or Structure window. When you use this approach, JDeveloper automatically binds the component to the data or operation exposed by the data control. Therefore, if you are creating data-bound components, you want to use this approach.
- Dragging a component from the Components window to the visual editor, source editor, or Structure window. When you use this approach, you can create a component that has no bindings, and you specify the bindings later. For example, you might decide to create an unbound component when you want to do UI prototyping.
- Using the context menu in the visual editor or Structure window. When you use this approach, you can select an existing component, and choose to insert the new UI component before, inside of, after, or around the selected component. The advantage of using this approach is that you can select from a list of components that are valid at the specified insertion point.
- Typing the XML for the component directly into the source editor.

Using the Data Controls Panel

- Provides a visual representation of your business service:
 - Methods
 - Query parameters and results
 - Attributes
 - Collections
 - Built-in operations
- Provides automatic data binding for business services

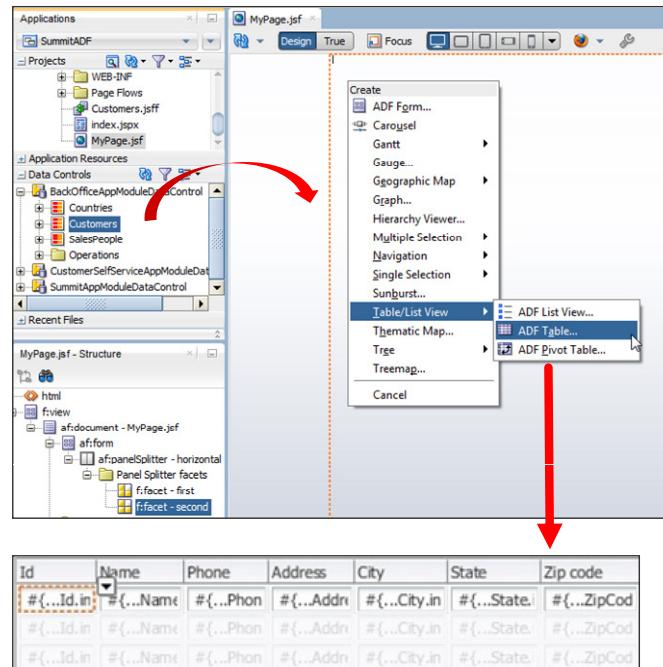


Table in the Design View

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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, and operations that are available for binding to UI components. Each root node in the Data Controls panel represents a specific data control. Under each data control is a hierarchical list of objects, collections, and operations.

The Data Controls panel provides a convenient mechanism for automatically creating data-bound 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. For example, dragging a data control that represents a collection to the page gives you the choice of creating the component as a diagram, form, graph, table, and so forth. After you select a component from the pop-up list, JDeveloper automatically creates the JSF page component code and the XML binding code needed to access the data control object. JDeveloper also defines the properties of the new UI component using EL to refer to the bindings.

Tip: Whenever you update a business component, remember to click the Refresh button in the Data Controls panel to see your changes.

Example: Input Text Component with Bindings

- Data-binding expressions are written in EL.
- They are evaluated at run time to determine what data to display.
- ADF EL expressions typically have the form:
`# {bindings.BindingObject.propertyName}`

The diagram illustrates the relationship between the rendered component and its source code. At the top, a screenshot of a web browser shows an input text field with the placeholder text "* Name". Below this, a section titled "Component Rendered in HTML" contains the rendered input field. At the bottom, a section titled "Source Code for Component" contains the ADF binding code. Red underlines highlight specific EL expressions in the source code, which are then connected by arrows to the corresponding parts in the rendered component. The underlined expressions include `value="#{bindings.Name.inputValue}"`, `label="#{bindings.Name.hints.label}"`, and `required="#{bindings.Name.hints.mandatory}"`.

```
<af:inputText value="#{bindings.Name.inputValue}"
    label="#{bindings.Name.hints.label}"
    required="#{bindings.Name.hints.mandatory}"
    id="it1">
</af:inputText>
```

Bindings in EL

Source Code for Component



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you use the Data Controls panel to add 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. You can add ADF data-binding expressions to any component attribute that you want to populate with data from a binding object.

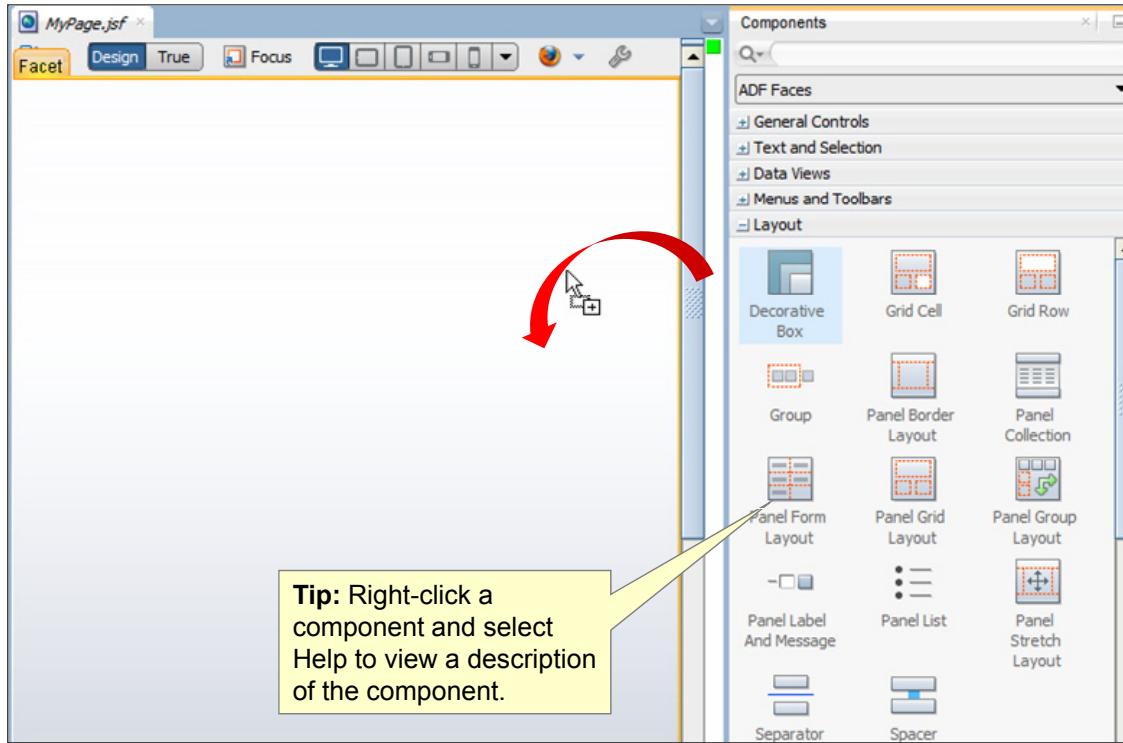
ADF binding expressions are written in Expression Language (EL). As you learned earlier, EL is a scripting language used in JSF pages that enables the view layer (webpages) to communicate with application logic through a simple syntax. The syntax for accessing business services through the ADF binding layer is slightly different than standard JSF. A typical ADF data binding EL expression uses the following syntax:

`# {bindings.BindingObject.propertyName}`.

Instead of referencing a managed bean, the expression references a binding object in the binding container for the page. You learn more about binding objects and the binding container in a later lesson. For now, you need to understand that:

- ADF bindings manage interactions between UI components and business services.
- When you drag and drop data controls to a page, JDeveloper generates bindings for you automatically.
- ADF bindings are written in expression language.

Using the Components Window



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Components window shows all the components that are available for building the user interface. To add components to a JSF page, you drag and drop them to the page in the visual editor or Structure window. By default, the Components window displays the library of components that are relevant to the type of file that you are editing. For example, when you edit a JSF page, the Components window contains ADF Faces components. You can select from a drop-down list to display other libraries, such ADF Data Visualizations, ADF Faces, CSS, Facelets, HTML, and JSF.

To make it easier to find specific components, the ADF components are grouped by types:

- **General Controls:** Components for adding controls like buttons, images, and links to a page
- **Text and Selection:** Components for displaying, entering, and selecting text, such as check boxes, selection lists, and text fields
- **Data Views:** Components for displaying collections of data, including tables, trees, and carousels
- **Menus and Toolbars:** Components for building menus and toolbars
- **Layout:** Components for laying out other components on a page, including panels, boxes, and grids

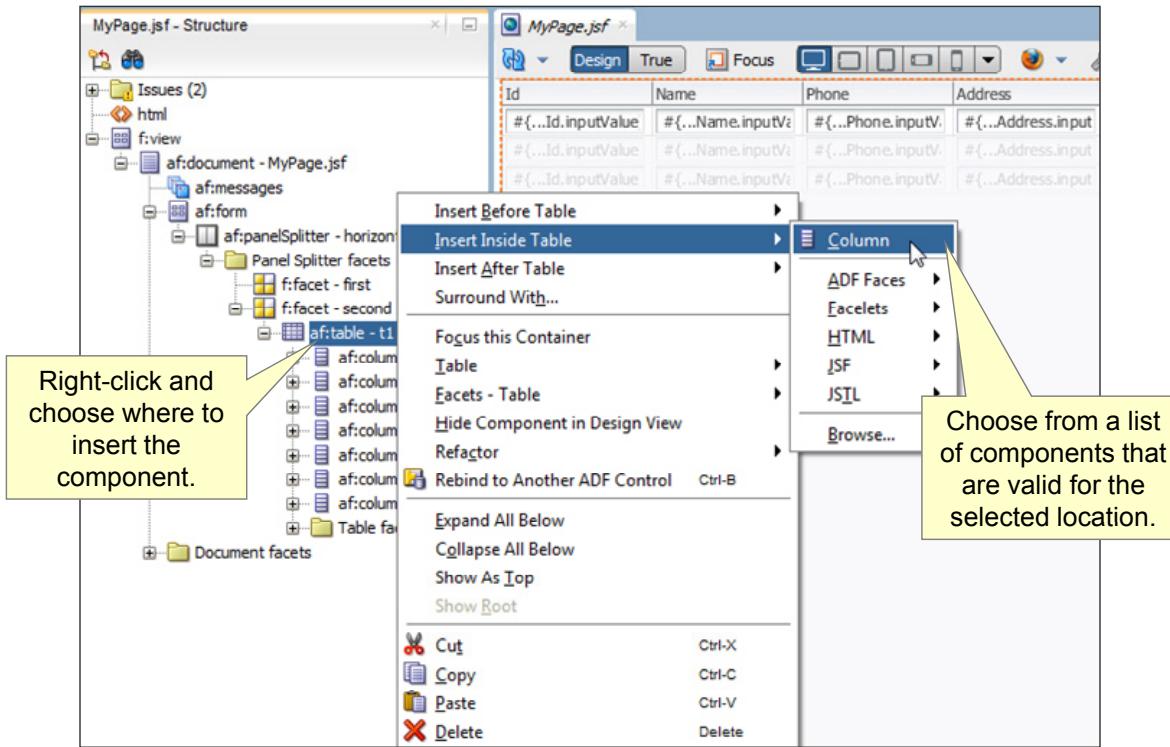
- **Operations:** Components for performing operations, such as listening for events and performing validation

To search for a component by name, enter the name or part of the name in the Find field and press Enter.

By default, components are displayed in an icon view with labels. You can change the display to show icons only or to show a list view (a list of icons with labels). To change the view, right-click in the Components window, and select the view you want to display. From the context menu, you can also view help on a component or add tag libraries.

When you use the Components Window to add components to a page, the bindings are not generated for you automatically. You must bind the components to data manually.

Using the Context Menu



ORACLE

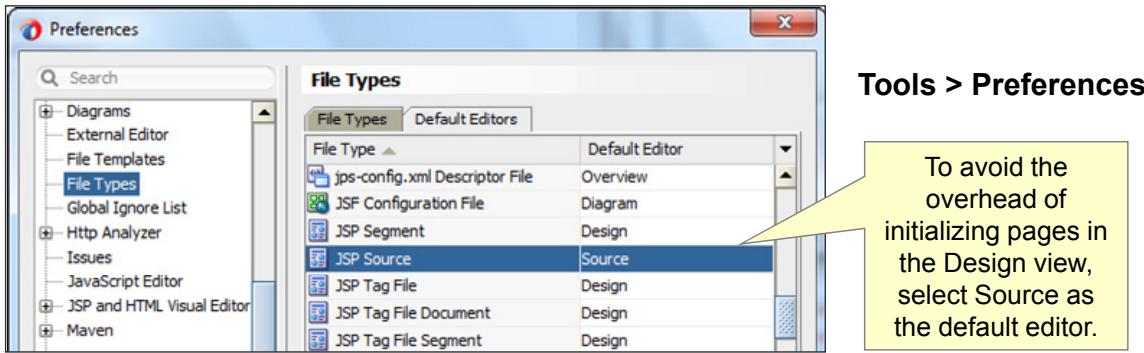
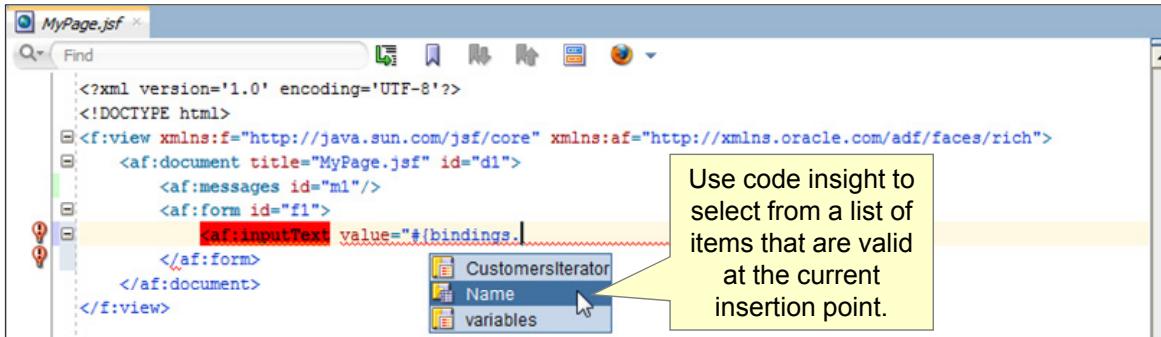
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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 selected one
- Surround the selected component with another component

The context menu contains a list of components that are valid for the selected location. You can browse to locate other components.

Using the Code Editor



ORACLE

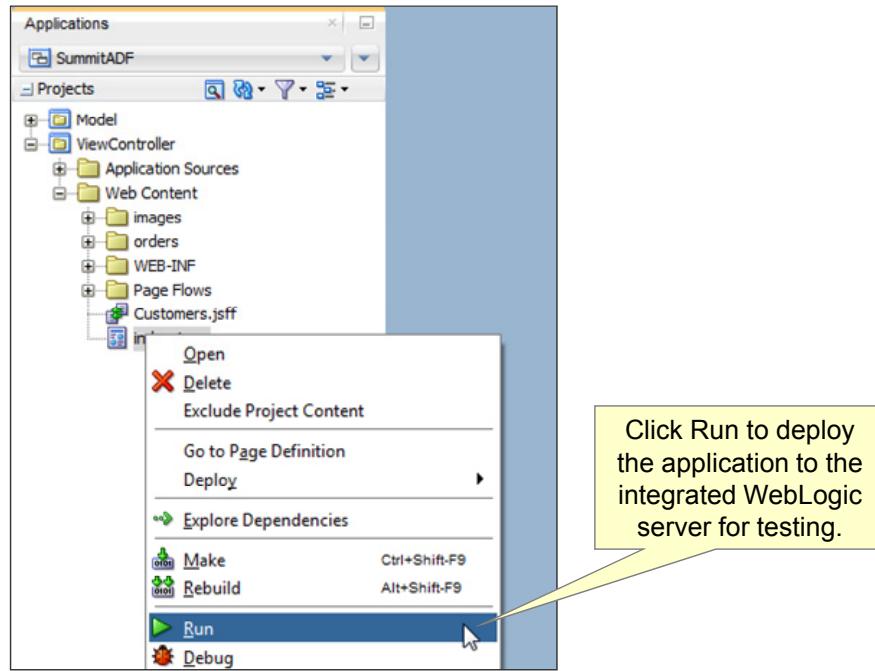
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Within the source code editor, you can add components by dragging and dropping data controls or components to page (just as you do in the Design view). Alternatively, you can add components by typing the XML code directly into the source editor.

When you type code directly into the source editor, JDeveloper provides code insight. With code insight, you can type a name followed by a dot (or press Ctrl + Alt + Space) and select from a list of items that are valid at the current insertion point.

Tip: To avoid the overhead of initializing pages in the Design view, you can set JDeveloper preferences to open specific file types (such as JSP Source and XHTML source) in the Source view. To specify the type of editor to use, select Tools > Preferences > File Types and click the Default Editors tab.

Running and Testing the Page



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

JDeveloper provides an integrated WebLogic application server for running and testing pages. To run the JSF page, in the Applications window under Projects, right-click the page and select Run. When you run the page, JDeveloper:

- Saves all unsaved files
- Compiles the source files and creates the application JAR, WAR, and EAR files
- Starts the embedded application server
- Deploys the application's JAR, WAR, and EAR files to the application server
- Deploys connection information about data sources defined for the application
- Launches the application in your default browser, where you can test the application

Previewing Changes

If you simply want to verify the layout of the page, you can use the Browser Preview button in the visual designer to avoid the overhead of deploying the application. The preview shows static information only. Therefore, dynamic content does not display in the preview, and the UI does not respond to events, such as mouse clicks. The preview shows only areas that are visible in the visual editor. If your page contains multiple tabs or areas that expand and collapse, make the area that you want to preview active in the visual editor before you click the Browser Preview button.

Modifying and Rerunning a Page

The hot deploy feature in JDeveloper allows you to modify most aspects of a page without having to redeploy (rerun) the application to see your changes. For example, you can change the value of a text label, change the layout of components, and even add new data-bound components to the page without redeploying the application. You simply save the changes and rebuild the project. However, you must redeploy the application when you make changes that are incompatible with the current application state. For example, you must redeploy the application when you make any of the following changes:

- Remove a key attribute in a view object or an entity
- Rename an attribute
- Change the type of attribute
- Modify the where clause for a view object

Summary

In this lesson, you should have learned how to:

- Define JavaServer Faces and the JSF component architecture
- Explain how JSF managed beans are used in an application
- List some JSF component types included in the standard implementation
- List some ADF Faces component types that extend the basic JSF component types
- Explain the purpose of a data control
- Decide between using pages based on Facelets or JSP XML
- Create a JSF page that contains data-bound components



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 4 Overview: Creating and Running a JSF Page

This practice covers the following topics:

- Creating a simple, stand-alone JSF page
- Adding data-bound components to the page, including a table, a form, and a query
- Adding buttons and links to the page
- Running and testing the page



Copyright © 2014, 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 NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

Defining Task Flows and Adding Navigation



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

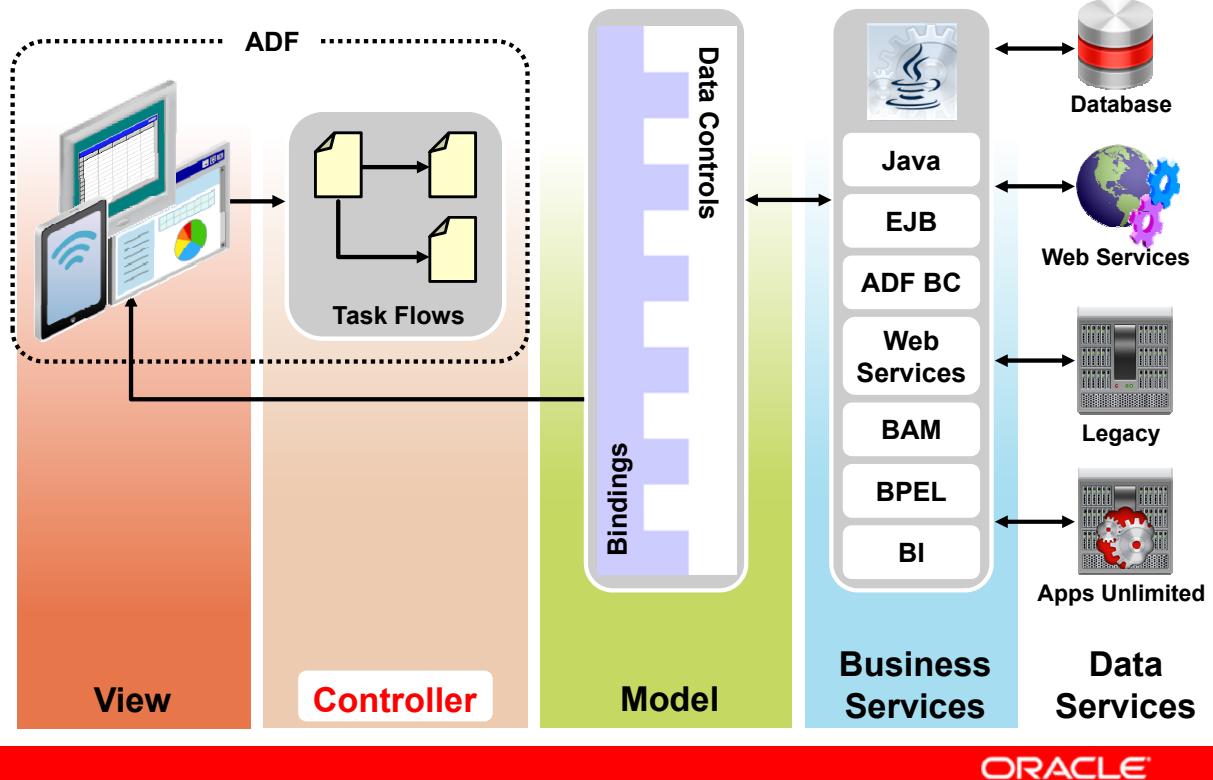
After completing this lesson, you should be able to:

- Explain how ADF extends the capabilities of the JSF controller
- Create task flows that include views, control flows, and global navigation
- Implement navigation using buttons and links



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Building the Controller Layer



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the previous lesson, you learned how to build the view layer of your application. You learned about the evolution of web technologies from the early design of dynamic web pages through to newer Java-based technologies. Then, you learned about ADF Faces and the underlying JavaServer Faces technology on which ADF Faces is built. You also learned about ADF Faces UI components and how to use data controls to build data-bound UI components.

In this lesson, you learn the fundamentals of building the controller layer. First, you learn about the ADF Controller and how it extends the capabilities of the JSF Controller. Then, you learn about task flows and how you use them to define control flow in an application. Finally, you learn about some common navigation components, such as links and buttons, that are used to trigger navigation between pages.

This lesson introduces you to the basic concepts that you need to understand to build the controller layer of your application. You learn more about task flows and navigation controls in a later lesson.

Traditional Navigation Compared to JSF Controller

Hard-Coded Buttons:

```
<body>
  <form action="NewPage.html">
    <button type="button"/>
  </form>
</body>
```

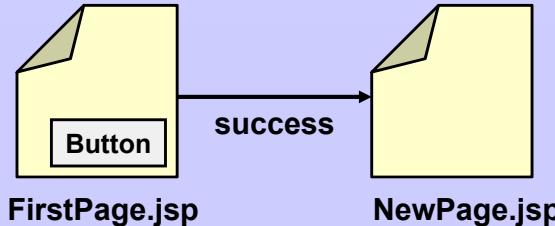
Hard-Coded Links:

```
<body>
  <a href="http://NewPage.html">
    Go To New Page </a>
</body>
```

Traditional Navigation

Navigation Rules in faces-config.xml:

```
<navigation-rule>
  <from-view-id>FirstPage.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>NewPage.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```



JSF Controller



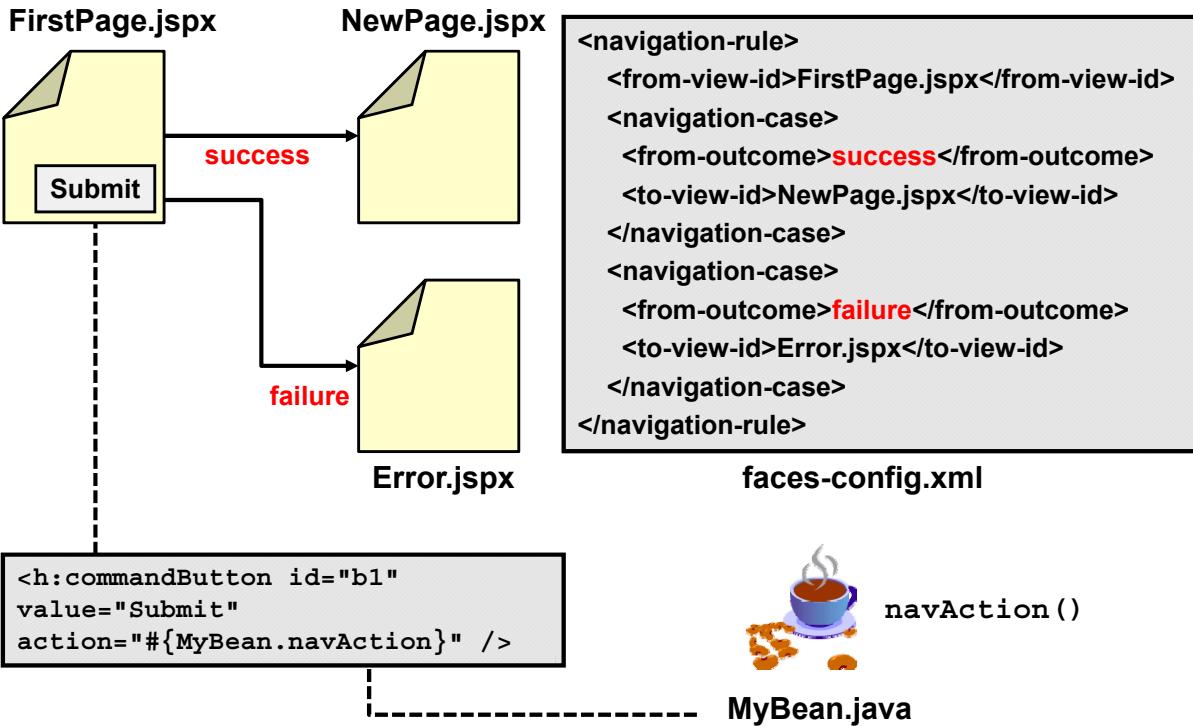
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To understand how navigation works in an ADF application, let's first look at how basic JSF navigation works before learning about the ADF Controller.

With traditional navigation (also referred to as Model 1 type), a button or hyperlink is hard-coded to navigate to a specific page. The examples in the slide show the HTML coding for a form and a link that are hard-coded to point to a page called `NewPage.html`. If the physical location or name of the page changes, you must modify all links and forms that point to that page.

With the JSF Controller (or any Model 2 type implementation of MVC), you avoid the maintenance nightmare of updating multiple pages when the location or name of a page changes. Instead of hard-coding links, you define navigation rules in a separate XML file. In JSF applications, the navigation rules are stored in the `faces-config.xml` file. Each navigation rule has one or more navigation cases that determine the next page to display based on a specific outcome. 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 a different page.

JSF Navigation: Example



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example, a page named `FirstPage.jspx` contains an `h:commandButton` component that renders in HTML as a submit button. The component has three attributes:

- **id:** The ID of the component. In the example, the ID is “`b1`.”
- **value:** The label to display when the button renders. This value can be a string, or an EL expression that obtains the value from a resource bundle. In the example, the value is “Submit.”
- **action:** The logical outcome of clicking the Submit button. This value can be a string, or an EL expression that calls a method in a managed bean to return a string value. In the example, the expression calls the `navAction()` method defined in the `MyBean` class:

```

public class MyBean {
    public MyBean() {}
    public String navAction() {
        if (<check some condition>) {
            return "success";
        }
        else {
            return "failure";
        }
    }
}

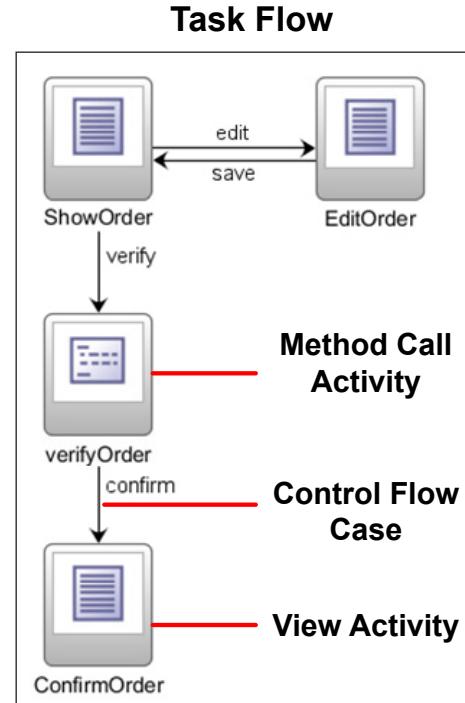
```

The `navAction()` method checks for some condition. If the condition is true, the method returns “success.” If the condition is false, it returns “failure.”

The navigation rule defined in `faces-config.xml` for `FirstPage.jspx` has two navigation cases. If the outcome of clicking the Submit button (and calling the `navAction()` method) is “success,” then the first navigation case executes, and `NewPage.jspx` is displayed. If the outcome is “failure,” then the second navigation case executes, and `Error.jspx` is displayed.

ADF Controller

- Enhances the JSF navigation model
- Uses task flows to manage page flow:
 - Promotes page reuse through abstraction
 - Supports calling methods and other task flows
- Supports application tasks:
 - Input validation
 - State management
 - Security
 - Declarative transaction control
- Provides Back-button control



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ADF Controller provides an enhanced navigation and state management model on top of the JSF Controller. Instead of representing an application as a single JSF page flow, you can break up the page flow into a collection of reusable task flows. Each task flow contains activities that represent simple logical operations, such as displaying a page, executing application logic, or calling a method or another task flow. Transitions between task flow activities are defined by control flow cases.

In addition to providing navigation and the ability to call methods from within the task flow, ADF Controller 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.

Task flows can be reused and can also be nested within pages. Task flows that are nested within pages become regions that contain their own set of navigable pages, allowing users to view a number of different pages and functionality without leaving the main page. (You learn more about regions in a later lesson.)

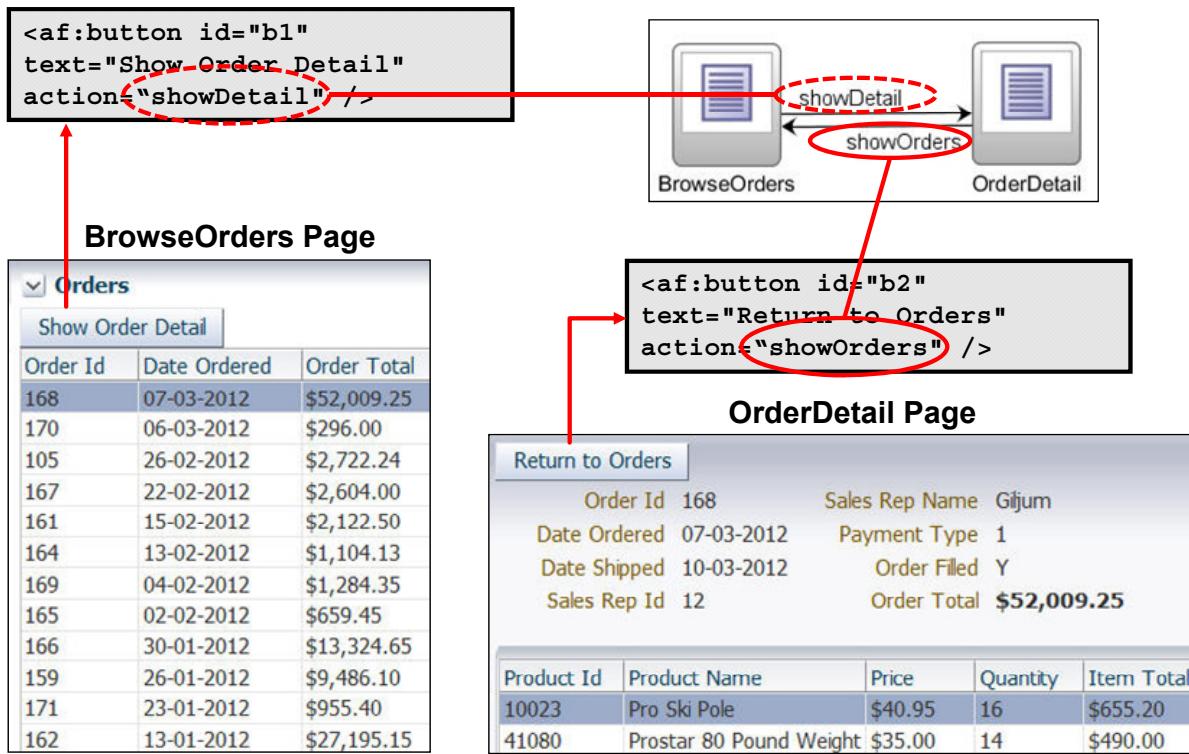
The example shows a simple task flow that has three view activities, ShowOrder, EditOrder, and ConfirmOrder, and one method call activity, verifyOrder. The arrows between the activities represent control flow cases. The labels next to each arrow represent the outcome that must be returned for that particular control flow case to execute. You define the task flow in a visual diagram, and it is saved as XML.

Task Flow Advantages

Task flows provide a number of advantages over using standard JSF page flows for navigation:

- Unlike JSF page flows, which represent the entire application in a single page navigation file, ADF task flows can be broken into a series of modular flows that call one another.
- ADF task flows are not limited to page navigation; they can call methods, other task flows, or activities that perform other tasks, such as conditional navigation.
- ADF task flows are reusable within the same or an entirely different application.
- Whereas JSF page flows provide no shared memory scope between multiple requests (except for session scope), ADF task flows provide a shared memory scope that enables data to be passed between activities within the task flow. (You learn more about memory scopes in a later lesson.)

ADF Controller: Task Flow Example



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Like any JSF application, an application that uses the ADF Controller contains a set of rules for choosing the next page to display when, for example, a button or link is clicked. You define the rules in an ADF Faces application by adding control flow rules and cases to task flows.

The example shows a simple task flow that defines navigation between two view activities: BrowseOrders and OrderDetail. In the XML source for the task flow definition, two control flow rules are defined.

The following control flow rule is defined for the BrowseOrders view:

```
<control-flow-rule>
  <from-activity-id>BrowseOrders</from-activity-id>
  <control-flow-case>
    <from-outcome>showDetail</from-outcome>
    <to-activity-id>OrderDetail</to-activity-id>
  </control-flow-case>
</control-flow-rule>
```

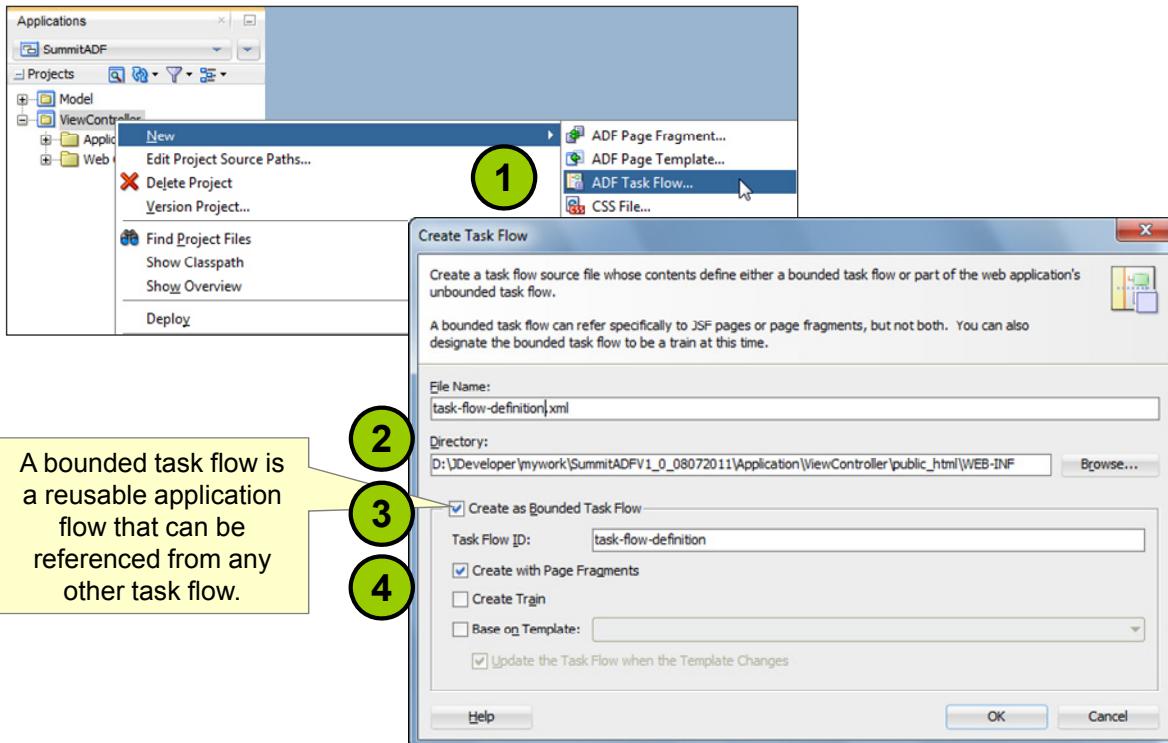
When a user clicks the Show Order Detail button, the outcome (set by the button's action attribute) is "showDetail." Therefore, the first control flow rule executes, and the OrderDetail view is displayed.

The following control flow rule is defined for the OrderDetail page:

```
<control-flow-rule>
  <from-activity-id>OrderDetail</from-activity-id>
  <control-flow-case>
    <from-outcome>showOrders</from-outcome>
    <to-activity-id>BrowseOrders</to-activity-id>
  </control-flow-case>
</control-flow-rule>
```

When a user clicks the “Return to Orders” button, the outcome (set by the button’s action attribute) is “showOrders.” Therefore, the second control flow rule executes, and the BrowseOrders view is displayed.

Creating Task Flows



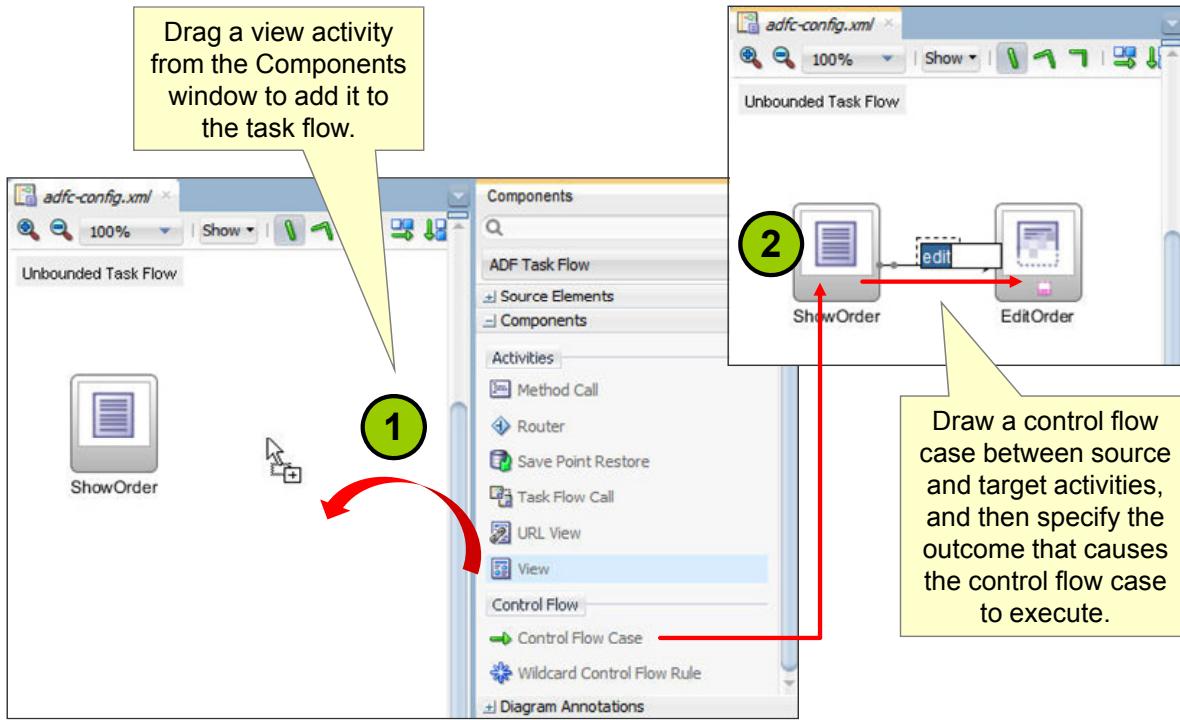
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To create a task flow:

1. In the Applications window under Projects, right-click a ViewController project, and select New > ADF Task Flow.
2. Specify a name and location for the task flow definition file.
3. Choose whether you want to create the task flow as a bounded task flow. An ADF application typically consists of one unbounded task flow and many bounded task flows.
 - The unbounded task flow is the entry point for the application. It contains any pages that can launch the application. JDeveloper automatically creates a default unbounded task flow, `adfc-config.xml`, when you create a new ADF fusion web application.
 - Each bounded task flow represents a reusable application flow that can be referenced from any other task flow. A bounded task flow has a single entry point (an activity that can be directly requested by a browser) and zero or more exit points. A bounded task flow supports reuse, parameters, transaction management, reentry, and can render within an ADF region in a JSF page.
4. For now, you can accept the defaults for other options. You learn more about these options and about bounded task flows in a later lesson.

Defining Activities and Control Flow Rules



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To define navigation, you add activities and control flow rules to the task flow diagram in the editor. As you learned earlier, a task flow is composed mainly of:

- Activities, which represent simple logical operations (such as displaying a page, executing application logic, or calling another task flow)
- Control flow rules, which define navigation between those activities

Typically, most of the activities in a task flow are view activities, which represent pages in the flow. In this lesson, you learn how to create task flows and define control flow rules between view activities. You learn more about other types of activities in a later lesson. To define a task flow:

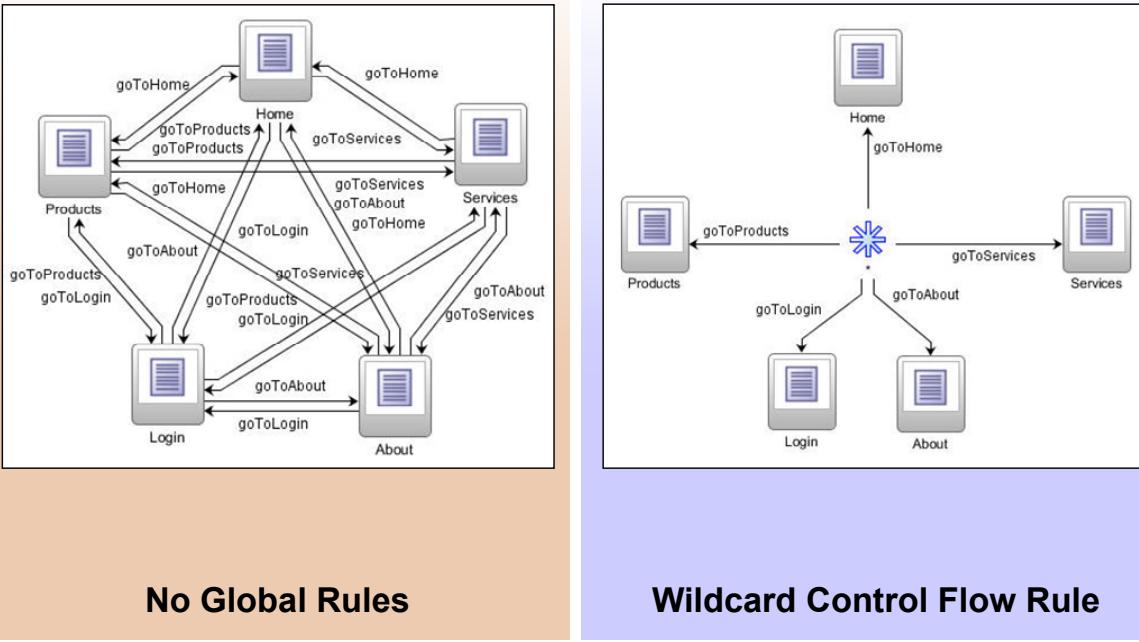
1. Add view activities to the task flow diagram:

- If you already have a page defined that you want to add to the task flow, you can select the page in the Applications window and drag it to the diagram. JDeveloper adds a view activity for the page to the task flow.
- Otherwise, select a view activity in the Components window and drag it to the diagram. Enter a name for the view activity. If you want JDeveloper to generate a JSF page for the view activity, double-click the view activity and specify the file name, the document type (facelets or JSP XML), the location, and whether to create a blank page or a page that is based on a template or quick-start layout.

2. Create control flow cases between all activities in the task flow. The easiest way to create control flow cases is by using the navigation modeler:
 - a. In the Components window, select the Control Flow Case component.
 - b. Click the source activity in the task flow, and then click the target activity to draw an arrow between the two activities.
 - c. Specify the from-outcome value for the control flow case. The control flow case will execute when a user performs an action on a navigation component (such as a button) and the outcome of that action matches the from-outcome that you specified for the control flow case. If the value is a string, you can specify it by typing the value next to the arrow in the diagram (or by typing the value in the From Outcome field in the Properties window). If the value is not a string (for example, because the outcome is determined by a method), you must specify the outcome in the From Action field in the properties window.

After you create the task flow, you can update it using the diagram, overview, and source editors. You can also use the Structure window to update the task flow.

Defining Global Navigation with Wildcards



No Global Rules

Wildcard Control Flow Rule

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Most control flow rules specify navigation from one activity to another. However, you can also define global rules that affect all pages by using a wildcard control flow rule. In a wildcard control flow rule, you specify a wildcard expression that allows the rule to be called from any activity that matches the expression. Wildcard expressions can include a single wildcard character (*) or a trailing wildcard character (foo*). You use a single wildcard character when you want to pass control from any activity in the task flow to the wildcard control flow rule. You use a trailing wildcard when you want to constrain the activities that can pass control to the wildcard control flow rule.

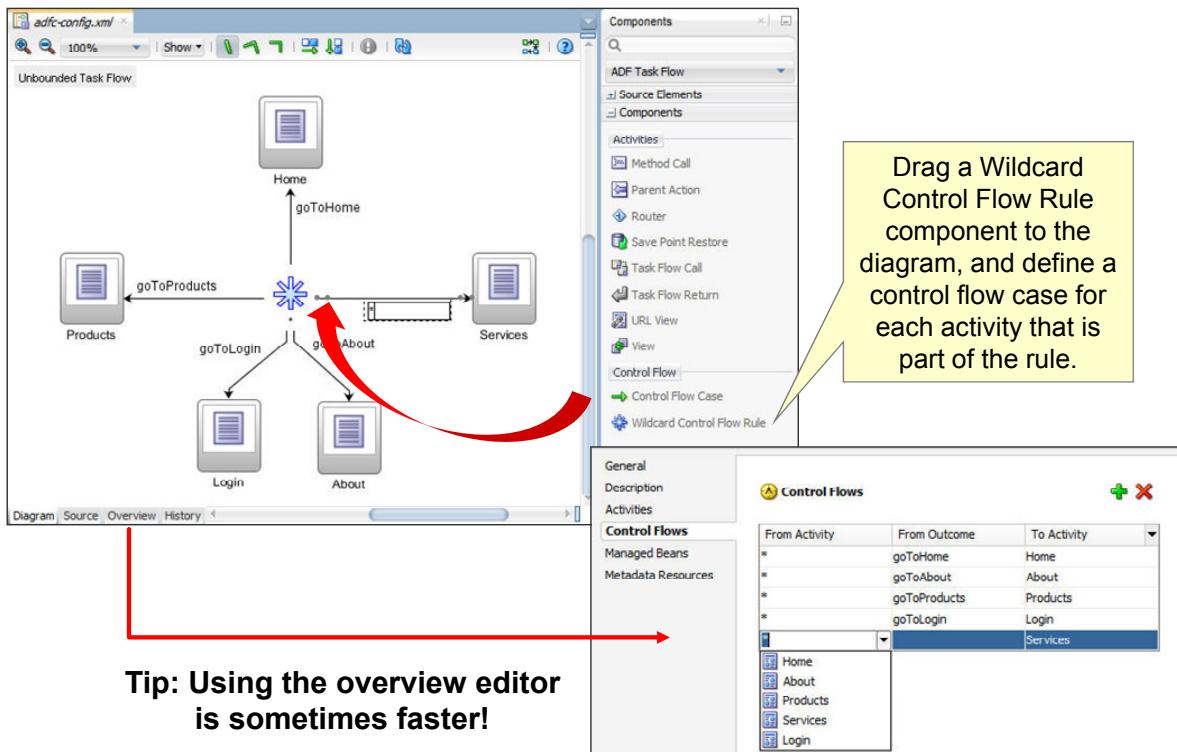
The example in the slide shows a task flow that uses a normal control flow rule compared to a task flow that uses a wildcard control flow rule. In the first example, every view activity in the flow contains a control flow case that points to every other activity in the task flow. The diagram is difficult to read and maintain. In the second example, a wildcard control flow rule allows every activity in the task flow to navigate to every other page in the task flow. The diagram in the second example is easier to read and maintain, but performs the same navigation as the first example.

The source for the wild card control flow rules looks like this:

```
<control-flow-rule>
  <from-activity-id>*</from-activity-id>
  <control-flow-case>
    <from-outcome>goToHome</from-outcome>
    <to-activity-id>Home</to-activity-id>
  </control-flow-case>
  <control-flow-case>
    <from-outcome>goToAbout</from-outcome>
    <to-activity-id>About</to-activity-id>
  </control-flow-case>
  <control-flow-case>
    <from-outcome>goToProducts</from-outcome>
    <to-activity-id>Products</to-activity-id>
  </control-flow-case>
  <control-flow-case>
    <from-outcome>goToServices</from-outcome>
    <to-activity-id>Services</to-activity-id>
  </control-flow-case>
  <control-flow-case>
    <from-outcome>goToLogin</from-outcome>
    <to-activity-id>Login</to-activity-id>
  </control-flow-case>
</control-flow-rule>
```

Notice that an asterisk is specified for the `<from-activity-id>` element in the control flow rule. This rule allows any page in the task flow to pass control to the rule if the outcome of the page (for example, the outcome of clicking a link or button) matches the value specified in the `<from-outcome>` of a control flow case in the rule.

Defining a Wildcard Control Flow Rule



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To define a wildcard control flow rule:

1. Drag a Wildcard Control Flow Rule component from the Components window to the task flow diagram. By default, the value of the `<from-activity-id>` of the rule is an asterisk (*), which means that any page in the task flow can pass control to this rule. You can change this value to a trailing wildcard character if you want to constrain the activities that can pass control to the wildcard rule.
2. Create a control flow case between the asterisk icon in the diagram and each activity that is part of the control flow rule.

Tip: If you already have control flow cases defined between activities and want to convert them to use a wildcard control flow rule, you can do this very quickly in the Overview editor. Click the Overview tab, and under Control Flows, change the From Activity to an asterisk (*) for every control flow case that you want to include in the rule. Then, delete any redundant control flow cases.

ADF Faces Navigation Components

Links:



Buttons:



Breadcrumbs:

Customers > Orders > Products

Trains:



Navigation Trees:



Menus:



And others!

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

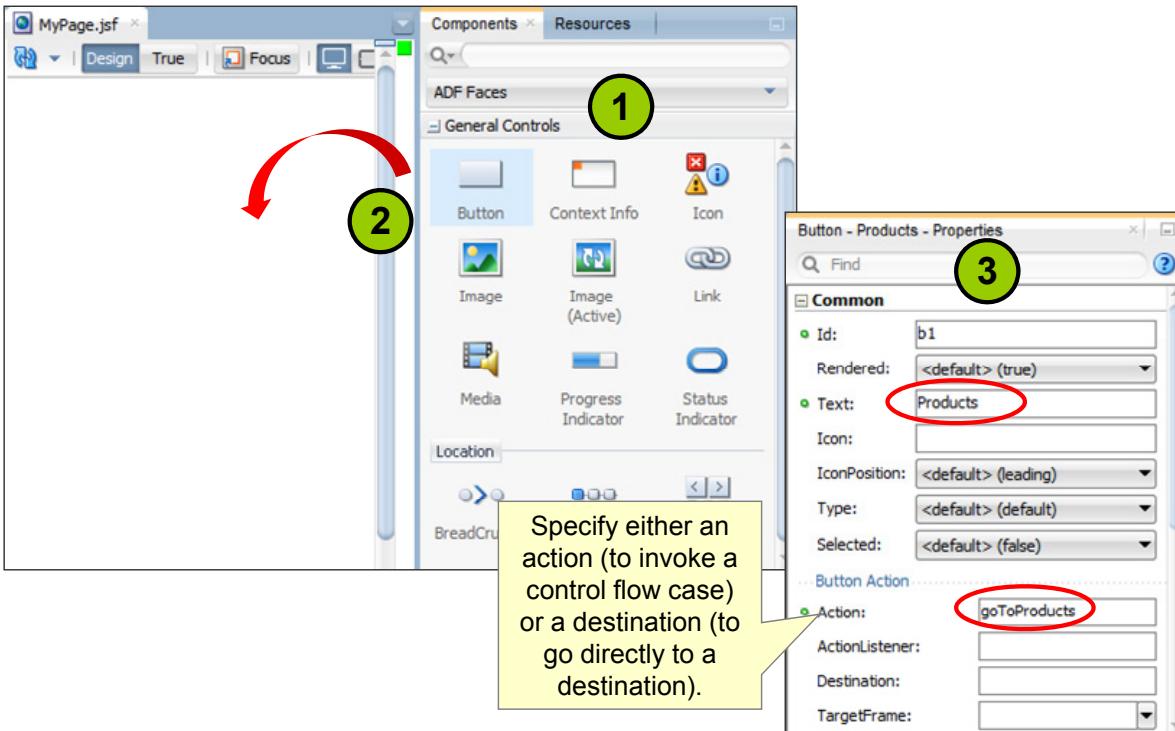
Navigation components allow users to drill down for more information, to navigate to related pages or windows, and to perform specific actions on data and navigate at the same time. The most common forms of navigation components are buttons and links. Other types of navigation components include components for navigating hierarchical pages (such as breadcrumbs), components for navigating through a multi-step process (such as trains), and components for defining menus. This lesson focuses on simple navigation components: buttons and links. You learn about the other types of navigation components in a later lesson.

When a user activates a navigation component (usually by clicking it), an event is generated. As you learned earlier, the ADF Controller uses the outcome string on the activated component to find a match in the set of control flow rules defined for the page. When the ADF Controller locates a match, the corresponding target page is selected and rendered.

ADF Faces provides two main components for implementing buttons and links:

- **Button component:** The Button component (`af:button`) consolidates the capabilities of several different button components (`af:goButton`, `af:commandButton`, `af:commandToolbarButton`, and `af:activeCommandToolbarButton`), which are now deprecated. The Button component behaves differently depending on whether the button has an action or destination specified. A button that has an action specified, when clicked, generates an action event on the server. (Remember that the ADF Controller uses the outcome of this action to determine which page to render next.) A button that has a destination specified navigates directly to another location (defined by a URI) instead of delivering an action. If both an action and destination are specified, the destination takes precedence. A button can be contained in a toolbar or placed elsewhere on the page. (You learn more about toolbar components in a later lesson.)
- **Link component:** The Link component (`af:link`) consolidates the capabilities of several different link components (`af:goLink`, `af:commandLink`, `af:goImageLink`, and `af:commandImageLink`), which are now deprecated. Just like the Button component, the Link component behaves differently depending on whether the link has an action or destination specified. A link that has an action specified, when clicked, generates an action event on the server. A link that has a destination specified navigates directly to another location (defined by a URI) instead of delivering an action. If both an action and destination are specified, the destination takes precedence. A link can be contained in a toolbar or placed elsewhere on the page.

Adding Navigation Components to a Page



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You add navigation components to a JSF page in the same way that you do other types of ADF Faces components:

1. In the Components window, expand General Controls.
2. Select a Link or Button component, and drag it to the Design view (or you can select a node in the Structure window, and click a Button or Link component to add it at the selected location).
3. Specify values for the following properties:
 - **Text:** Specify the text to display on the button or link. You can click the arrow next to the field to select a text resource from a resource bundle. (You learn more about resource bundles in a later lesson.)
 - **Action or Destination:** Specify either an action (to invoke a control flow case) or a destination (to go directly to a destination). To specify an action, enter either a static string that is the logical outcome of the action, or a method call that returns a string. The ADF Controller uses this string value to determine which page to render next. To specify a destination, enter a URI for the destination. (If you specify both an action and destination, the destination takes precedence.)
 - Specify other properties, as required. For example, for the Icon property, you can specify an icon to display in the button or link.

Summary

In this lesson, you should have learned how to:

- Explain how ADF extends the capabilities of the JSF controller
- Create task flows that include views, control flows, and global navigation
- Implement command buttons and links



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 5 Overview: Defining Task Flows and Adding Navigation

This practice covers the following topics:

- Building an unbounded task flow
- Adding view activities to the flow
- Defining control flow rules
- Adding global control flow rules
- Creating buttons and links for navigation



Copyright © 2014, 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 NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

Declaratively Customizing ADF Business Components

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

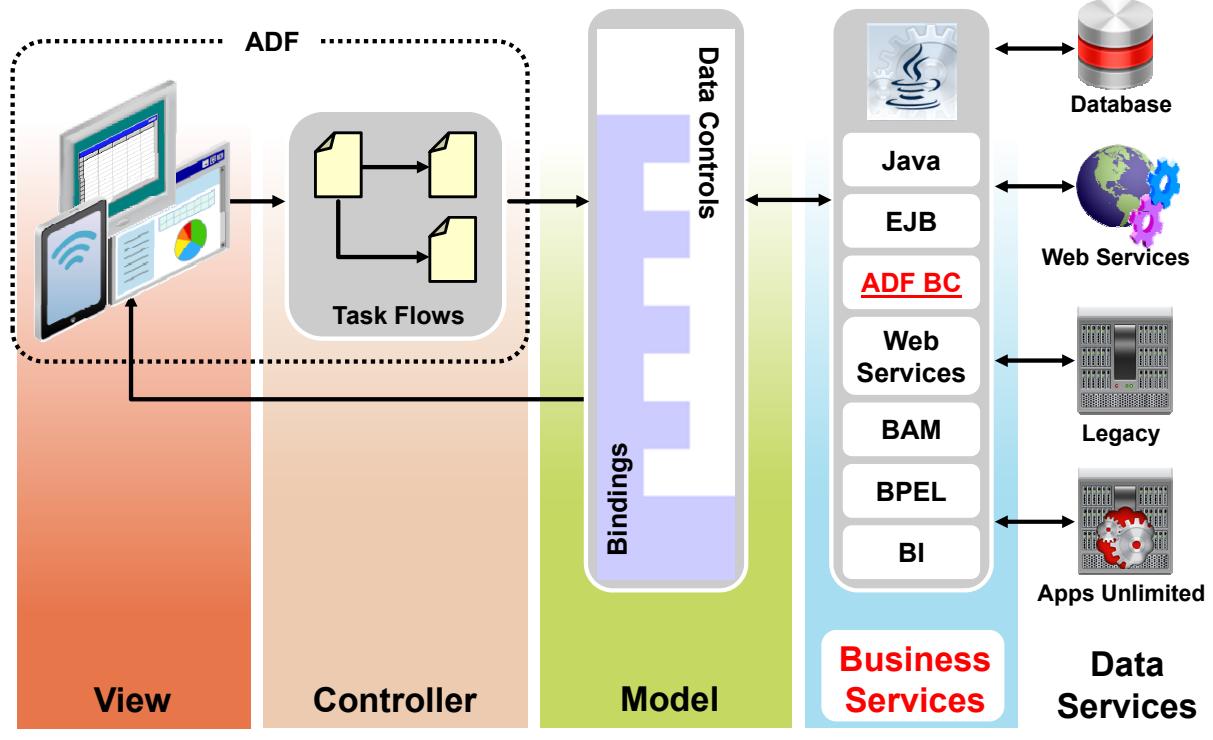
After completing this lesson, you should be able to:

- Declaratively modify entity objects, view objects, and application modules
- Create LOVs
- Create nested application modules



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Customizing Business Components



ORACLE

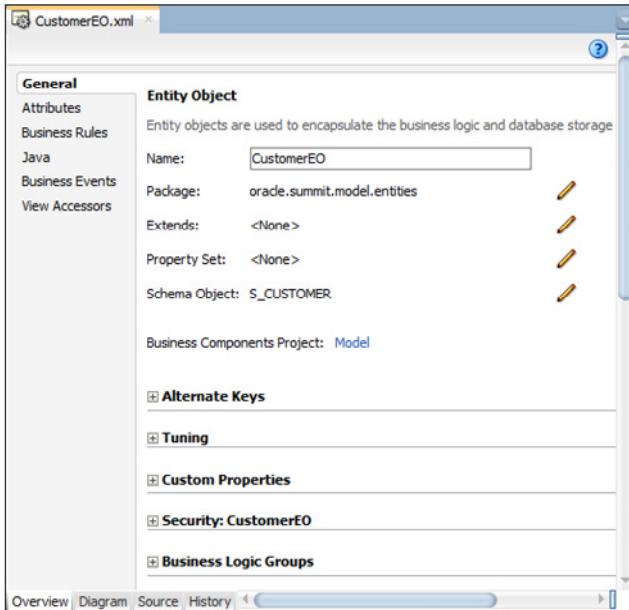
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In an earlier lesson, you learned about ADF Business Components, including entity objects, associations, view objects, view links, and application modules. You learned how to use the wizards in JDeveloper to create a first cut of your business components.

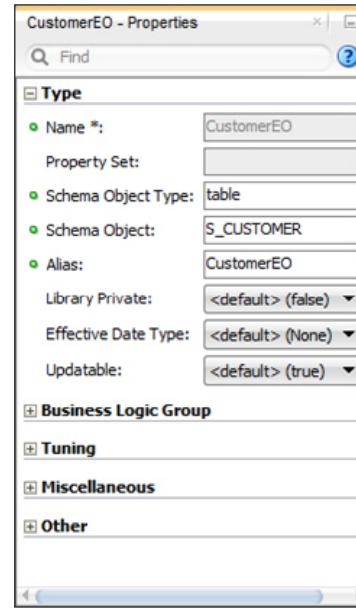
In this lesson, you learn how to customize the default behavior of business components.

Editing Business Components

Editors provide access to business component properties:



Entity Object Editor



Properties Window

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After you create a first cut of the business components for your application, you typically need to modify the default behavior of the components. For example, you might need to delete attributes from an entity object or change the data type of an attribute. JDeveloper provides two main windows for editing component properties declaratively: the main editor window and the Properties window. You can access all the properties of entity objects, associations, view objects, view links, and application modules through these windows.

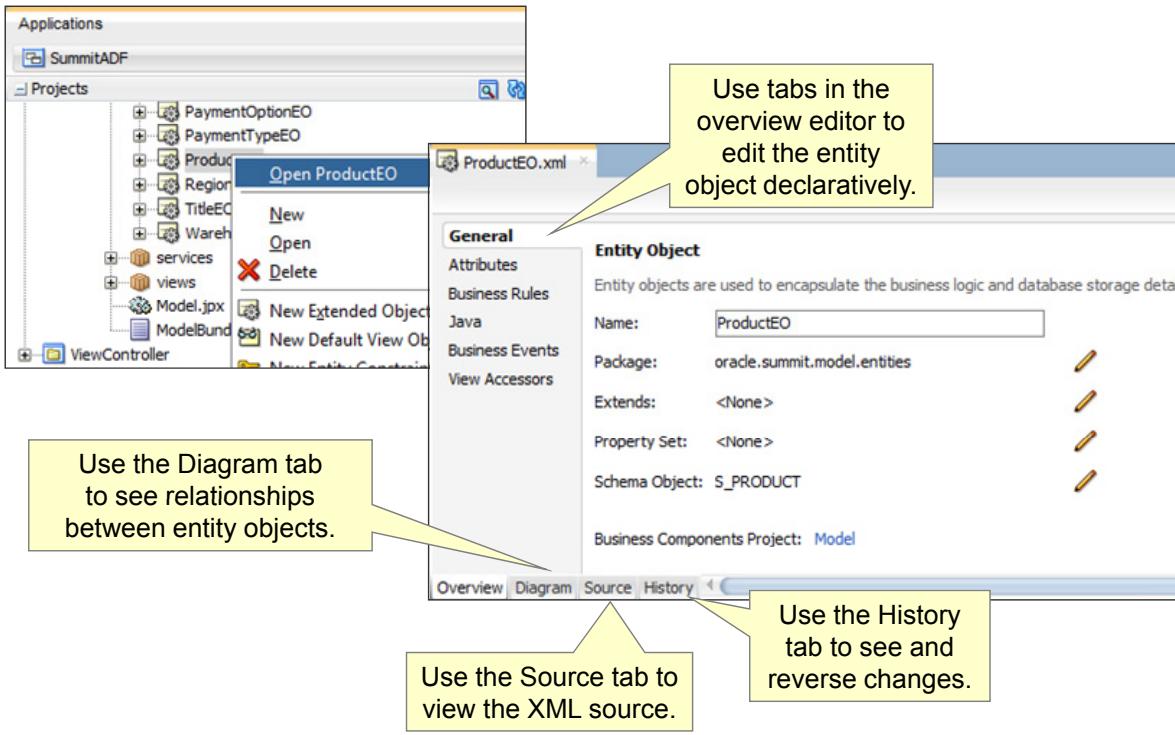
To open a business object in the main editor, double-click the object in the Applications window, or right-click the object and select Open.

In the main editor, you can edit a business component declaratively on the Overview tab, or you can click the Source tab at the bottom of the editor to edit the XML directly (editing the source directly is not generally recommended for ADF Business Components). The History tab enables you to see changes that have been made and to reverse those changes.

In addition to using the main editor, you can use the Properties window to edit business components. If the Properties window is not visible, select Window > Properties to display it.

The Properties window is divided into sections that show properties for the selected object. The Properties window, main editor, and source are all kept in sync. In the Properties window, you can search for a specific property by name.

Editing Entity Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To edit an entity object, double-click it in the Applications window, or right-click it and select Edit <EO Name> from the context menu. The XML file for the entity object opens in the main editor.

You can edit the entity object declaratively on the Overview tab, or, in rare instances, you might need to edit the entity object directly by clicking the Source tab at the bottom of the editor. Use the History tab to see changes that have been made and, if necessary, reverse those changes.

The Overview tab of the editor contains the following tabs:

- **General:** Set alternate keys, tuning, custom properties, security, and business logic groups for the entity object.
- **Attributes:** Modify or add attributes, add validation rules, set security, and define custom properties for the selected attribute. You can change the columns that are displayed in the table on this tab by clicking the down arrow at the upper-right corner of the table and selecting (or removing) columns from the table.
- **Business Rules:** Create and maintain business rules based on declarative settings, Groovy expressions, regular expressions, SQL queries, and Java methods for the entity object and its attributes.

- **Java:** Generate Java files or expose custom methods to client applications.
- **Business Events:** Define business events to publish to SOA.
- **View Accessors:** Define the list of available view accessors on the current entity object. You create a view accessor to point from a base entity object attribute or view object attribute to a source view row set. The view accessor returns a list of all possible values to the attribute of the base object. Entity-based view objects inherit all view accessors that are defined in the referenced entity object.

Modifying the Default Behavior of Entity Objects

With declarative settings, you can:

- Define attribute UI hints
- Create transient attributes
- Specify default values
- Synchronize with trigger-assigned values
- Define alternate key entity constraints

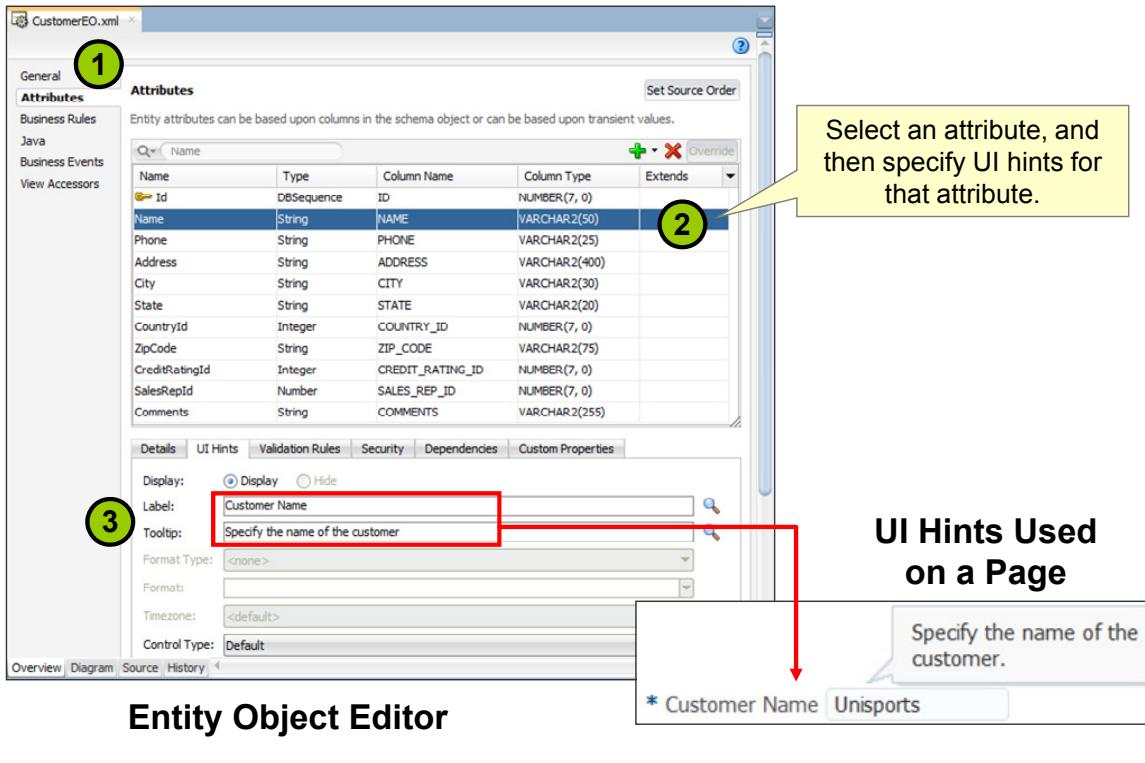


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

With JDeveloper, you have the ability to define many behaviors declaratively, which reduces the learning curve and the amount of time it takes to develop applications.

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

Defining Attribute UI Hints



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

One of the many built-in features of ADF Business Components is the ability to define UI hints on attributes. UI hints are additional attribute settings that the view layer can use to automatically display data model attributes and values to the user in a consistent, locale-sensitive way. For example, you can use UI hints to specify settings like whether or not an attribute can be displayed in the client, the label to use for a control, and the tooltip text to display for a control. View objects inherit UI hints that are set on the entity object, and can override the settings, if required.

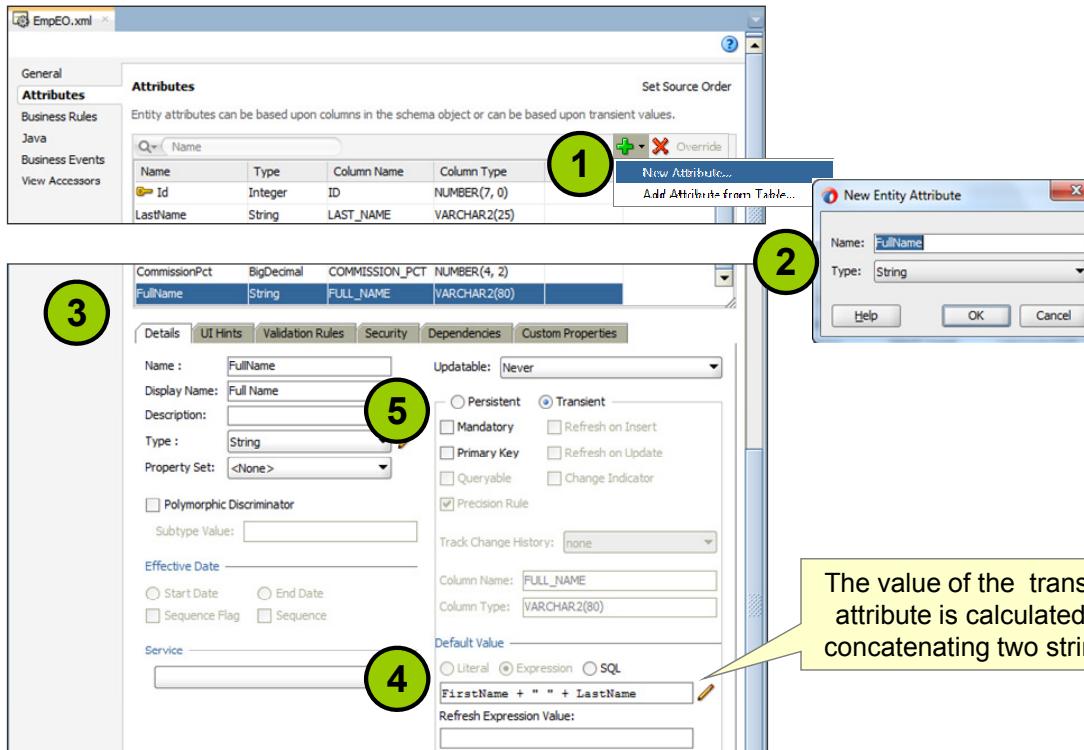
To add UI hints for an entity object or a view object attribute:

1. Open the object editor, and click the Attributes tab in the tree at the left.
2. Select an attribute.
3. In the attribute editor, click the UI Hints tab. Specify values for the UI hints that you want to set. You can type text values directly into the editor and JDeveloper will add the text resource to the default resource bundle for the page, or you can click the Select Text Resource icon to add the text to a specific resource bundle. (You learn more about resource bundles later in the course.)
4. Save your changes.

Some of the UI hints that you can specify are:

- **Display:** Determines whether the attribute should be displayed
- **Label:** Changes the default label for the attribute. Labels are prompts or table headers that precede the value of a field.
- **Tooltip:** 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)
- **Control Type:** Specifies the default control type that the client UI uses to display the attribute (The UI developer can override this for a particular page.)

Creating Transient Attributes



ORACLE

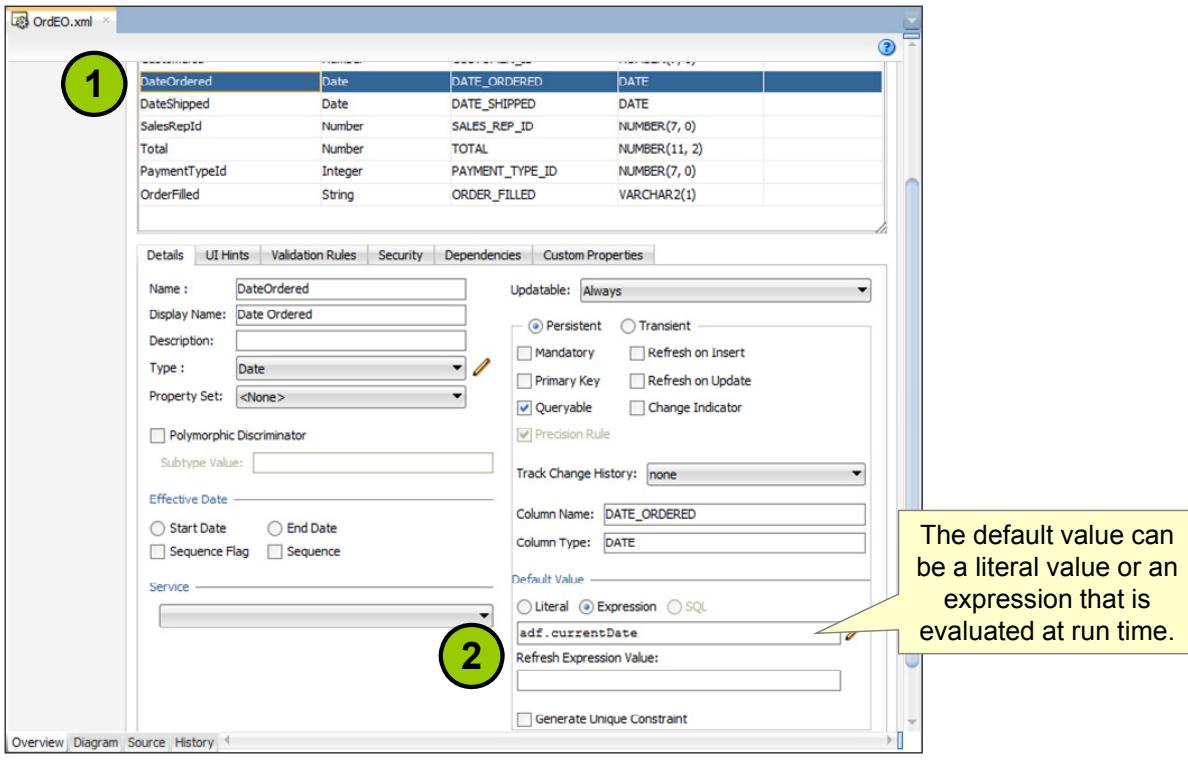
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Transient attributes are simply read-only attributes that are not persisted to the database. For example, you might use a transient attribute to provide a calculated value (such as a sub-total) that is not stored in the database.

To create a transient attribute, open the entity object or view object for which you want to define a transient attribute (in the Applications window, double-click the entity object or view object) and do the following:

1. On the Attributes tab of the object editor, click the Create New Attribute button and select New Attribute.
2. Enter a name for the attribute and select the data type. Then click OK.
3. On the Details tab for the new attribute, do one of the following:
 - Select Literal and specify a literal value for the transient attribute.
 - Select Expression and click the Edit Value (pencil) icon to specify the Groovy expression that will be used to calculate the attribute value. You can set additional options that control when to recalculate the expression or reset the attribute value when the value of a dependent attribute changes.
 - Select SQL and enter the SQL expression that will be used to calculate the attribute value. Adjust other settings, such as the type, as required.

Defaulting values



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You specify the default value of a persistent attribute in the same way that you specify the value of a transient attribute. The default value can be a literal value, or you can specify a Groovy expression or SQL statement that is evaluated at run time to return the default value.

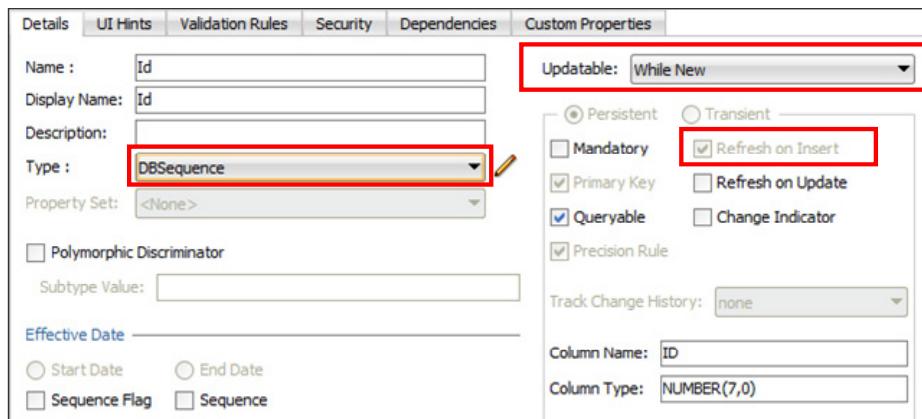
To specify the default value of an attribute:

1. On the Attributes tab of the Entity Editor, select the attribute for which you want to define a default value.
2. In the Default value field, do one of the following:
 - Select Literal and specify a literal value.
 - Select Expression and click the Edit Value (pencil) icon to specify the expression that will be used to calculate the attribute value. You can set additional options that control when to recalculate the expression or reset the attribute value when the value of a dependent attribute changes.
 - Select SQL and enter the SQL expression that will be used to calculate the attribute value.

When more than one attribute has a default value, the values are assigned in the order in which they appear in the entity object's XML file. Keep this in mind if you want the default value of an attribute to be based on another attribute with a default value.

Synchronizing with Trigger-Assigned Values

To use a database sequence to generate a primary key, set the Type to DBSequence.



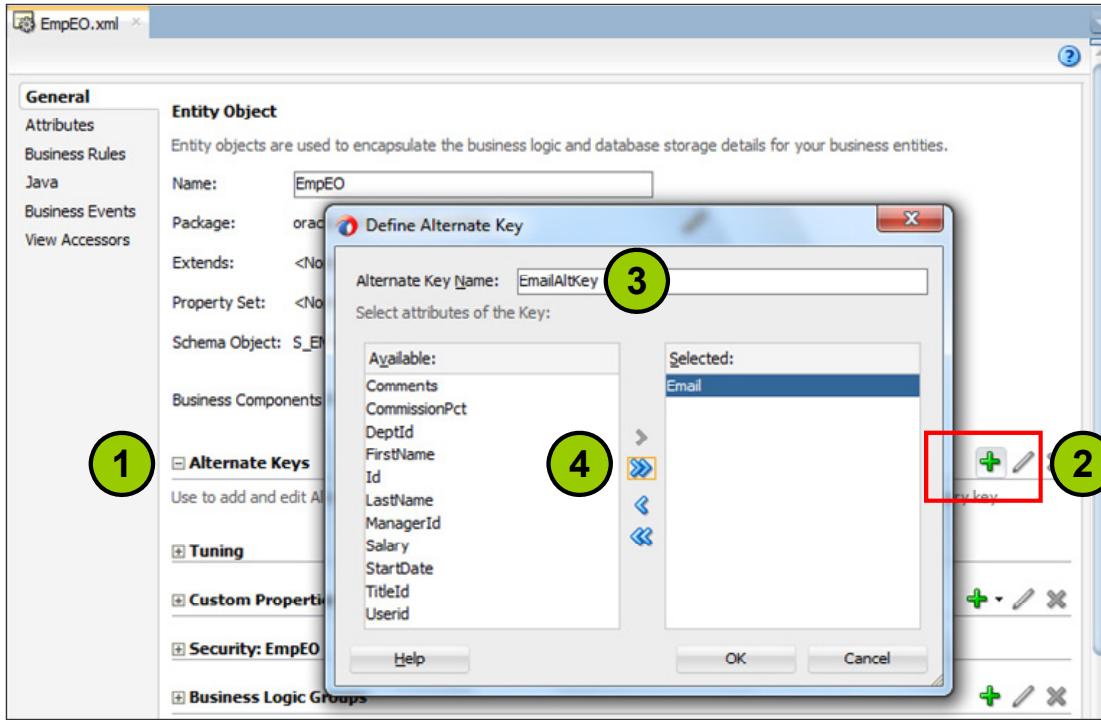
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If you are using a trigger to assign a primary key from a database sequence, you can set the Type of the primary key attribute to DBSequence. When you choose DBSequence, JDeveloper automatically sets the Updatable property to True and selects “Refresh on Insert.” These settings 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.

When you create a new entity row whose primary key is DBSequence, a unique negative number is 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 primary key value assigned by the database trigger. Any related new entities that previously used the temporary negative value as a foreign key will have that value updated to reflect the actual new primary key of the master.

Defining Alternate Key Values



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Alternate keys enable users to find entity rows at run time by using values other than primary or unique keys. For example, you might have a Customer entity object that includes an email attribute. If the customer ID is based on a database sequence, you might decide that you want to expose the customer's email address rather than the customer ID. To do this, you create an alternate key definition on the Customer entity and specify that the email attribute participates in the key definition.

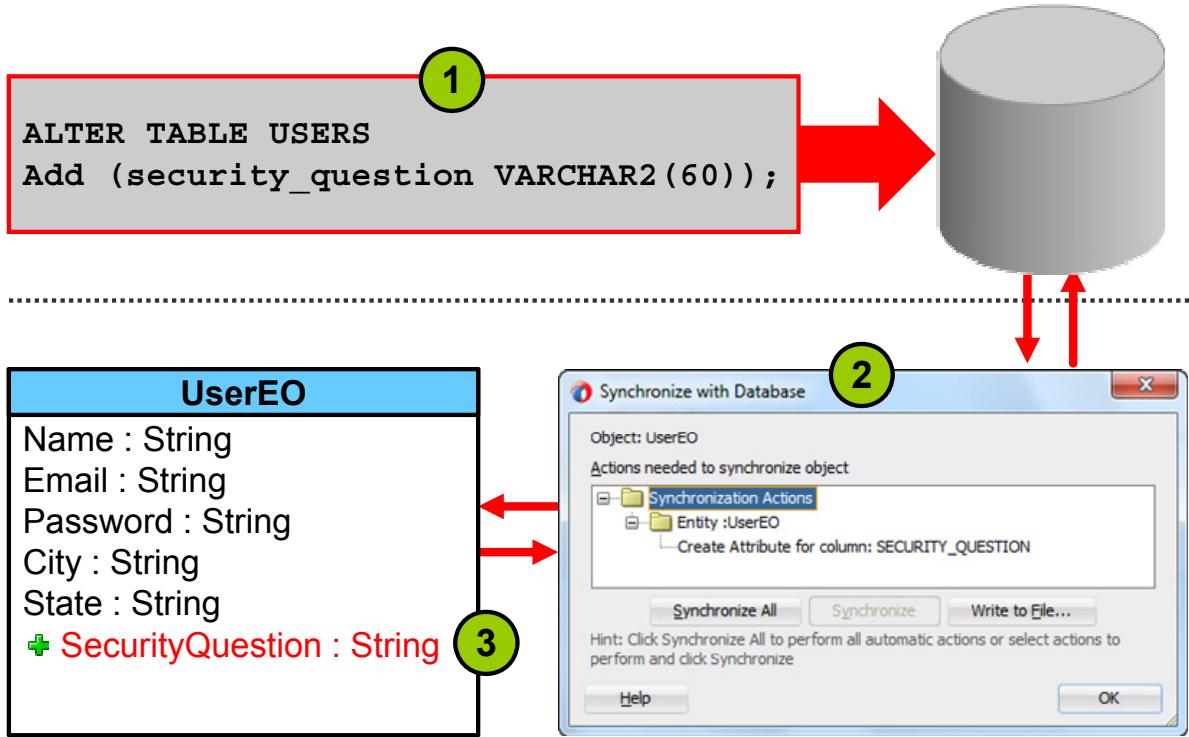
Alternate keys are useful because they do not create indexes in the database. Also, they are more efficient at searching the entity object cache.

To create an alternate key:

1. Open the entity object in the main editor and, on the General tab, expand Alternate Keys.
2. Click the Add Alternate Key (green plus sign: +) icon.
3. Specify the name of the alternate key.
4. Select the attributes that will participate in the key constraint.

You can also create an alternate key entity constraint from the Applications window. Right-click the entity object and select New Entity Constraint to run the Create Entity Constraint Wizard and define the alternate key.

Synchronizing an Entity Object with Changes to Its Database Table



ORACLE

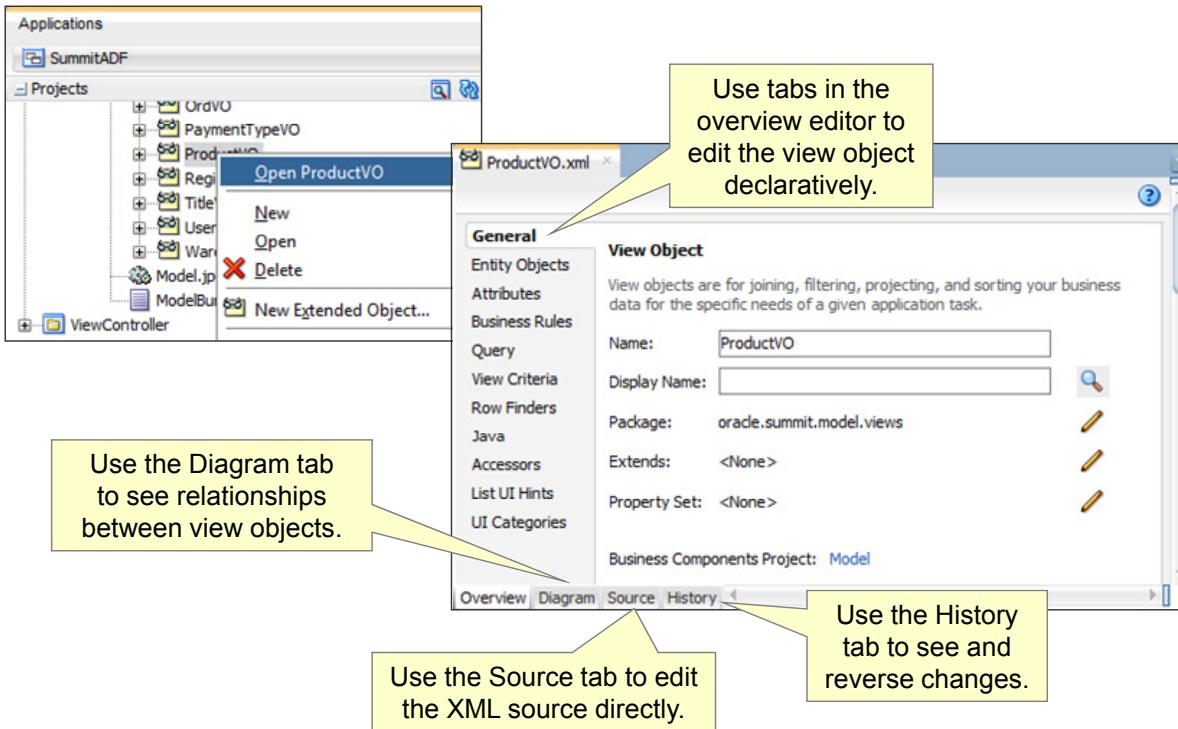
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If you alter a table for which you have already created an entity object, the existing entity is not affected by minor changes to the underlying table, such as the addition or removal of columns or changes to data types. However, if you want to access the changes in your application, you first need to synchronize the entity object with the database table. To perform this synchronization in JDeveloper, right-click the entity object and select “Synchronize with Database” from the context menu.

For example, suppose that you want to add a new column to the `USERS` table to store a user’s security question:

1. Issue the following SQL*Plus command to add a new `SECURITY_QUESTION` column to the `USERS` table:
`ALTER TABLE USERS ADD (security_question VARCHAR2 (60));`
2. Right-click the entity object and select “Synchronize with Database.” The “Synchronize with Database” dialog box lists the actions that it can perform for you automatically. You can click the `Synchronize All` button to perform all actions that are required to synchronize the object, or you can click the `Synchronize` button to perform only the selected actions.
3. After synchronization of the entity object, the new `SecurityQuestion` attribute is available for adding to the entity object.

Editing View Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The process for editing view objects is very similar to the process for editing entity objects. To edit a view object, double-click it in the Applications window, or right-click it and select Edit <VO Name> from the context menu. The XML file for the view object opens in the main editor.

You can edit the view object declaratively on the Overview tab, or, in rare instances, you might need to edit the view object directly by clicking the Source tab at the bottom of the editor. Use the History tab to see changes that have been made and, if necessary, reverse the changes.

The Overview tab of the editor contains the following tabs:

- **General:** Set custom properties, tuning, and 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. You can change the columns that are displayed in the table on this tab by clicking the down arrow at the upper-right corner of the table and selecting (or removing) columns from the table.
- **Business Rules:** Create and maintain business rules based on declarative settings, Groovy expressions, regular expressions, SQL queries, and Java methods for the view object and its attributes.

- **Query:** Modify the SQL query and set bind variables (to parameterize the WHERE clause). You learn more about bind variables later in this lesson.
- **View Criteria:** Refine the view object's query of the data source. From this tab, you can build a query filter condition that uses the attribute and view object names instead of the target view object's corresponding SQL column names.
- **Row Finders:** Find view rows at run time by using view criteria.
- **Java:** Generate Java files or expose custom methods to client applications.
- **Accessors:** Define view accessors that enable the view object to access values in other view objects; you use view accessors 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.
- **UI Categories:** Create identifiers to be used by the dynamic rendering user interface to group attributes for display.

Modifying the Default Behavior of View Objects

With declarative settings, you can:

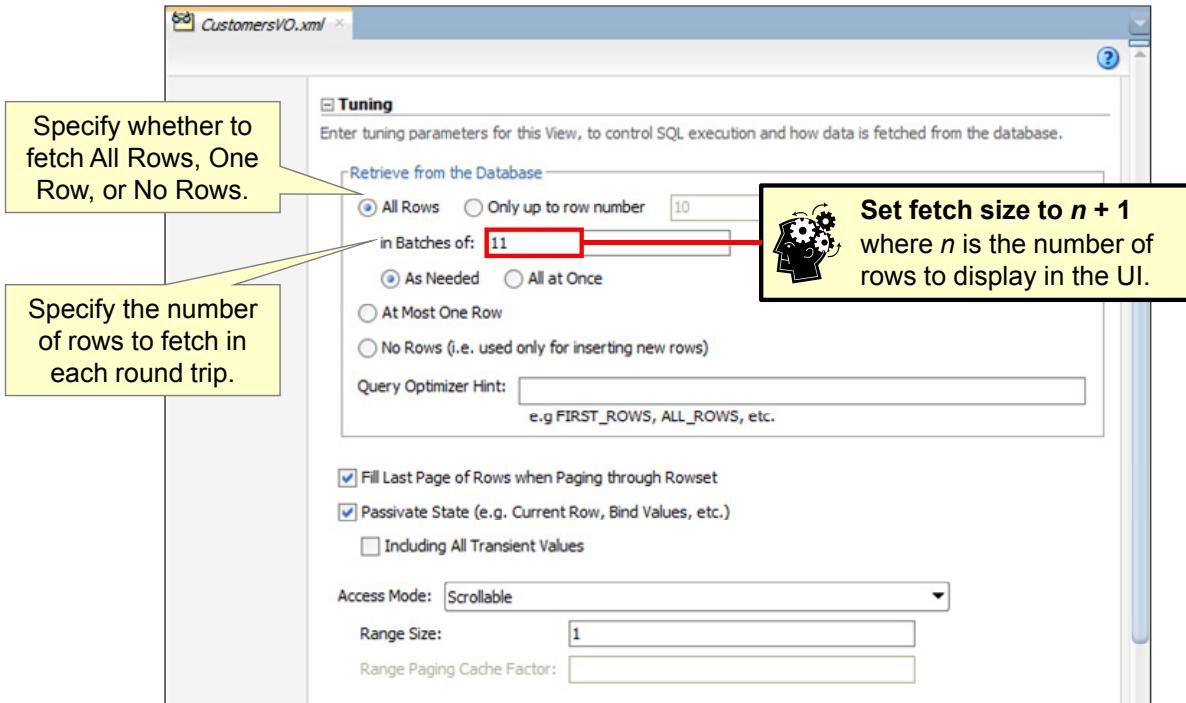
- Tune view objects
- Override attribute UI hints
- Perform calculations
- Restrict and reorder columns
- Restrict the rows retrieved by a query
- Change the order of queried rows
- Create join view objects
- Create master-detail relationships with view objects
- Define a list of values (LOV)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

Tuning View Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You set tuning options for a view object on the General tab of the view object editor. The tuning options can dramatically affect the query's performance.

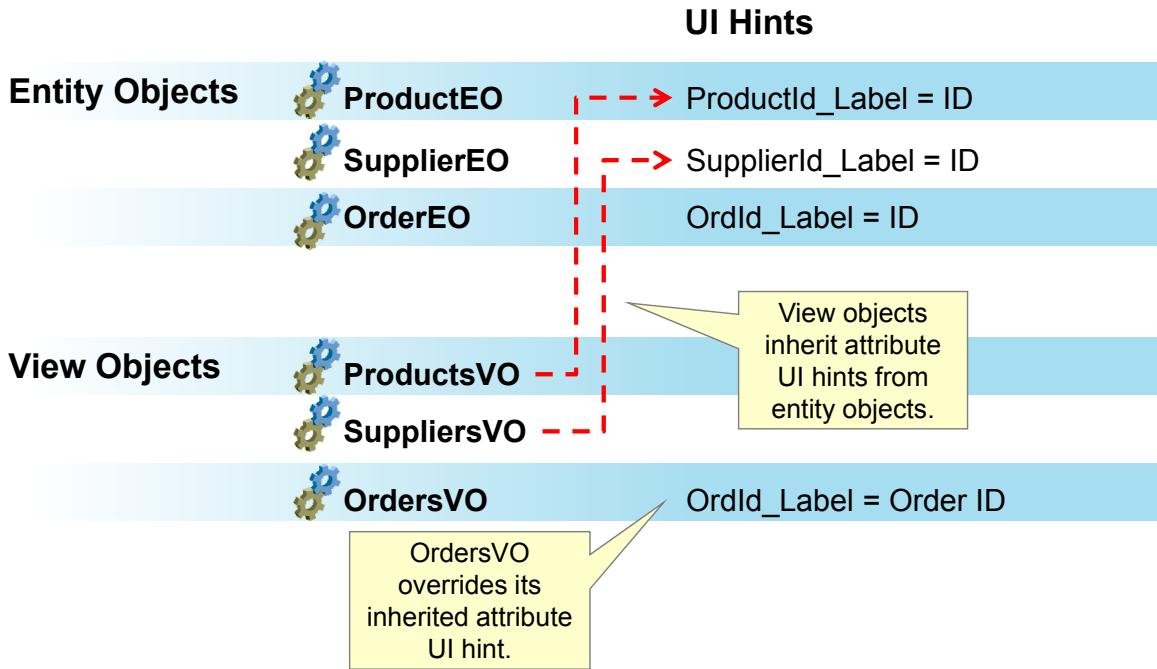
In the “Retrieve from the Database” area, you specify how Oracle ADF retrieves rows from the database. You can choose to fetch all rows, at most one row, or no rows. For most view objects, you use the All Rows option and specify whether the rows should be retrieved as needed or all at once.

For view objects in which the WHERE clause expects to retrieve a single row, select the At Most One Row option for best performance. This setting enables the framework to skip its normal test to determine if there are more rows.

By default, the framework fetches rows in batches of one row at a time. If you normally expect to fetch more than one row, you can control how many rows are returned in each round trip to and from the database by specifying a fetch size in the “in Batches of” field. As you increase the fetch size, keep in mind that you also increase the client-side buffer requirements. If you are displaying results n rows at a time in the user interface, it makes sense to set the fetch size to at least $n + 1$, so that each page of results can be retrieved in a single round trip to and from the database.

If you use the view object to create new rows only, select the No Rows option so that the query is never performed.

Overriding Attribute UI Hints



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

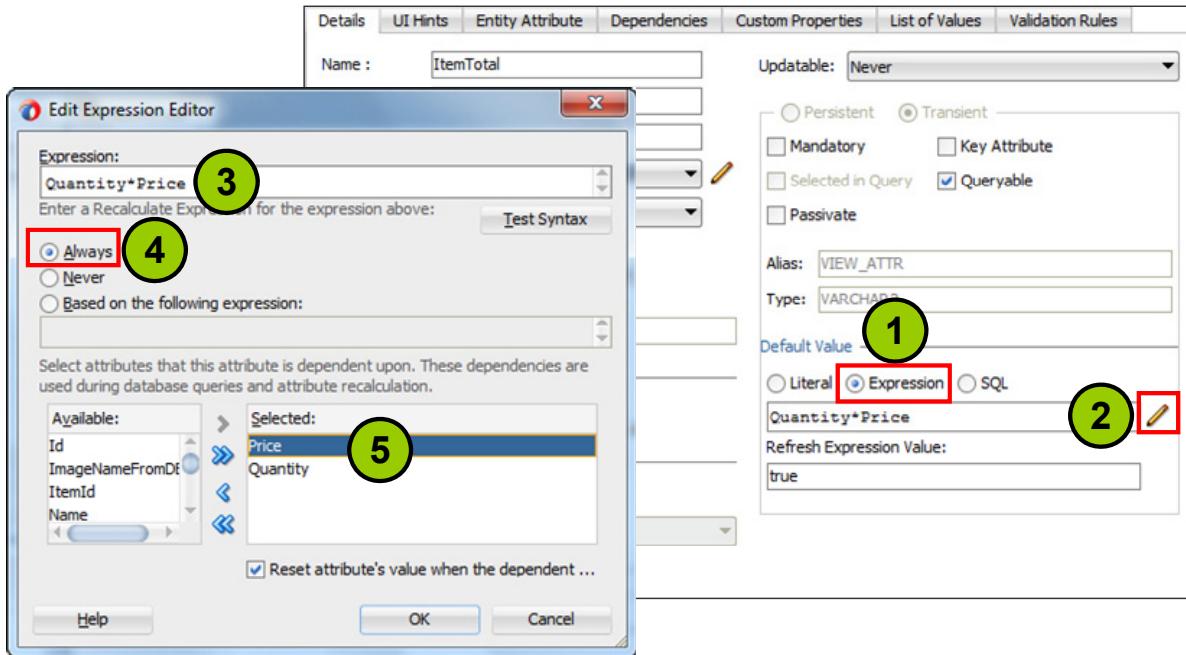
View objects inherit the UI hints that are defined by their associated entity objects. In most cases, you define UI hints at the entity object level to maintain consistency across all the views that use the entity object. However, you might have some use cases that require you to display data differently depending on the shape of the data. For these use cases, you can override the entity object UI hints by specifying UI hints at the view object level.

For example, suppose that you have several different entity objects with primary key attributes (such as ProductId, SupplierId, and OrdId). The UI hints for these attributes define the value of the label as simply ID. In most of your views, this label is sufficient because the context for the ID is clear. However, if you have a view that displays the OrdId along with another ID, like ProductId, you probably want to specify a unique label for OrdId. To do this, in the UI hints for the view object, specify a label that is more specific, such as Order ID.

Other use cases for overriding UI hints include when you want to use the same attribute twice, as in a recursive relationship, or when you want to specify UI hints for transient attributes that are defined at the view object level.

The process for specifying view object UI hints is similar to the process that you learned about earlier, but instead of editing the attributes in the entity object, you edit the attributes in the view object.

Performing Calculations



Set recalculation dependencies.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

For example, a view object based on ItemEO might have a transient attribute called ItemTotal. To implement a calculation for the ItemTotal attribute:

1. On the Details tab under Default Value, select Expression. Expressions use the Groovy expression language and can reference attributes from the view object definition.
2. Type the expression, or click the Edit Value (pencil) icon to open the edit expression editor.
3. In the Expression field, enter the calculation for the default value. The example uses a simple calculation, `Quantity*Price`. But in a real-world application, the expression should test for null values. You learn about the syntax of Groovy expressions in a later lesson.
4. Select the appropriate recalculate setting. If you select Always (default), the expression is evaluated each time any attribute in the row changes. If you select Never, the expression is evaluated only when the row is created.
5. Shuttle the attributes on which the calculated attribute is dependent into the Selected list. In the example, the attributes are `Quantity` and `Price`. This ensures that the value for `ItemTotal` is recalculated whenever the value for either `Quantity` or `Price` changes.

Restricting and Reordering Columns

The screenshot shows the Oracle ADF View Object Editor's Attributes tab. On the left, a sidebar lists various categories like General, Entity Objects, Business Rules, etc. The main area displays a table of attributes:

Name	Type	Alias Name	Entity Usage
ManagerId	Integer	MANAGER_ID	EmpEO
TitleId	BigDecimal	TITLE_ID	EmpEO
DeptId	Integer	DEPT_ID	EmpEO
Salary	BigDecimal	SALARY	EmpEO
CommissionPct	BigDecimal	COMMISSION_PCT	EmpEO

Annotations on the Attributes tab:

- A yellow callout points to the "Delete selected attribute(s)" button in the toolbar with the text: "Click here to delete selected attributes."
- A red callout points to the "Set Source Order" button in the toolbar with the text: "Click here to reorder attributes."

Annotations on the queries:

- A red callout points to the "Salary" and "CommissionPct" columns in the "Original Query" with a large red X, indicating they have been deleted.
- An arrow points from the "Original Query" to the "Modified Query".
- A yellow callout points to the "Modified Query" with the text: "The query changes to reflect the new SELECT clause."

Original Query:

```
SELECT EmpEO.ID,
       EmpEO.LAST_NAME,
       EmpEO.FIRST_NAME,
       EmpEO.USERID,
       EmpEO.START_DATE,
       EmpEO.COMMENTS,
       EmpEO.MANAGER_ID,
       EmpEO.TITLE_ID,
       EmpEO.DEPT_ID,
       EmpEO.SALARY,
       EmpEO.COMMISSION_PCT
  FROM S_EMP EmpEO
```

Modified Query:

```
SELECT EmpEO.ID,
       EmpEO.USERID,
       EmpEO.LAST_NAME,
       EmpEO.FIRST_NAME,
       EmpEO.START_DATE,
       EmpEO.COMMENTS,
       EmpEO.MANAGER_ID,
       EmpEO.TITLE_ID,
       EmpEO.DEPT_ID
  FROM S_EMP EmpEO
```

A modal dialog titled "Set Source Order" is shown, listing attributes with their corresponding entity usage:

- Id(EmpEO:ID)
- Userid(EmpEO:USERID)
- Lastname(EmpEO:LAST_NAME)
- Firstname(EmpEO:FIRST_NAME)
- Startdate(EmpEO:START_DATE)
- Comments(EmpEO:COMMENTS)
- ManagerId(EmpEO:MANAGER_ID)
- TitleId(EmpEO:TITLE_ID)
- DeptId(EmpEO:DEPT_ID)

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

On the Attributes tab of the view object editor, you can restrict the columns that are retrieved by the query by deleting attributes. You can also add attributes: either new ones or, for entity-based view objects, attributes that are based on columns 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 several columns containing employee data, including SALARY and COMMISSION_PCT. After you delete and reorder attributes, notice that the query changes. The SALARY and COMMISSION_PCT columns are no longer selected, and USERID appears before LAST_NAME.

Restricting the Rows Retrieved by a Query

The screenshot shows the Oracle ADF View Object Editor interface. On the left, there's a sidebar with various tabs: General, Entity Objects, Attributes, Business Rules, **Query**, View Criteria, Row Finders, Java, Accessors, List UI Hints, and UI Categories. The main area has tabs for 'Query' and 'Attribute Mappings', with 'Query' selected. Under 'Mode', it says 'Normal'. The 'Select' section contains the following SQL:

```
SELECT EmpEO.ID,  
       EmpEO.USERID,  
       EmpEO.LAST_NAME,  
       EmpEO.FIRST_NAME,  
       EmpEO.START_DATE,  
       EmpEO.COMMENTS,  
       EmpEO.MANAGER_ID,  
       EmpEO.TITLE_ID,  
       EmpEO.DEPT_ID  
FROM S_EMP EmpEO
```

The 'Where:' section contains the condition `EmpEO.DEPT_ID = 45`. A yellow callout box with the text "Specify a WHERE clause to restrict the rows retrieved by the query." points to this condition.

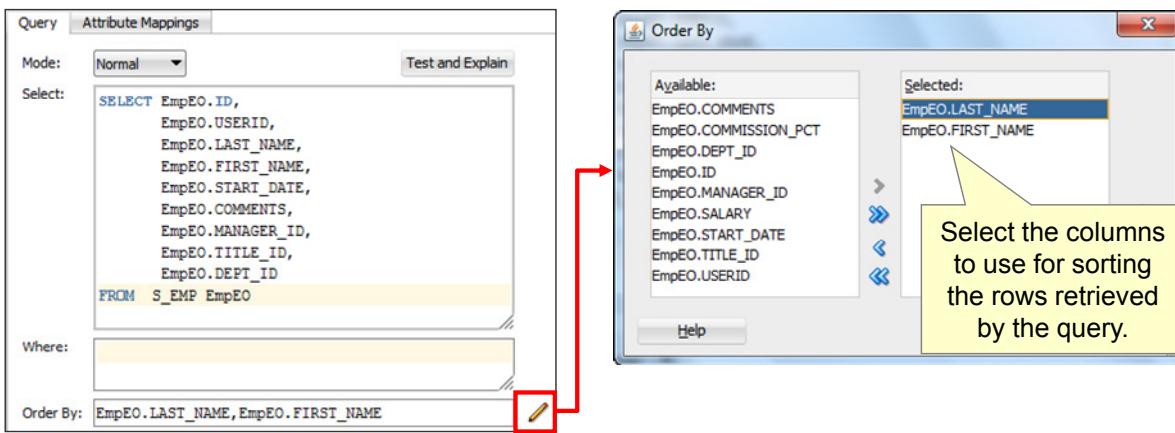
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

One way to restrict the rows retrieved by a query is to set a WHERE clause in the Query tab of the view object editor. You can hard code the value for the WHERE clause, or you can use a bind variable to parameterize the WHERE clause. You learn about bind variables later in this lesson.

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 the Edit (pencil) icon next to the Order By field.
3. In the Order By dialog box, select the columns for sorting the rows retrieved by the query.

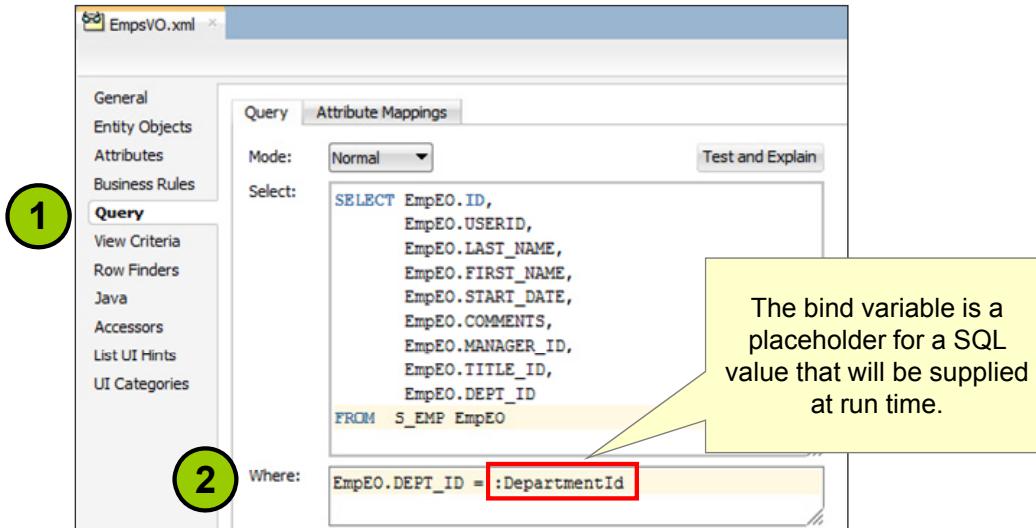


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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 the 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. You can specify the direction of the sort by adding `ASC` (for ascending) or `DESC` (for descending) to the end of the `ORDER BY` clause. By default, the sort order is ascending.

Using Parameterized WHERE Clauses



ORACLE

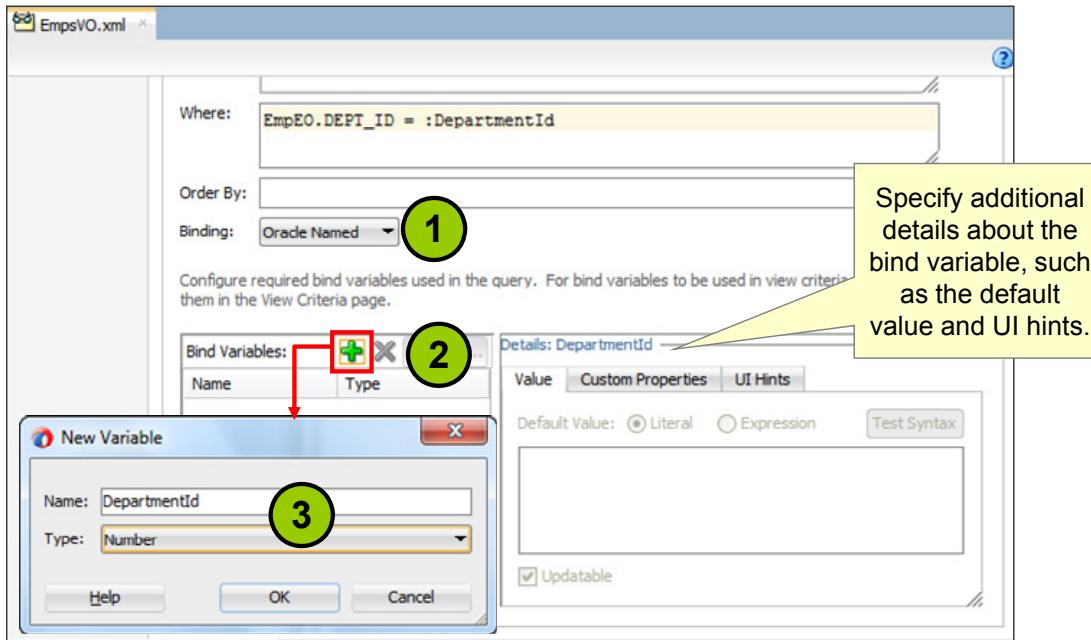
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You use a parameterized WHERE clause when you need to pass parameter values to the query at run time. To create a parameterized WHERE clause, specify a bind variable that serves as a placeholder for a SQL string that is supplied at run time. When you use bind variables, the query itself does not change across multiple executions, which means that the database can efficiently reuse the same parsed representation of the query, resulting in better runtime performance.

To use a parameterized WHERE clause in a view object definition:

1. Open the view object in the Overview editor, and click the Query tab.
2. Modify the WHERE clause to refer to a bind variable. Named bind variables begin with a colon followed by the name of the variable (for example, `:DepartmentId`). You learn how to create new bind variables in the next slide.

Creating Named Bind Variables



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Named bind variables enable developers to assign logical names to SQL query parameters, set default values, and reuse and reorder the parameters as needed. Bind variables also make the queries of view objects less susceptible to SQL injection attacks.

To create a new named bind variable:

1. At the bottom of the Query tab, next to Binding, verify that Oracle Named is selected. The other options in the list enable you to create unnamed, positional bind variables.
2. In the Bind Variables area, click the Create New Bind Variable icon.
3. Specify the name and data type of the bind variable. The bind variable must be of the same data type as the view object attribute that you intend to reference in the WHERE clause. 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. Add a new bind variable for each parameter that you want to set at run time.

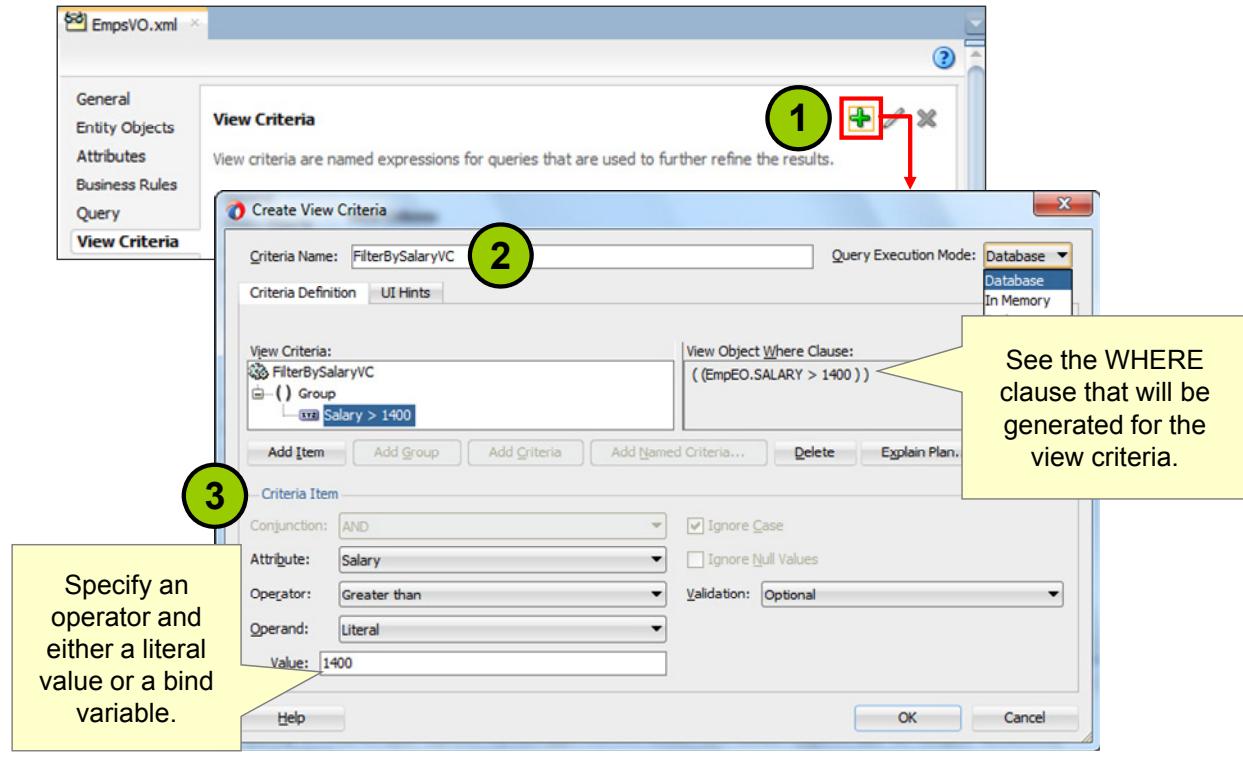
In the Details area, you can specify additional details about the bind variable, such as the default value (a literal value or expression) and UI hints. The view layer uses bind variable UI hints when you build user interfaces (such as search forms that enable the user to enter values for the named bind variables).

After defining bind variables, you reference them in the SELECT list or WHERE clause of a SQL statement. You saw an example of how to do this in the previous slide. You also learned that you reference the bind variable by using a colon followed by the variable name (an example is :DepartmentId). You can reference the bind variables in any order and repeat them as many times as needed in the SQL statement.

For example, you can create a bind variable called `ProdId` and reference it in the following WHERE clause: `WHERE Products.PRODUCT_id= :ProdId`.

You can test bind variables by assigning values to them in the Oracle ADF Model Tester, or you can click the “Test and Explain” button on the Query tab and then click the Query Results tab to enter values for bind variables and see the query results.

Creating Named View Criteria (Structured WHERE Clauses)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can augment a view object's WHERE clause by using a named view criteria. A view criteria is a filter that you define at design time and can apply to a view object instance at run time. The view criteria augments the view object's WHERE clause. However, unlike the WHERE clause defined by the view object query statement, which applies to all instances of the view object, the view criteria query condition can be added to specific view object instances. You can think of the view criteria as a WHERE clause "fragment" that is added to the selected view object instance. The view criteria enables you to define query conditions at design time that can be used to filter the results of specific view object instances at run time.

For example, suppose you have a view object definition called EmpsVO that includes the attributes Id, LastName, FirstName, Salary, and DeptId. In your application, you want to apply a filter to the view object to display a list of top earners (employees who make over 1400). To do this, you define a view criteria called FilterBySalaryVC that tests for the condition `Salary > 1400` and filters out any employee rows that do not meet this criteria. Then you apply the view criteria to an instance of the EmpsVO called TopEarners.

To create a named view criteria:

1. On the View Criteria tab of the view object editor, click the Create New View Criteria icon.
2. Enter a name to identify the view criteria. The name you enter should help identify the view criteria's purpose in the project.

3. Add one or more criteria items to the view criteria. A criteria item consists of an attribute name, an attribute-appropriate operator, and an operand. The criteria item shown in the slide uses a literal value (1400). However, if you want to pass in a value at run time, select Bind Variable for the operand and select (or create) the bind variable that you want to use. The process you follow for creating the bind variable is similar to the process you learned about earlier, except that you create the bind variable on the View Criteria tab (not shown here) rather than the Query tab.

You can (optionally) create groups of criteria items. By default, multiple items will be joined by a logical AND conjunction. You can change the item's conjunction to a logical OR in the Criteria Item group box. You can also add nested view criteria as well as named criteria that you have already defined.

Notice that you can select the query execution mode to control whether the rows for the view object are retrieved from the database, from memory, or from both. Select Both when you have newly created (but not yet committed) rows in the transaction and want to filter them. The newly entered rows, which do not yet exist in the database, will be combined with the filtered query results from the database table.

The View Object Where Clause field displays the WHERE clause that will be used.

Applying Named View Criteria to a View Object Instance

The screenshot shows the Oracle ADF Data Model Components interface. On the left, under the 'Data Model' tab, there is a tree view of 'Available View Objects' containing 'model.Model', 'model', 'CustomersVO', 'OrdsVO via SOrdCustomerIdFkLink', 'DeptsVO', 'EmpsVO via SEmpDeptIdFkLink', and 'EmpsVO'. A yellow circle labeled '1' is placed over the 'Data Model' tab.

In the center, under 'View Object Instances', there is a tree view of 'Data Model' containing 'AppModule', 'EmpsV01', and 'TopEarners'. A yellow circle labeled '2' is placed over the 'Edit...' button next to 'TopEarners'.

A callout bubble points to the 'Edit...' button with the text: "Select one or more view criteria to apply to the view object instance."

On the right, the 'View Criteria' panel is open. It shows a 'View Definition' of 'model.EmpsVO' and a 'View Criteria' section. The 'Available' list contains 'Selected:' and 'FilterBySalaryVC'. A yellow circle labeled '3' is placed over the 'Selected:' list.

Resulting Query:

```

SELECT EmpEO.ID,
       EmpEO.LAST_NAME,
       EmpEO.FIRST_NAME,
       EmpEO.SALARY,
       EmpEO.DEPT_ID
  FROM S_EMP_EmpEO
 WHERE ( (EmpEO.SALARY > 1400) )
  
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After defining a named view criteria, you can apply it to specific view object instances. By using this approach, you can define multiple view usages for a single base view object definition in an application module.

To apply named view criteria to a view object instance:

1. Double-click the application module to open it, and click the Data Model tab.
2. Under View Object Instances, in the Data Model area, click the Edit button.
3. Under View Criteria, shuttle one or more view criteria into the Selected list. If you select multiple view criteria, they are combined with an AND operator.

Now, when the view object query is issued, it includes a `WHERE` clause that is based on the selected view criteria.

This slide shows only one of the possible use cases for named view criteria: defining multiple view usages for a single base view object definition in an application module. Another common use case for view criteria is using a view criteria in combination with the `af:query` component to build query-by-example search forms. You learn more about search forms and other uses for view criteria later in the course.

Creating Join View Objects

The screenshot shows the 'EmployeesVO.xml' view object configuration in Oracle ADF. The 'Entity Objects' tab is selected. In the 'Available' list, 'DeptEO' is selected. In the 'Selected' list, 'EmpEO' and 'DeptEO' are listed under the 'Subtyp' heading. A yellow callout points to the 'Selected' list with the text: 'The view object uses attributes from two different entity objects.' Below the lists, configuration options include 'Entity Usage: DeptEO', 'Definition: model.DeptEO', 'Association: SEmpDeptIdFkAssoc.DeptEO', 'Source Usage: EmpEO', 'Join Type: left outer join', and checkboxes for 'Updatable', 'Reference', and 'Participate in row delete'. A yellow callout points to the 'Join Type' dropdown with the text: 'The join is based on an association between the two entities.'

The selected object, DeptEO, contains reference information.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A join view object is one that is based upon multiple entity objects that are related by one or more associations. You can define joins when you create a view object, or you can define joins by editing an existing view object.

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

The join type that you select determines how the data in the view object is shaped. Therefore, you must consider the needs of your application before determining the best join type to use.

Reference Entities

In business applications, you commonly need to supplement information from a primary business domain object with secondary reference information to help end users understand what the foreign key attributes represent. For example, when displaying employees, it might be more meaningful for users to see the department name rather than the department number. The department name is in a separate entity from the employees' entity. To include the department name with the employee information, you can create a join view object that includes attributes from both objects. Select the Reference check box to indicate that the entity (DeptEO in the example) is a reference entity. A view object can include multiple reference entities. In fact, typically all but one of the entities in a view object are reference entities. The example in the slide shows the EmpEO entity as the primary entity for the view object.

Selecting Attributes in Join View Objects

Attributes tab > Add Attribute from entity

The screenshot shows the Oracle ADF Model Tester interface. On the left, the 'Attributes' dialog box is open, displaying the 'Available' section with entities EmpEO and DeptEO. Under EmpEO, attributes like Id, LastName, FirstName, Salary, and DeptId are listed. Under DeptEO, attributes Id and Name are listed. The 'Selected' section contains attributes Id(EmpEO:ID), LastName(EmpEO:LAST_NAME), FirstName(EmpEO:FIRST_NAME), Salary(EmpEO:SALARY), DeptId(EmpEO:DEPT_ID), Id1(DeptEO:ID1), and Name(DeptEO:NAME). A callout bubble points to the 'Name(DeptEO:NAME)' attribute with the text 'Read-only reference attributes are disabled.' In the center, a red box highlights the 'Resulting Query' which shows the SQL code:

```
SELECT EmpEO.ID,
       EmpEO.LAST_NAME,
       EmpEO.FIRST_NAME,
       EmpEO.SALARY,
       DeptEO.ID AS ID1,
       DeptEO.NAME
  FROM S_EMP EmpEO, S_DEPT DeptEO
 WHERE EmpEO.DEPT_ID = DeptEO.ID (+)
```

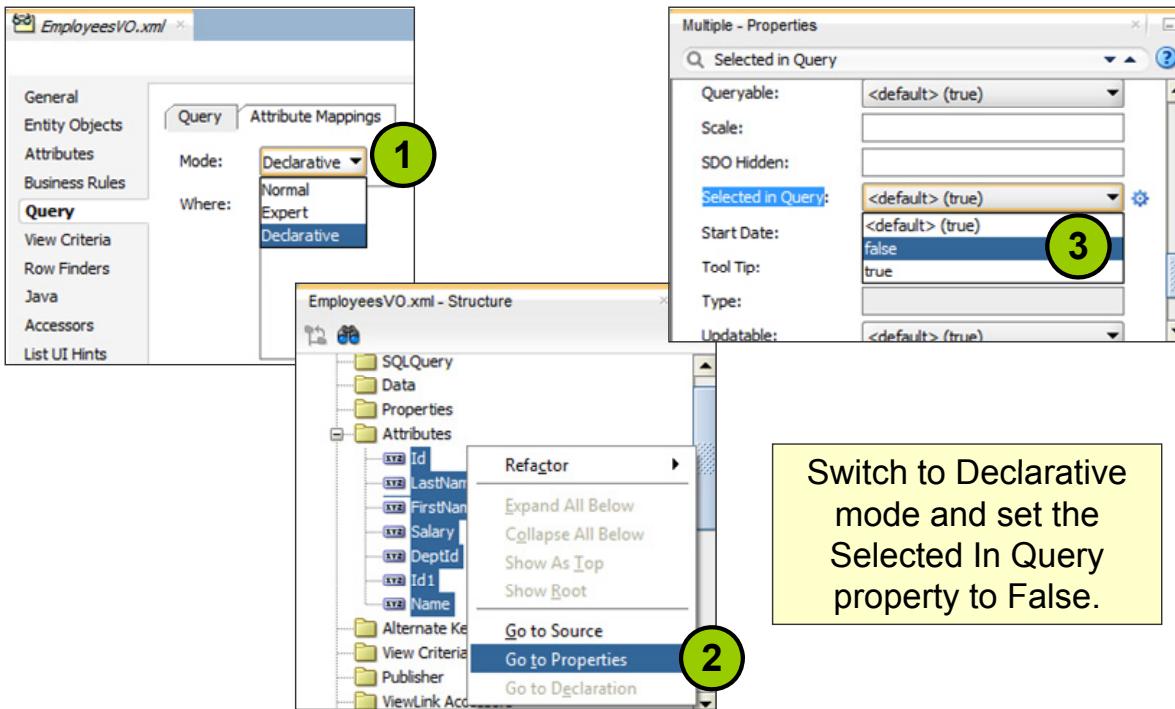
To the right, another red box highlights the 'Model Tester' window showing a table with data:

	Id	LastName	FirstName	Salary	DeptId	Id1	Name
1	Velasquez	Carmen	2500	10	10	Finance	

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After selecting entities to include in the view object and specifying details of the join, you need to select attributes to include in the view object. In the example, the Id, LastName, FirstName, Salary, and DeptId attributes are added from the primary entity object, EmpEO. The Id and Name attributes are added from the reference entity, DeptEO, to be displayed as read-only attributes. When you run this view object in the ADF Model Tester, the reference attributes are disabled. If you change the value DeptID in the view object, the framework automatically retrieves the department name into the current view.

Creating More Efficient Queries by Using Declarative View Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can make the SQL query for an entity-based view object more efficient by switching from Normal mode to Declarative mode. In Declarative mode, SQL generation is deferred until the runtime execution of the object. The SELECT and FROM lists are generated at run time based on the data-bound UI component's usage of one or more attributes. Attributes that are not exposed in the UI are not included in the query (they are pruned). As an option, you can apply named view criteria to filter the SELECT and FROM lists that are generated at run time.

Selecting Declarative mode often results in a simpler SQL statement and ensures that the query selects only the attributes that are needed. For example, you might have a use case that requires you to create a join view object that is based on multiple entity objects. However, the page that you build might expose attributes from only one of the entity objects. When Normal mode is selected, the query selects all attributes that meet the criteria defined by the join. However, when you select Declarative mode, the query selects only the attributes that are exposed in the UI by a data-bound component, thereby avoiding the overhead of an unnecessary join.

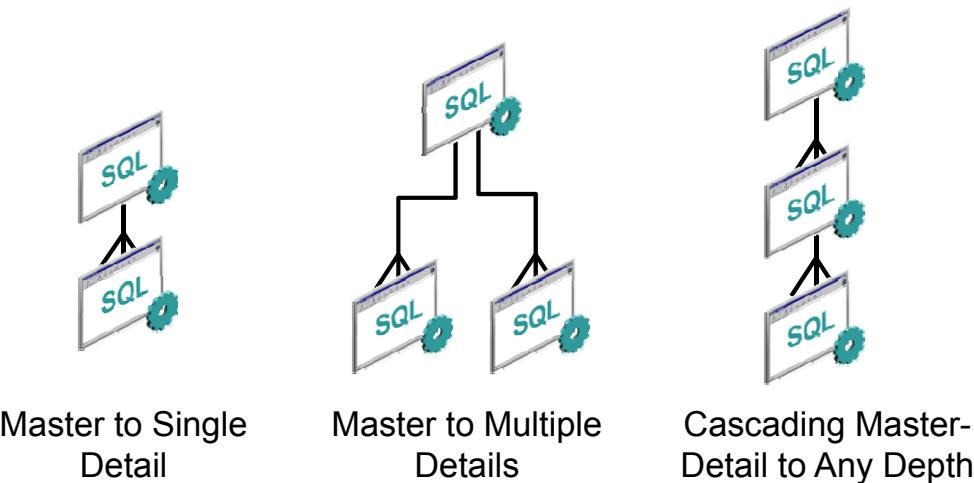
To create a declarative view object, perform the following steps after creating the view object:

1. On the Query tab of the view object editor, change the mode to Declarative. Notice that the tab no longer displays the query because the actual query is constructed at run time.
2. In the Structure window, under Attributes, select all the attributes in the view object. Then right-click and select “Go to Properties” to perform a global change on all the selected attributes.
3. In the Properties window, change the value of the “Selected in Query” property to False. With this property set to False, the attributes are selected only if they are needed.

You can go to Tools > Preferences > ADF Business Components > View Objects to enable Declarative SQL mode as the default for any new objects that you create.

Creating Master-Detail Relationships between View Objects

Different types of master-detail relationships are supported.



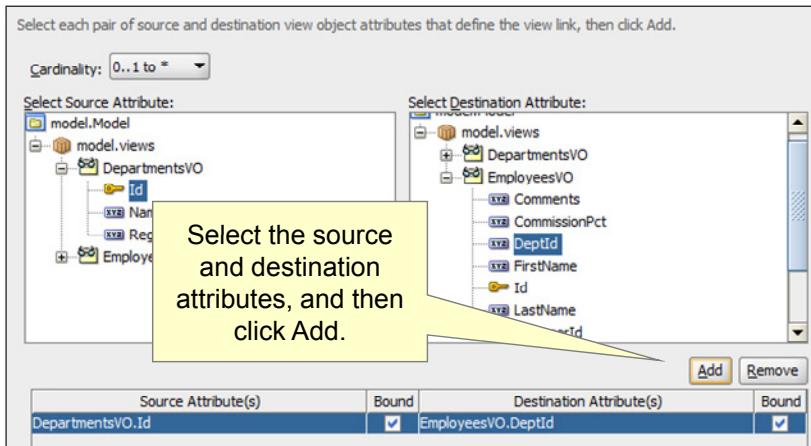
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Earlier, when you learned about view objects, you learned that a view link represents the relationship between two view objects, which is usually (but not necessarily) based on a foreign-key relationship between the underlying data tables. The view link associates a row of one view object instance (the master object) with one or more rows of another view object instance (the detail object). You can link a view object to one or more others to create master-detail hierarchies of any complexity. Some common types of master-detail relationships that you can create include:

- **Master to single detail:** One master and a single detail. For example, you might have a customer with a single credit rating.
- **Master to multiple details:** One master and multiple details. For example, you might have a customer with multiple orders.
- **Cascading master-detail:** Any type of master-detail relationship of any depth. For example, you might have a customer with multiple orders, and each order might have multiple items.

When you expose master-detail relationships in your application, the view links that you define between view objects enable the framework to synchronize the data across all the related components.

Linking View Objects

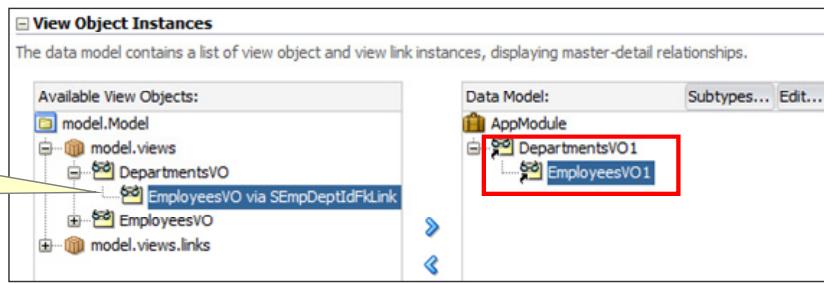


New > View Link

Rule: When possible, select an association to define the view link.

Edit Application Module

Add the master-detail hierarchy to the data model.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you use the "Create Business Components from Tables Wizard" to create business objects, JDeveloper automatically creates view links based on associations that exist between entities.

To create view links manually, you can use the Create View Link Wizard to do the following:

- Create a view link that is based on an existing association between entity objects (the source and target view objects must be based on entity objects that are related through an association). As a best practice, you should create view links that are based on associations whenever possible.
- Create a view link that defines how the source and target view objects are related.

To use the Create View Link Wizard:

- Create a view object that queries the master table, and create another view object that queries the detail table.
- Right-click the package where you want to create the view link, and select New > View Link.
- Specify a name for the view link. The name should identify how the link will be used. For example, you might use the name SEmpDeptIdFKLink to identify the link between a department list and its employees when the link is based on a foreign key.

4. Select the source (master) attribute and the destination (detail) attribute that defines the view link, and click Add to create the source-destination pair. Remember that, if an association exists, you should select it for the source and destination attributes. (To make it easier for you understand how the view objects shown in the slide are related, the example shows a view link created by selecting attributes rather than associations. However, in the real world, for the kind of example that is shown in the slide, you would select an association to define the relationship, rather than attributes.) 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.
5. Expose the master-detail hierarchy in the application module. To do this, open the application module in the editor and then, on the Data Model tab, add the master view object instance to the data model, followed by the detail view object instance via the link that you have defined.

Model-Driven List of Values (LOV)

An LOV:

- Specifies a list of valid values for an attribute
- Is defined on view object attributes
- Is based on one or more view object attributes
- When bound to a page, displays in an appropriate UI component that presents the list of valid values
- Makes it easier for the user to enter values
- Helps protect the application from invalid values



ORACLE

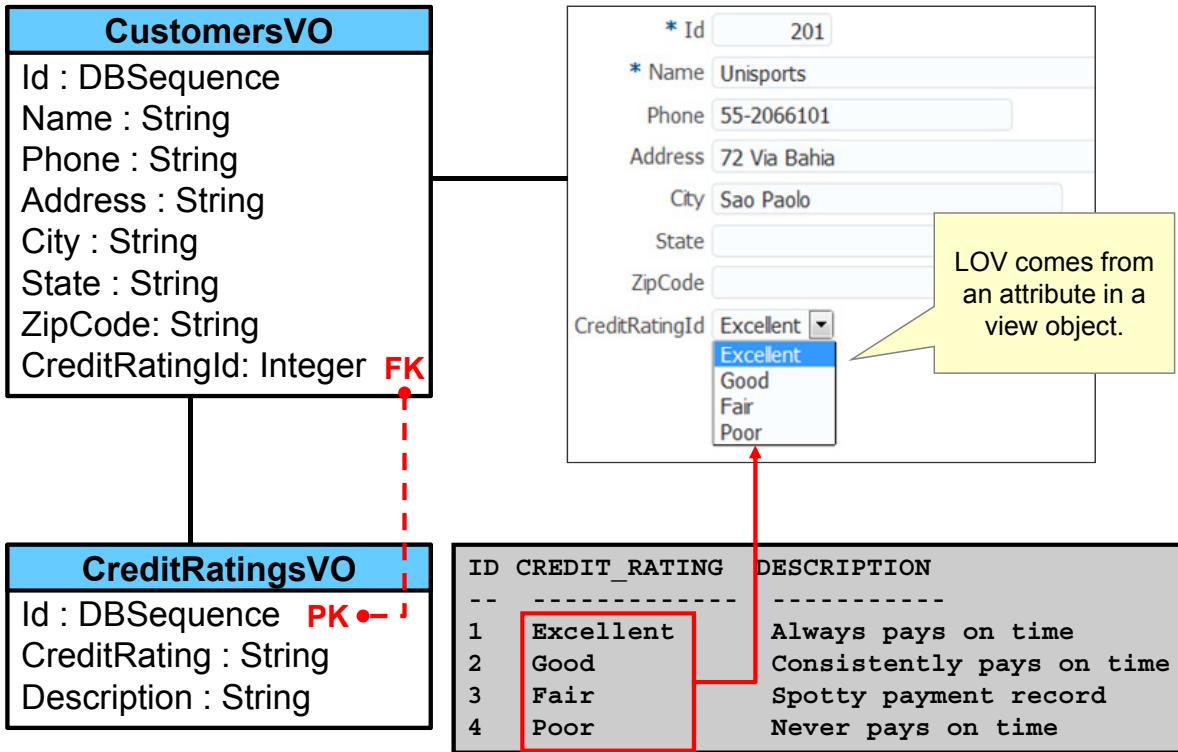
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When building an application, you might want to constrain data entry by allowing users to select from a list of valid values. For example, you might have a drop-down list of credit ratings, or a list of product codes in a filterable dialog box. With ADF Business Components, you can define a list of values for a specific view object attribute and then bind the attribute to an ADF Faces page. When the page appears, the list of values is displayed using an appropriate UI component that presents the list of valid values.

You define an LOV on a view object attribute. For example, the slide shows an LOV that is defined on an attribute called CreditRatingId. The source for the values in the LOV is a view object attribute. You can use an attribute from an existing view object, or you can create a new view object that contains the attribute you want to use. The view object can be based on a query or a static list of values.

Specifying an LOV makes it easier for users to enter values, and it helps protect your application from invalid values.

Model-Driven List of Values: Example



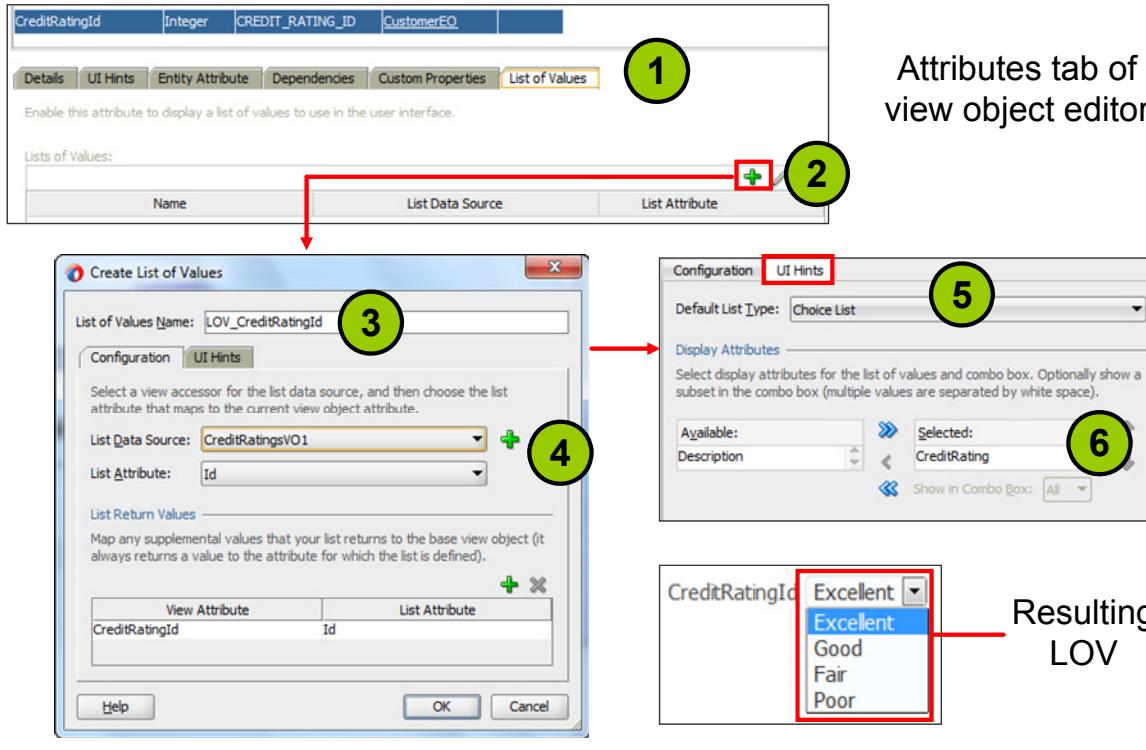
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, you see a view object named CustomersVO that contains several attributes. All the attributes in the view object are exposed in the data model and bound to the page as input fields. The CreditRatingId attribute in CustomersVO specifies an integer value that represents the credit rating of the selected Customer (Unisports, in this example). Notice that CreditRatingId is a foreign key that points to the Id attribute in CreditRatingsVO. Within CreditRatingsVO, the string value of the credit rating is specified by the CreditRating attribute. The page displays an LOV that is based on the Id attribute in CreditRatingsVO. However, the Id is not very meaningful to users, so the LOV uses the CreditRating attribute, rather than the Id, for the display attribute.

The next slide describes how to create the LOV shown in this example.

Defining the LOV



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To define a list of values:

1. In the view object editor, click the Attributes tab and select the attribute for which you want to define the LOV (CreditRatingId, in the example). Then click the “List of Values” tab.
2. Under “Lists of Values,” click the “Add List of Values” (green plus sign) icon.
3. Specify a name for the LOV. LOVs usually begin with LOV_.
4. Define where the list of values is coming from:
 - a. Next to List Data Source, click the Create New View Accessor (green plus sign) icon to open the View Accessors dialog box. From the Available View Objects list, shuttle the view object that contains the source for the LOV into the Selected list and click OK. (In the example, the view object is CreditRatingsVO, so the view accessor that you create is CreditRatingsVO1.)
 - b. Next to List Attribute, select the attribute that will be the source for the LOV. In the example, the list attribute is Id. Under List Return Values, notice that JDeveloper creates a mapping between the view attribute (the attribute that you expose on your page) and the list attribute (the source for the LOV).

5. Click the UI Hints tab, and then select the default list type (Choice List, in this example). This is the default UI control that will be used to display the list. This can be overridden in the UI.
6. Under Display Attributes, select the attributes that you want to display in the list. Notice that the display attribute does not need to be the attribute that is used to return the list values. So, in the example, you use the Id attribute to retrieve the list, but (to make the experience better for the user) you display the list of credit ratings instead of Ids. You can select multiple display attributes, which will be separated by spaces.

Cascading (Dependent) LOVs



Selecting an option in one component...

...filters the LOV in another component.

How is this different from defining a regular LOV?

- View criteria are used to filter the dependent list.
- Dependencies are defined on the dependent attribute.
- Autosubmit is set to TRUE on the first UI component.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

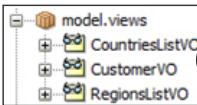
A cascading LOV (also known as a dependent LOV) is a list of values that is dependent on a value selected in another component. For example, you might want to provide a page that contains a selection list of regions. When the user selects a specific region, you want another selection list to display all countries that are valid for the selected region. The example in the slide shows two Select One Choice components that show a cascading relationship, but of course you can use any type of component that displays selection lists.

The procedure for defining a cascading LOV is similar to the procedure that you learned earlier. However, when you create a cascading LOV, you also need to do the following:

- Create view criteria to filter the list of values that appears in the dependent list (CountryId list) based on the value (RegionId) selected in the first list.
- In the view object for the dependent list, specify that the attribute (CountryId) has a dependency on another attribute (RegionId).
- Set the value of the Autosubmit property on the first UI component (RegionId list) to TRUE so that the framework updates the dependent list when the user selects a different value in the other component. (You learn more about the Autosubmit property later in the course.)

Defining Cascading LOVs

Create view objects.



1

Create view criteria.

2

Create LOVs on attributes.

3

• LOV_RegionId
• LOV_CountryId

Apply view criteria.

4

Set dependencies.

5

Set AutoSubmit to true.

6

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To create cascading LOVs:

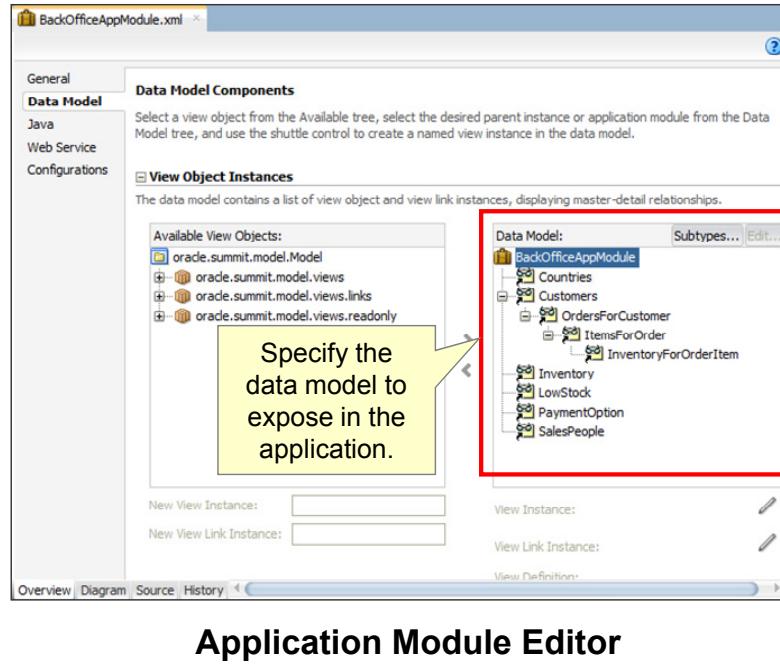
1. Create (or use) a view object for each list of valid values that you want to return. In the example, the names of the view objects are RegionsListVO and CountriesListVO.
2. Create a view criteria that will filter the dependent LOV based on the value selected in another component. You create the view criteria on the view object that is used to return valid values for the list. Use a bind variable to parameterize the view criteria so that you can pass in the value at run time. For example, for the dependent LOV in the previous slide (the list of countries), you would create a view criteria that returns all countries in the selected region.
3. In the main view object (CustomersVO), on the “List of Values” tab, create LOVs on the attributes that will be used to display the lists (RegionID and CountryID).
4. Apply the view criteria to the view accessor for the dependent list:
 - a. In the Accessors tab of the view object editor, select the view accessor for the dependent list (CountriesListVO) and click the pencil icon to edit the selected view accessor.
 - b. On the View Object page, under View Criteria, shuttle the view criteria you created for the dependent list (CountriesByRegionViewCriteria) into the Selected list.

- c. Under Bind Parameter Values, specify the attribute value (RegionId) that you want to pass in at run time. In this example, you specify RegionID because you want to filter the list of countries based on the value that is selected in the list of regions.
5. On the Attributes tab of the main view object (CustomersVO), select the dependent attribute (CountryId). Then, on the Dependencies tab, select the attribute on which the attribute depends (RegionId).
6. After exposing the LOVs on the page (as Select One Choice components in the example), select the component on which the dependent list depends (the RegionId list). Set the Autosubmit property to True. Setting this property enables the framework to update the dependent list when the user selects a different value in the other component.

Modifying Application Modules

Use the editor to:

- Set tuning parameters
- Refine the data model
- Specify nested app modules
- Create Java classes
- Enable a service interface
- Specify runtime properties



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

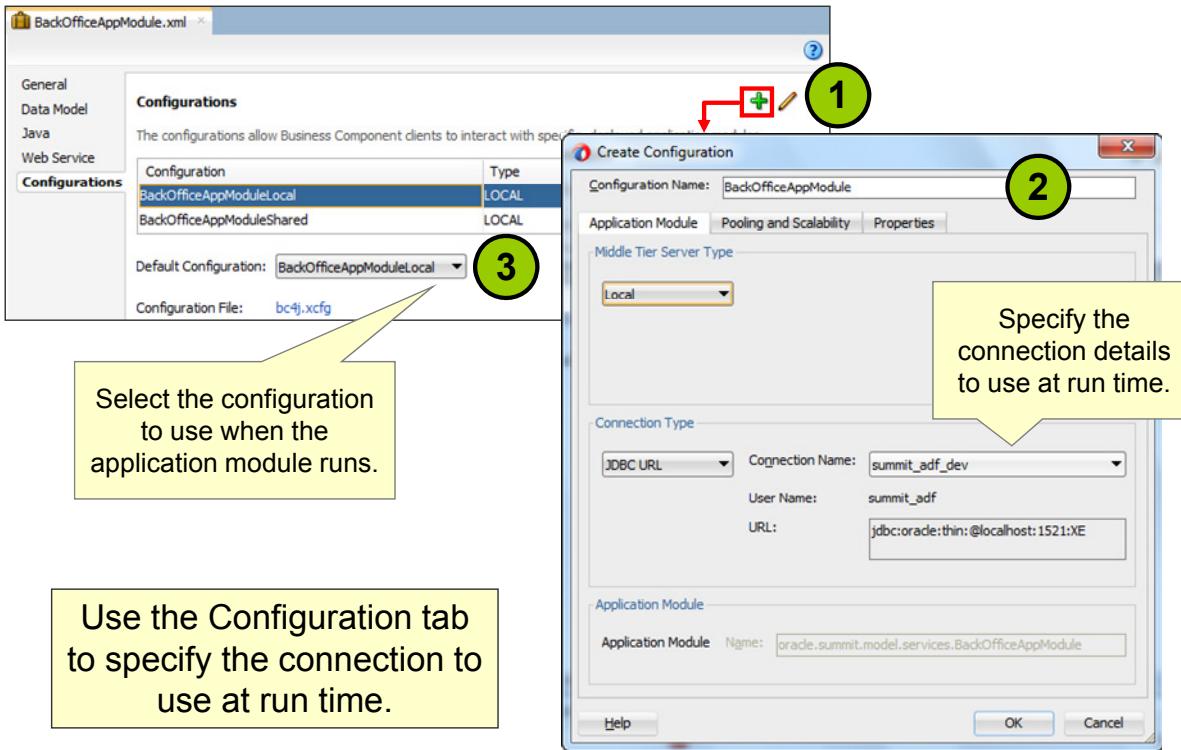
In an earlier lesson, you learned that application modules define the updateable data model and service methods that UI clients use to work with application data. You also learned that the application module provides a single connection to the database and handles all database transactions.

In the Applications window of JDeveloper, the application module appears as a single artifact. Double-click the application module to open it in the editor.

In the application module editor, you can:

- Set tuning parameters and define custom properties (General tab)
- Refine the data model by specifying view object instances and view link instances to expose in the application (Data Model tab)
- Specify nested application module instances (Data Model tab)
- Create Java classes and expose methods to the client interface (Java tab)
- Create a web service implementation wrapper and other files that are required to expose the application module as a web service (Web Service tab)
- Define your application module's database connection and runtime properties (Configurations)

Changing the Database Connection



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you create business components for an application, you specify a named database connection that is stored in the project properties of the model project. However, the connection that the application module uses at run time is defined on the Configurations tab of the application module editor. On the Configurations tab, you can define multiple configurations for an application module so that you can use a specific database connection (or JDBC data source) or alter the runtime properties for testing purposes. By default, JDeveloper creates a configuration called *AppNameLocal* that is based on the named connection that you defined for the business components.

To create and use a different configuration at run time:

1. On the Configurations tab of the application module editor, click the Create New Configuration Objects (plus sign) icon and specify a name for the configuration.
2. For the connection type, either choose JDBC URL and select a connection that is already defined for the project, or choose JDBC DataSource and specify a connection string for the data source. Specifying a data source instead of a JDBC URL enables you to tune, reconfigure, or remap the connection without changing the deployed application.
3. In the Default Configuration list, select the configuration that you want to use when the application module runs. You can use either connection type to run the Oracle ADF Model Tester and test an application module.

Determining the size of the Application Module

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

- An application module is a logical unit of work.
- Let use cases drive application module decisions
- For example, you can group application modules by:
 - The domain business objects that are involved
 - The user-oriented view of business data that is required
- Consider the possibility of reuse of the application module.
- Consider service or transaction flow.



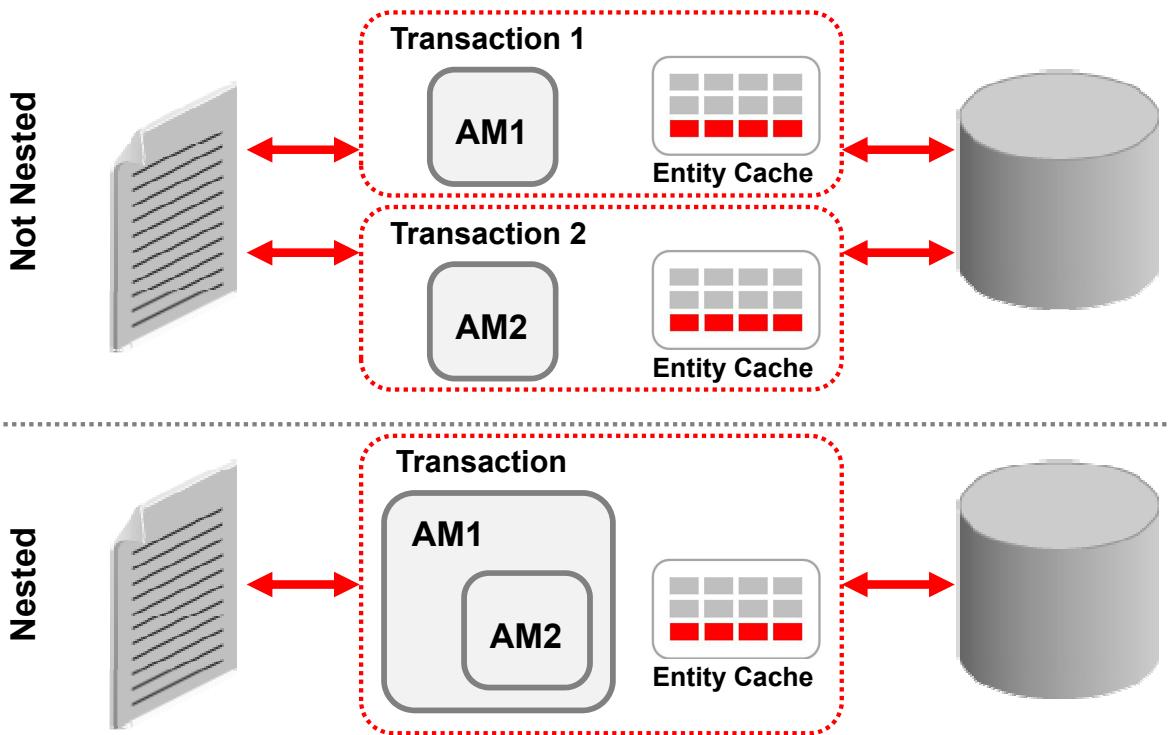
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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 that is 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 is the subset of columns, filtered set of rows, method of sorting, and method of grouping that support the use case?

Application Module Nesting



ORACLE

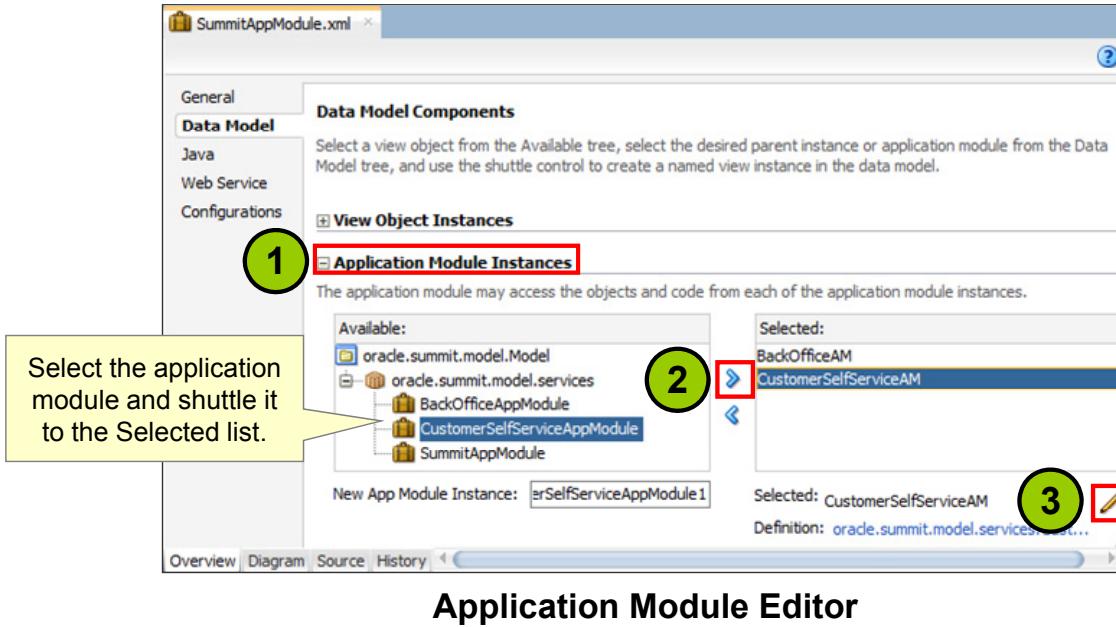
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As you learned earlier, an application module represents a logical unit of work. Applications are typically composed of multiple application modules, with each application module exposing a specific business use case. An application module sometimes needs to consume services that are exposed by other application modules. For example, an application module might need to reuse the business logic that is exposed by other application modules. You can implement this modularity by nesting all required application modules under a root application module. All nested application module instances share the same transaction and entity object cache as the root application module.

The slide compares an application that does not use nested application modules (top diagram in slide) to an application that does (bottom diagram in slide). In the top diagram, notice that each application module runs in its own transaction and has its own entity object cache and database connection. Each transaction must be committed and rolled back separately. Because the application modules use different entity object caches, the application modules might run into problems if they try to update the same database tables.

In the bottom diagram, however, notice that the second application module, AM2, is nested in AM1. Both application modules share the same transaction, entity object cache, and database connection. Any changes to the database are made in a single transaction that can be rolled back or committed at any time. Because the application modules share the same entity cache, the data is synchronized. And, of course, the application requires only a single connection to the database, which incurs less overhead.

Defining Nested Application Modules



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You define nested application modules on the Data Model tab of the application module editor.

To define application module nesting:

1. On the Data Model tab, expand Application Module Instances.
2. In the Available list, expand the hierarchy to see the available application modules. Select each application module that you want to access and shuttle it to the Selected list.
3. If you want to rename the selected application module instances, select an instance and click the pencil icon to rename the instance.

Summary

In this lesson, you should have learned how to:

- Declaratively modify entity objects, view objects, and application modules
- Create LOVs
- Create nested application modules



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 6 Overview: Declaratively Customizing ADF BC

This practice covers the following topics:

- Changing data labels and formats
- Specifying that an entity object should use a database sequence to generate the primary key
- Creating view criteria
- Creating a join view object
- Creating a static view object
- Defining an LOV
- Creating a transient attribute that has a calculated value



Copyright © 2014, 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 NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

Validating User Input

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

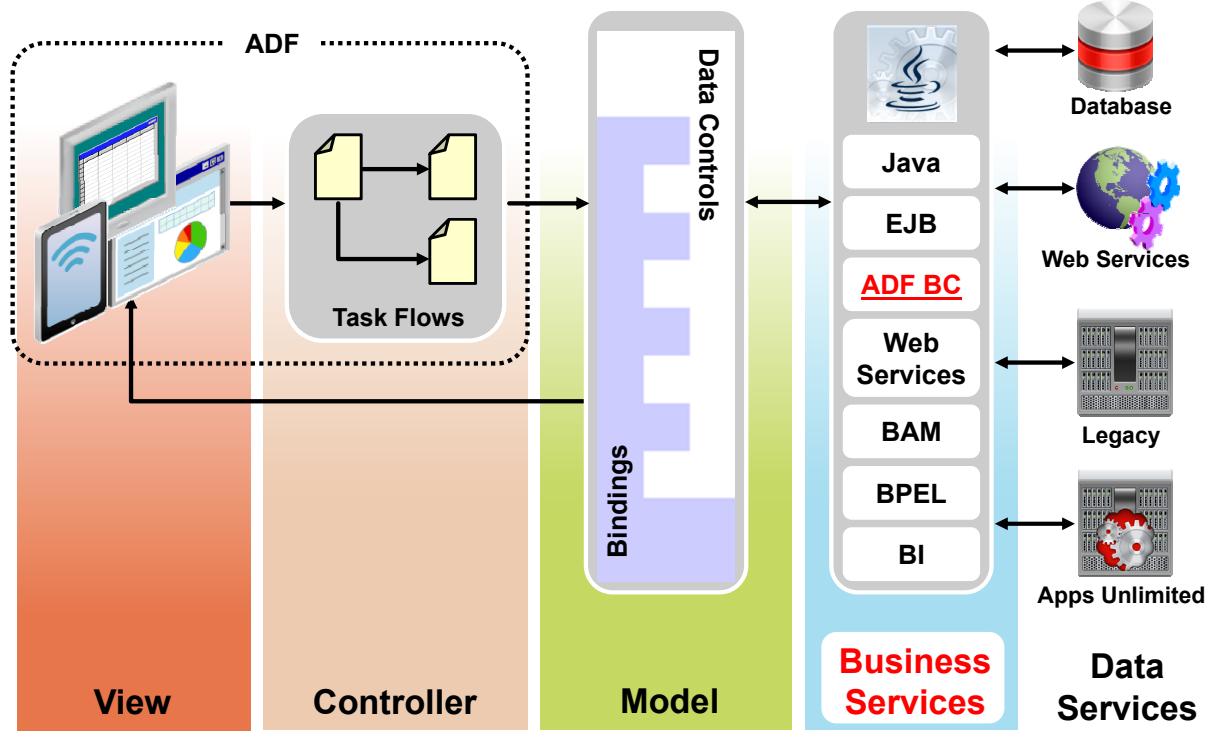
After completing this lesson, you should be able to:

- Describe the validation options in ADF BC applications and when to use them
- Decide when to use declarative validation and when to use “method” declarative validation
- Add validation to the user interface
- Write Groovy expressions to use for validation
- Internationalize messages by creating a resource bundle



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Adding Validation to Business Components

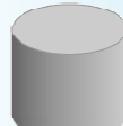


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

In the previous lesson, you learned how to build a richer business services layer for your application by customizing the default behavior of ADF Business Components. In this lesson, you learn how to make your business services layer even more robust by adding validation to ADF Business Components.

Validation Options for ADF BC Applications



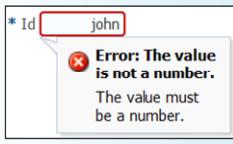
Database

- Add validation to the database:
 - Not exposed in the UI by default



ADF BC

- Add validation to business services:
 - Use declarative validation framework in ADF BC.
 - Use programmatic validation for complex validation rules.



ADF Faces UI

- Add validation to the UI:
 - Use ADF Faces input components that have built-in validation capabilities.
 - Equivalent validation is required in the business services layer.

ORACLE

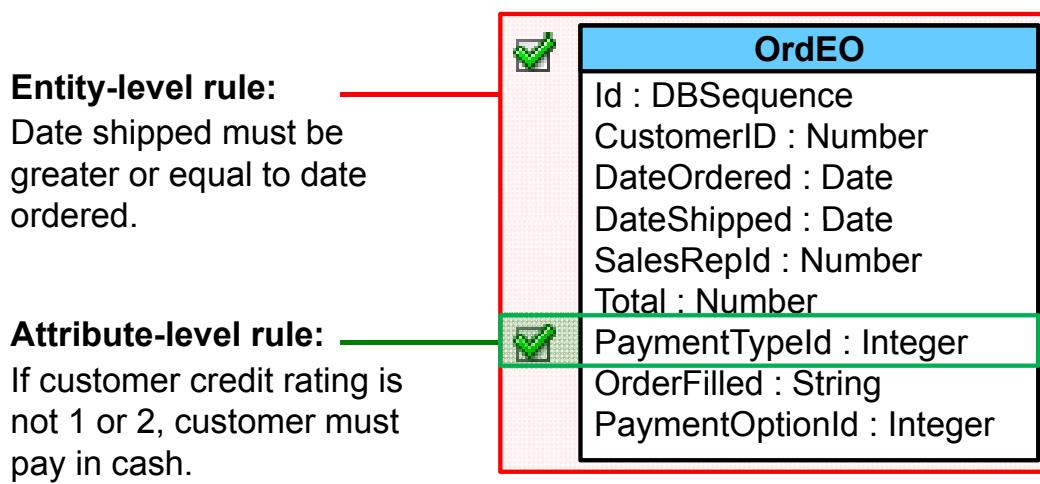
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In an ADF BC application, you can add validation in the following places.

- **Database (PL/SQL):** At the most basic level, you can implement validation in the database. However, such validation is not exposed in the UI by default. For example, you cannot use database validation to expose a list of valid values in the UI.
- **Business services:** ADF Business Components provides a declarative validation framework that allows user input to be validated against a static value, a list of values, a numeric range, a Groovy expression, and even values returned from queries. If you choose to write more complex validation rules in Java, the framework provides a convenient way to hook up the validation mechanism to a Java method. You should build validation into the business services layer to protect the integrity of the data in the database and to prevent malicious attacks. This lesson focuses on adding validation to this layer.
- **User interface:** The ADF Faces input components that you use to build the user interface have built-in validation capabilities. You can use prebuilt ADF Faces validators to ensure that data submitted through the UI 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.

Defining Validation Rules in ADF BC

- **Entity-level rules:** Fire when data is committed; used when validation depends on more than one attribute
- **Attribute-level rules:** Fire when the attribute value changes



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you define validation rules in ADF BC, you define them as part of the entity object. Encapsulating validation 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

You apply validation rules to the entity object itself or to individual attributes in the entity object:

- Entity-level validation rules (known as Entity Validators) fire when either the row changes or data is committed. You apply a validation rule at the entity level when the rule depends on more than one attribute value. For example, you might have a rule for an OrderEO entity object that says that the order date must not come before the shipped date. Because this rule requires values from two different attributes, you would define it at the entity level.
- Attribute-level validation rules fire whenever the attribute value changes. For example, you might have a rule for the PaymentTypId attribute in OrdEO that checks the customer's credit rating (in another table) and requires the payment type to be cash if the credit rating is not equal to 1 or 2.

Validation Rule Types: Entity and Attribute-level

Validator Type	Description
Compare	Validates an attribute by comparing it to a literal value, another attribute, or the result of a query or expression
Length	Compares the character or byte length of an attribute value to the specified size
List	Validates an attribute against a list of possible values
Key exists	Determines if the specified key value exists
Range	Tests for attribute values within a specified range
Regular expression	Validates an attribute value against a mask specified by a Java regular expression
Script expression	Calls a Groovy expression to perform validation
Method	Calls a Java method to perform validation



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The easiest way to create and manage validation rules is through the declarative validation framework. Oracle ADF provides a number of built-in validation rule classes (known as *validators*) that enable you to add business logic without writing a single line of code. You implement validation rules through the entity object editor, and they are stored in the entity object's XML file.

This slide shows the validator types that can be applied at both the entity level and the attribute level.

Validation Rule Types: Entity-Level Only

Validator Type	Description
Collection	Computes an aggregate value on an attribute by using an aggregate function such as Sum, Average, Count, Min or Max, and compares the result to a literal value, another attribute, or the result of executing a query or expression
Unique Key	Validates that the primary key for an entity object is unique; checks the entity cache and the database to ensure that the key is unique

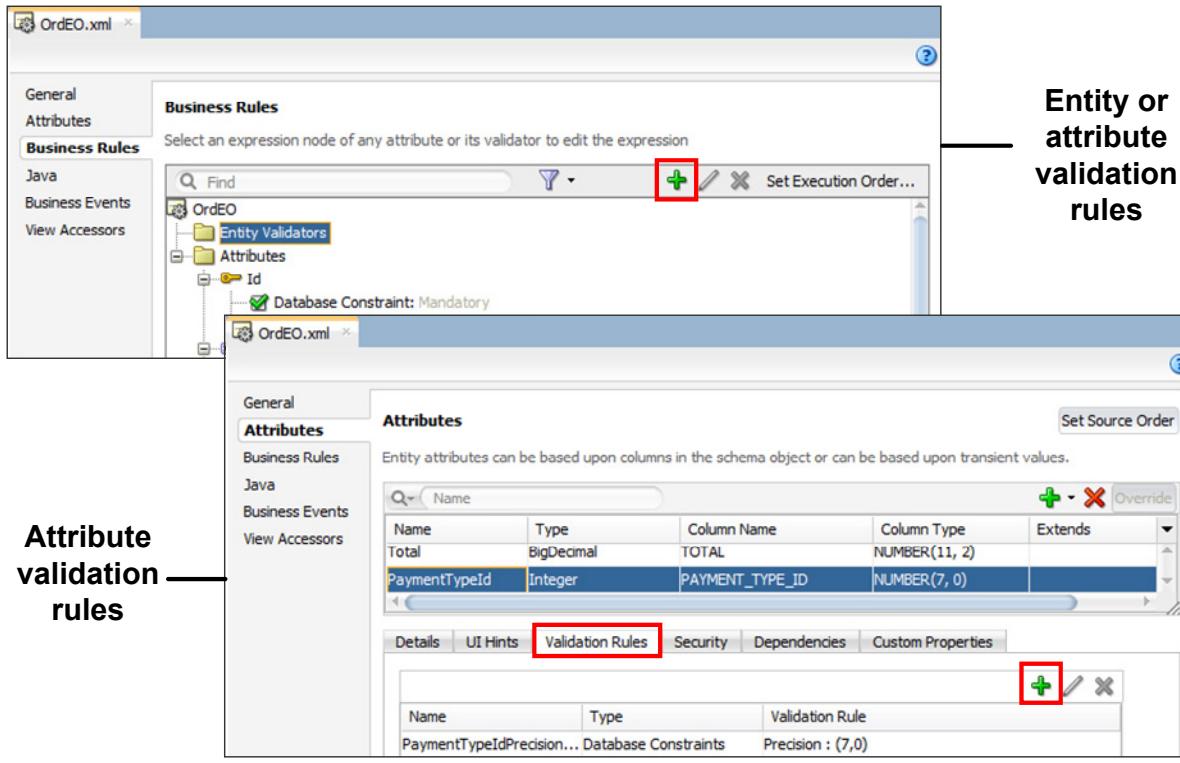


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The two rules shown in the slide are valid at the entity level only.

The next few slides show how to use the built-in declarative validation rules.

Creating Validation Rules



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Because validation rules are specified on entity objects (or attributes within an entity object), you define validation rules by using the entity object editor.

To create a validation rule:

- **For entity-level validation**

In the entity object editor, click the Business Rules tab and select the Entity Validators folder. Then click the Create New Validator (green plus sign) icon to define the rule.

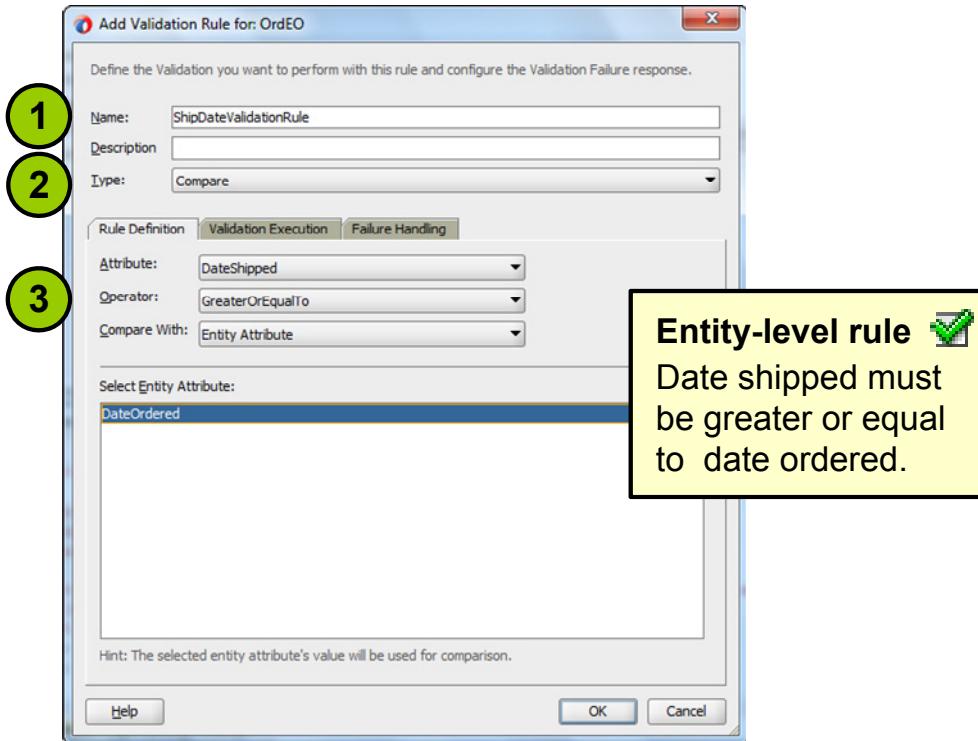
- **For attribute-level validation**

Do either of the following:

- On the Business Rules tab, under the Attributes folder, select the attribute that requires validation. Then click the Create New Validator (green plus sign) icon to add a rule.
- On the Attributes tab, select the attribute that requires validation, and click the Validation Rules tab. Then click the Add Validation Rule (green plus sign) icon to add a rule.

Next, you learn how to specify the details of the validation rule.

Specifying the Rule Definition



ORACLE

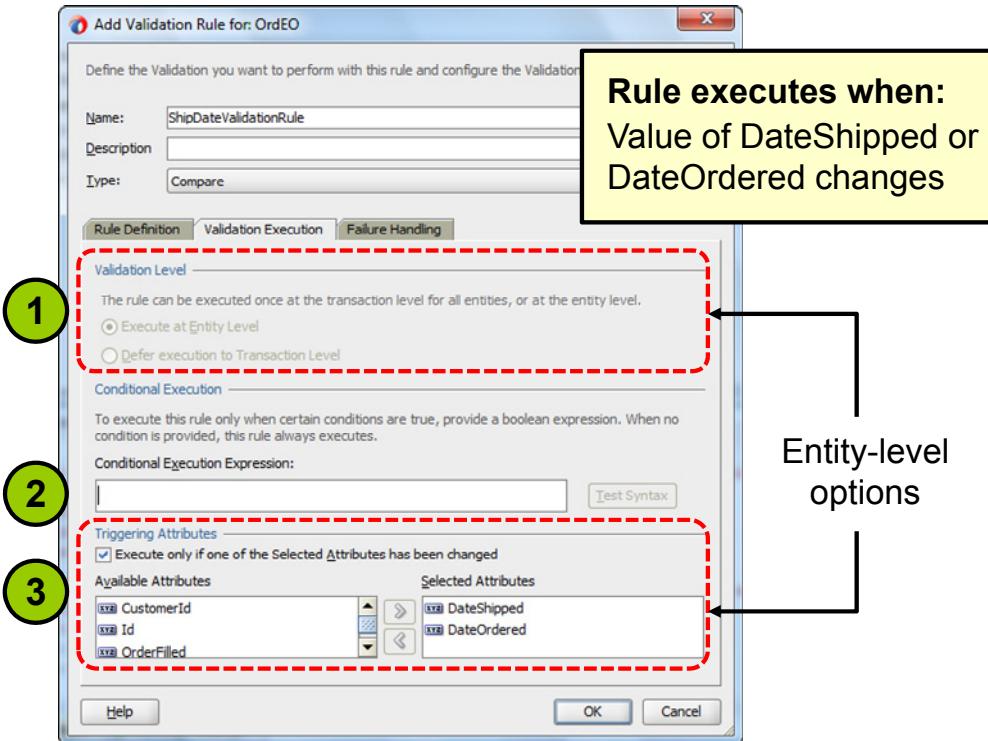
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the Add Validation Rule dialog box, you specify the details of the validation rule:

1. Specify the name of the validation rule and an optional description.
2. In the Type list, select the type of validator that you want to use. (See the previous slides for a description of each validator.) The options that appear on the Rule Definition tab change according to the type of validator that you select. In the example, the Compare validator is selected.
3. On the Rule Definition tab, define the details of the comparison:
 - a. In the Attribute list, select the attribute to which you are applying the validation rule (DateShipped in the example).
 - b. In the Operator list, select an operator to use for the comparison (GreaterOrEqualTo in the example).
 - c. In the Compare With list, select an option that indicates how the comparison value will be derived. In the example, Entity Attribute is selected because the rule compares the value of DateShipped to the value of another attribute, DateOrdered. When you select Entity Attribute, the Select Entity Attribute box is populated with a list of attributes that have the correct data type for the comparison. For the comparison in the example, you select DateOrdered.

In the next slide, you learn how to specify the conditions that cause the validation rule to fire.

Specifying Conditions for Executing Validation Rules



ORACLE

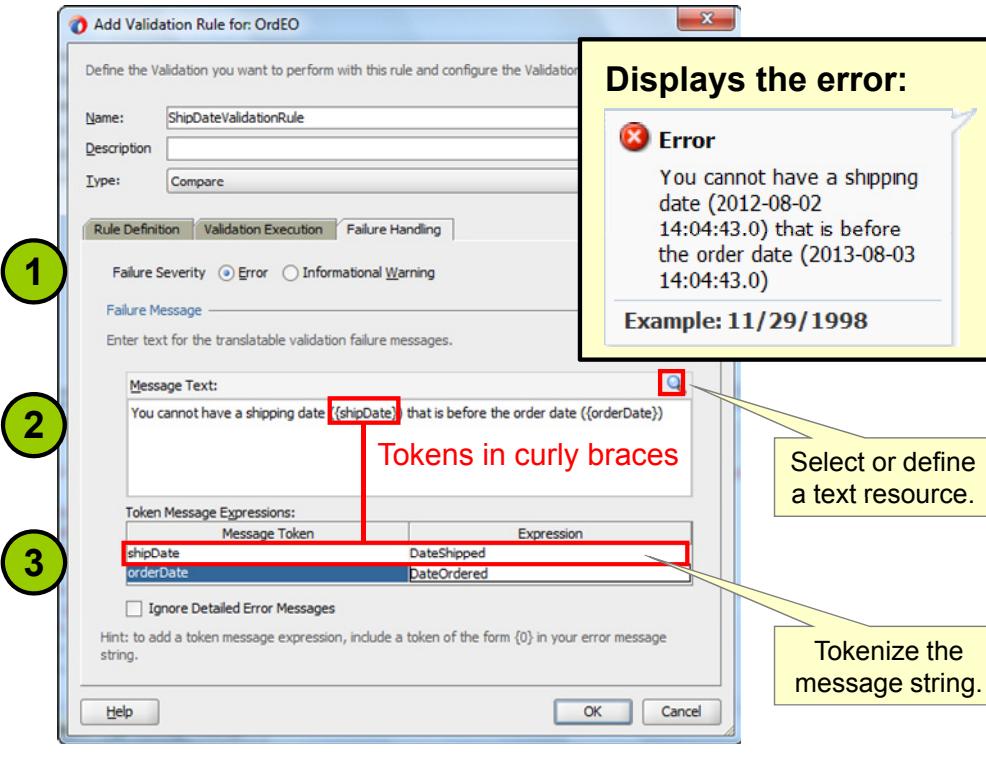
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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 before the entity is again considered valid.

On the Validation Execution tab, you can specify other conditions that control when validation is executed:

- Validation Level:** You can specify the validation level for entity-level validation rules that use Key Exists or Method validators. This setting specifies 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 other entity-level validation. For this reason, you should use this option when 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 of testing the expression to see if the syntax is valid.
- Triggering Attributes:** By default, a validator fires on an attribute whenever the entity as a whole is dirty. However, when an entity-level validation rule depends on the value of another attribute, you can choose to trigger validation when specific attributes change (when one of the triggering attributes is “dirty”).

Specifying Error Messages to Handle Failures



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

On the Failure Handling tab, you specify the error message that you want to display when the user enters an invalid value.

To specify failure handling:

1. Set the Failure Severity value to Error (default) or Informational Warning.
 - Error specifies that the end user must enter a valid value in the field before proceeding.
 - Informational Warning specifies that the user will see a warning message and be able to proceed after the message has displayed.
2. Specify the text to appear in the message. By default, JDeveloper adds the text that you specify to a resource bundle. Click the magnifying glass icon to select an existing text resource. You learn more about creating and using text resources later in this lesson.
3. If the message includes dynamic values, you can define tokens (enclosed in curly braces) within the message text. For each token, you must specify a Groovy expression that resolves to the expected value at run time. For example, you might want to use a token to represent a dynamic label, or (as in the example) you can use tokens to represent attribute values. To reference an attribute value in Groovy, you simply use the name of the attribute. You learn more about Groovy expressions next.

Introduction to 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
- Is used for values throughout the ADF BC model, including:
 - Default attribute values
 - Bind variables
 - Validation rules
 - Tokens in error messages

The screenshot shows a software interface for defining validation rules. At the top, there are tabs: 'Rule Definition' (which is selected), 'Validation Execution', and 'Failure Handling'. Below the tabs, a text area is labeled 'Enter the text for the validation expression. Click on Test to validate the syntax of your expression.' Underneath this, another text area is labeled 'Expression:' containing the following Groovy code:

```
cr = CustomerEO.CreditRatingId
ratings = [1, 2]
if (cr in ratings)
{return true}
else
{return false}
```

Groovy Script Expression
Used for Validation



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Groovy is an agile, dynamic language for the Java platform. It has many features that were inspired by languages such as Python, Ruby, and Smalltalk. Groovy interoperates seamlessly with any Java class and can be compiled and interpreted without disturbing normal operations.

You can use Groovy expressions for all sorts of declarative values, such as bind variables and attribute default values. For example, you can use a Groovy script that returns true or false for declarative validation. You can also use Groovy expressions in error messages (for example, to resolve the value of a message token).

Groovy supports object access via dot-separated notation, so it supports syntax like Empno instead of getAttribute(EMPNO). This notation simplifies expressions by making them more concise.

You can learn more about Groovy at <http://groovy.codehaus.org/>.

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>

Use the reserved name `adf` to get objects from the framework:

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



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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

There is one top-level reserved name, `adf`, to access 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.currentDate`: To reference the current date with time truncated
- `adf.currentTime`: 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.

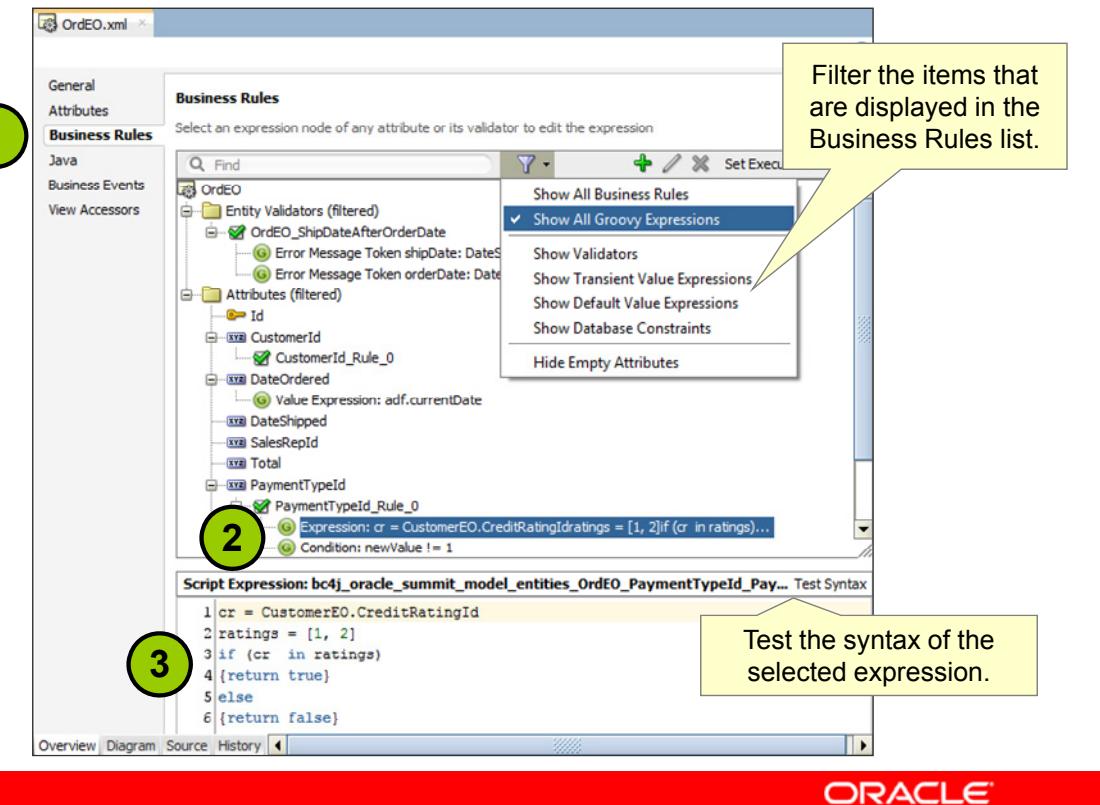
- **Transient attribute:** The context is the current entity or view row. You can reference all attributes by name in the entity or view row in which it appears, as well as any public method on that entity or view row. To access methods on the current object, you must use the `adf.object` keyword to reference the current object as follows:
`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 the entity (you do not use the `source` keyword simply to use an attribute value from an entity).

All Java methods, language constructs, and Groovy language constructs are available in the script.

Additional Tips

- 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 is 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")`
`rowSetAttr.count("GroovyExpr")`
`rowSetAttr.avg("GroovyExpr")`
`rowSetAttr.min("GroovyExpr")`
`rowSetAttr.max("GroovyExpr")`
- Use the `return` keyword just as 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 the SQL `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 the 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.

Editing Groovy Expressions



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

JDeveloper provides a Groovy editor that you can use to edit and debug Groovy expressions.

To use the editor:

1. On the Business Rules tab of the entity or view object editor, expand the hierarchy to find the expression that you want to edit. You can click the Rules Filter icon to filter the view if you like.
2. Select the Groovy expression that you want to edit.
3. In the Script Expression editor, modify the Groovy expression as required. Notice that you can test the syntax after making changes.

Later, when you learn about debugging, you learn how to set breakpoints on Groovy snippets in the editor.

Internationalizing Messages and Other Translatable Text

- Internationalization is the process of configuring support for multiple locales.
- Localization is the process of adding support for a specific locale.
- JDeveloper provides support for storing text strings (labels, error messages, tooltips, and so on) in resource bundles.
- ADF Faces provides pre-translated components (in 28 languages) that you can override, as required.

The figure shows two JDeveloper interface screenshots side-by-side. The left screenshot is titled 'en_US Locale' and displays a table with columns 'Last Name', 'First Name', and 'Email'. The data rows are: Ngao, LaDoris, Ingao@summit.com; Urguhart, Molly, murguhar@summit.com; Gijum, Henry, hgijum@summit.com; Maduro, Elena, emaduro@summit.com; Smith, George, gsmith@summit.com. The right screenshot is titled 'es_ES Locale' and displays a similar table with columns 'Apellido', 'Primer Nombre', and 'Correo Electrónico'. The data rows are: Ngao, LaDoris, Ingao@summit.com; Urguhart, Molly, murguhar@summit.com; Gijum, Henry, hgijum@summit.com; Maduro, Elena, emaduro@summit.com; Smith, George, gsmith@summit.com. An arrow points from the 'en_US Locale' table towards the 'es_ES Locale' table.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Internationalization is the process of configuring an application for easy adaptation to specific local languages and cultures. *Localization* is the process of actually adapting the application to a specific locale by translating text and adding locale-specific components. You internationalize ADF applications by placing translatable strings (such as labels, error messages, and tooltips) in files called *resource bundles* that are separate from your application code. You handle date formats by using locale-sensitive formatting Java classes (as described at <http://docs.oracle.com/javase/tutorial/i18n/resbundle/prepare.html>).

Next, you localize the application for specific languages by providing translated versions of the resource bundles for each locale that you want to support. When the application loads in a browser, it displays translated strings based on the language setting of the user's browser. So, for example, if you know that your application will be viewed in a Spanish-speaking country, you can localize your application so that the text strings in the application appear in Spanish when a user's browser is set to use the Spanish language. You can, of course, change this default behavior by setting the locale programmatically. For example, you can write application code that sets the locale for a specific user or even enables a user to select the locale on the fly.

To help with localization, ADF Faces provides pre-translated components. The resource bundles used for the components' skin (which determines the look and feel as well as the text within it) are translated into 28 languages. You can use the translations that are provided, or you can provide your own resource bundles to override some or all existing translations.

It's recommended that you internationalize your application by storing all translatable strings in resource bundles even if you have no plans to support multiple locales. When you internationalize your application, you should also make sure that you use images and icons that are appropriate in the target locale. When possible, use images and icons that do not contain text.

Steps to Internationalize an Application

1. Create a base resource bundle (or bundles).
2. Create a localized resource bundle for each locale that you plan to support.
3. Configure the application to support specific locales.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To internationalize and then localize your application:

1. Create a base resource bundle (or bundles) in the default language of your application. Resource bundles should contain all the translatable text strings except the pre-translated boilerplate text in the ADF components. The bundles that you create should be in the default language of the application.
2. Create a localized version of the bundle for each locale supported by the application.
3. Configure the application to support specific locales.

Each step is described in more detail in subsequent slides.

Resource Bundles

```
LastName_LABEL=Last Name
FirstName_LABEL=First Name
UserId_LABEL=User ID
Email_LABEL=Email
DeptId_LABEL=Dept ID
```

Properties Bundle (.properties)

```
<?xml version="1.0" encoding="windows-1252" ?>
...
<body>
    <trans-unit id="LastName_LABEL">
        <source>Last Name</source>
        <target/>
    </trans-unit>
    <trans-unit id="FirstName_LABEL">
        <source>First Name</source>
        <target/>
    </trans-unit>
...

```

XLIFF Bundle (.xlf)

```
...
public class ModelBundle extends ListResourceBundle {
{
    private static final Object[] contents = {
        { "LastName_LABEL", "Last Name" },
        { "FirstName_LABEL", "First Name" },
        { "UserId_LABEL", "User ID" },
        { "Email_LABEL", "Email" },
        { "DeptId_LABEL", "Department ID" },
    };
}

public Object[] getContents() {
    return contents;
}
}
```

List Bundle (.java)

Example:

Key	LastName_LABEL
Value	Last Name



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A resource bundle is a file that contains key-value pairs: the key is the name of the text resource that is used in the code, and the value is the translation—for example, LastName_LABEL=Last Name.

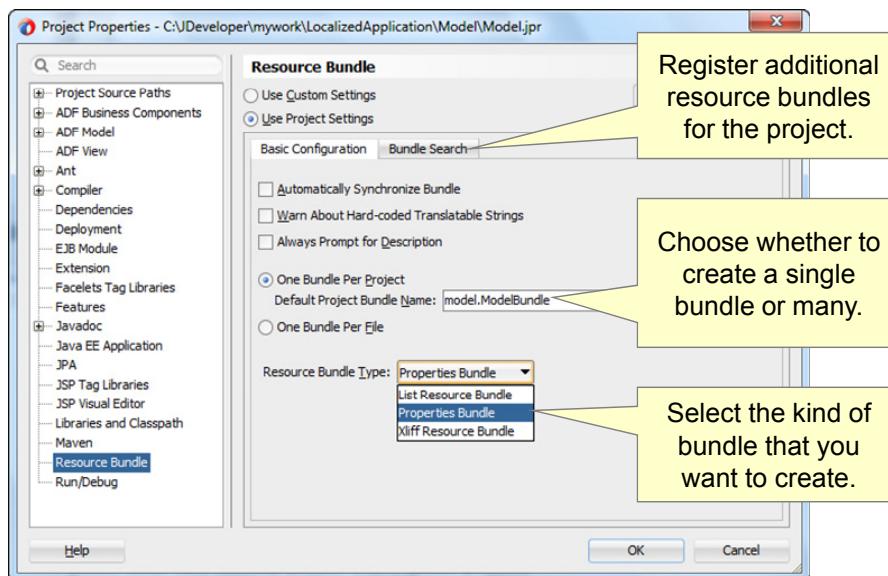
JDeveloper supports the following types of resource bundles:

- **Properties bundle:** Uses a properties file to manage the text resources for the project or page. The properties bundle is a plain-text file that specifies the key-value pairs. Properties files can contain values for String objects only. If you need to store other types of objects, you must use a list bundle instead. JDeveloper uses properties bundles by default.
Note: The ADF Skin Editor works with properties bundles only.
- **XML Localization Interchange File Format (XLIFF) bundle:** An XML-based format for exchanging localization data
- **List bundle:** A Java class that manages resources that are stored in a name-value array. You can store any locale-specific object in a list resource class.

This lesson describes how to create properties bundles. For more information about other types of resource bundles, see “Developing Applications with Oracle JDeveloper and Developing Web User Interfaces with Oracle ADF Faces” at <http://docs.oracle.com/middleware/1212/cross/developdocs.htm>.

Creating Resource Bundles: Setting Resource Bundle Options

Set project properties to control the resource bundles that are available to your projects.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

By default, JDeveloper creates one resource bundle for each project. For large applications, you might find it easier to organize your text resources into multiple files. You change this setting in the Project Properties window by selecting the Resource Bundle page. You can also select the type of resource bundle to create by default. (To open Project Properties, double-click the project in the Applications window.)

If you have additional resource bundles that you want to use, click the Bundle Search tab and register additional resource bundles. All registered resource bundles (including the default bundle) are available in the dialog boxes and windows in JDeveloper that allow you to specify text strings.

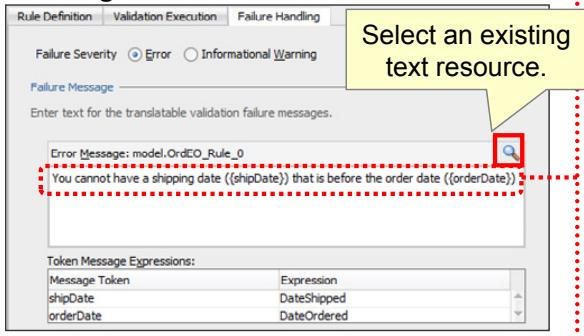
Important: Make sure that you change the resource bundle settings before you start defining text resources for your project. You can change the type of resource bundle used by a project at any time, and any new entries will use the specified resource bundle type. However, existing resource bundle entries will not be migrated to the new type. You must migrate them manually.

Creating Resource Bundles: Model

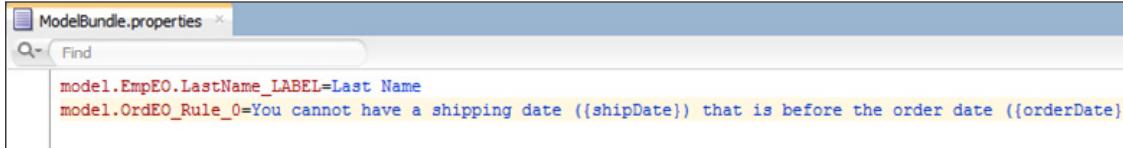
Attribute UI Hints



Message Text



Resource Bundle (Model)



- JDeveloper creates a default bundle the first time you enter translatable text.
- Name-value pair is added to the bundle automatically.
- Message text can include tokens.
- Click the magnifying glass icon to select a resource from an existing bundle.

ORACLE

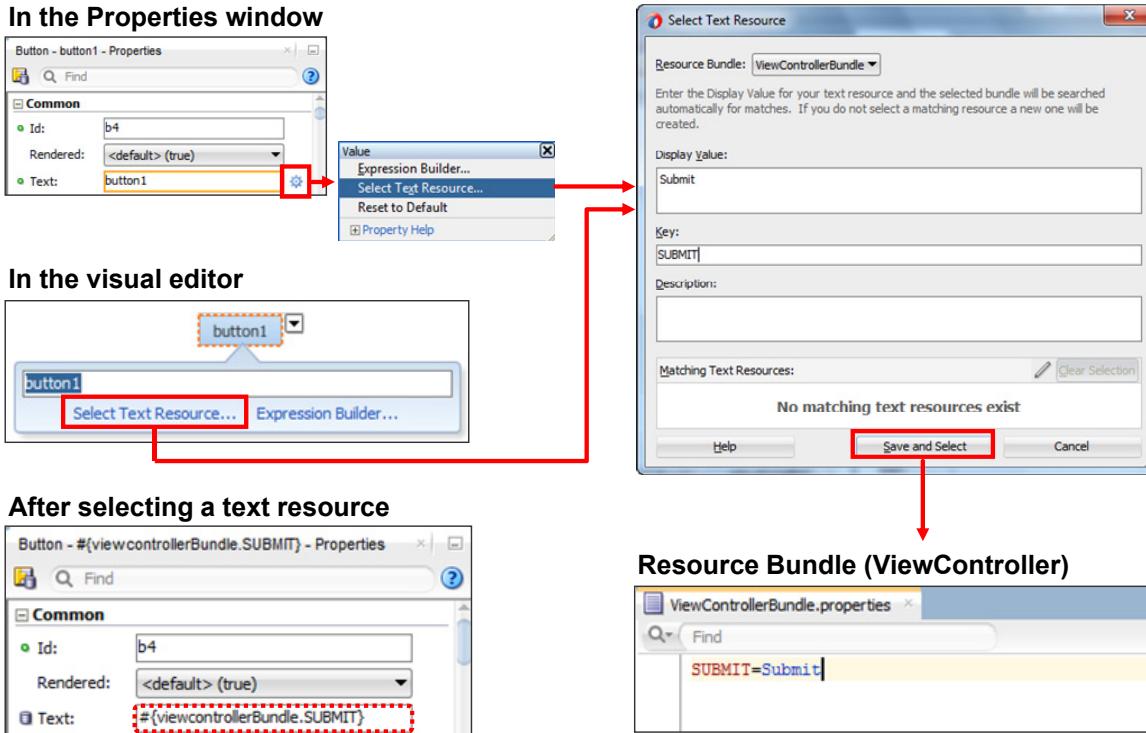
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You do not need to manually edit the resource bundle file to create a text resource (name-value pair). The JDeveloper IDE provides a simpler way to do this.

By default, JDeveloper creates a resource bundle the first time you enter translatable text (such as a UI control hint or a message) in an ADF business component editor. If you have accepted the default resource type, which is Properties Bundle, JDeveloper creates a file called `ModelBundle.properties`. JDeveloper automatically creates a text resource (name-value pair) and adds it to the default resource bundle.

However, if you want to select an existing text resource or create a text resource in a specific resource bundle, click the magnifying glass icon to add (or select) a text resource from a bundle.

Creating Resource Bundles: ViewController



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

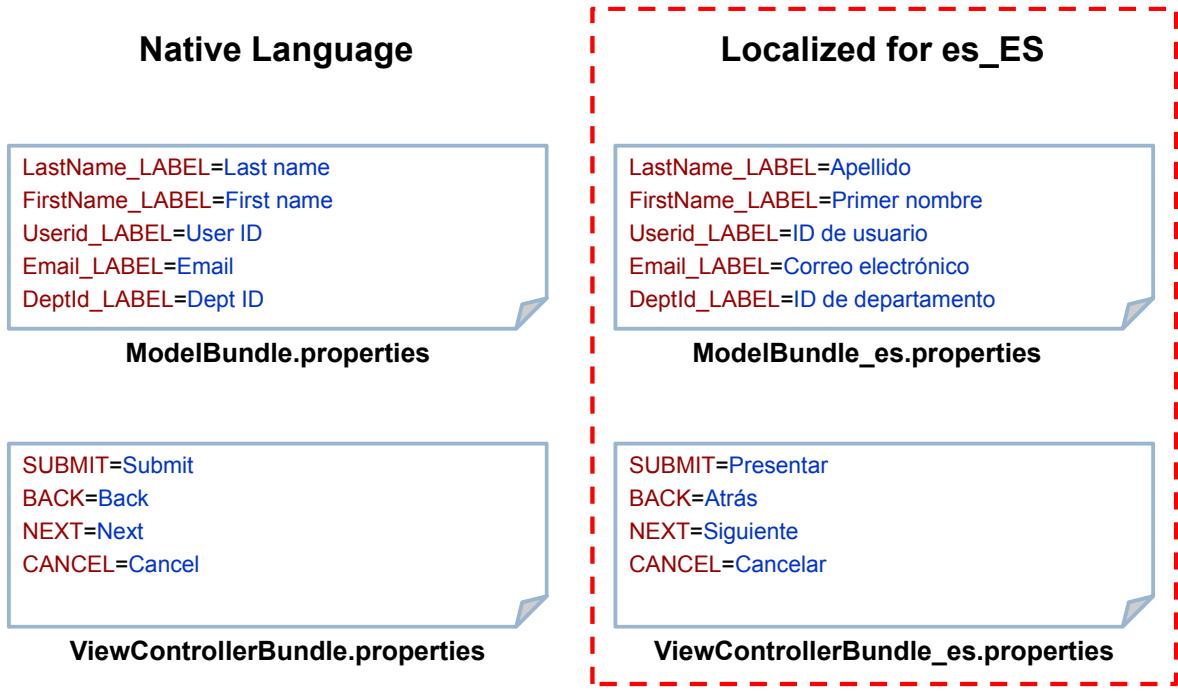
The process of creating resource bundles for the view controller project differs slightly from the model project. By default, JDeveloper creates a properties file called `ViewControllerBundle.properties` the first time you select (or add) a text resource to the project.

To select or add a text resource:

1. Do one of the following to invoke the Select Text Resource dialog box:
 - In the Properties window, place your cursor to the right of the text attribute for which you want to enter a text resource and click the Property Menu icon. From the pop-up menu, choose Select Text Resource.
 - In the visual editor, select the component and then click it again to invoke a pop-up box for selecting (or modifying) the text resource. Click Select Text Resource. This way of selecting a text resource is available only when the Automatically Synchronize Bundle option is selected on the Resource Bundle page of the Project Properties window.

2. In the Select Text Resource dialog box, you can either select an existing text resource or create a new one.
 - a. Enter a display value (the translatable text that will be displayed in the UI).
 - b. Enter a key (the default is based on the Display Value, but you can change it).
 - c. Enter a description of how the text resource is to be used.
 - d. Click “Save and Select” (or Select when choosing an existing text resource).
3. The text resource is added to the resource bundle. Notice that the visual editor shows the string that you entered, whereas the Properties window shows the EL expression that refers to the text resource in the resource bundle.
4. Use the resource bundle on the page.
 - a. Make sure that your page encoding is set to be a superset of all supported languages. By default, JDeveloper sets the page encoding to windows-1252. To set the default to a different page encoding, perform the following steps:
 1. Select Tools > Preferences.
 2. Select Environment from the list at the left.
 3. Set Encoding to the preferred default.
 - b. Bind all attributes that represent strings of static text on the page to the appropriate key in the resource bundle. When you use the JDeveloper facilities for adding and selecting text resources (as shown in the slide), the binding is created automatically for you.
 - c. Redeploy the application that uses the new resource bundle.

Creating Localized Resource Bundles



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You perform the internationalization process throughout development by storing translatable text strings in resource bundles. Localization of the application, however, should not occur until testing is complete in your application's default language.

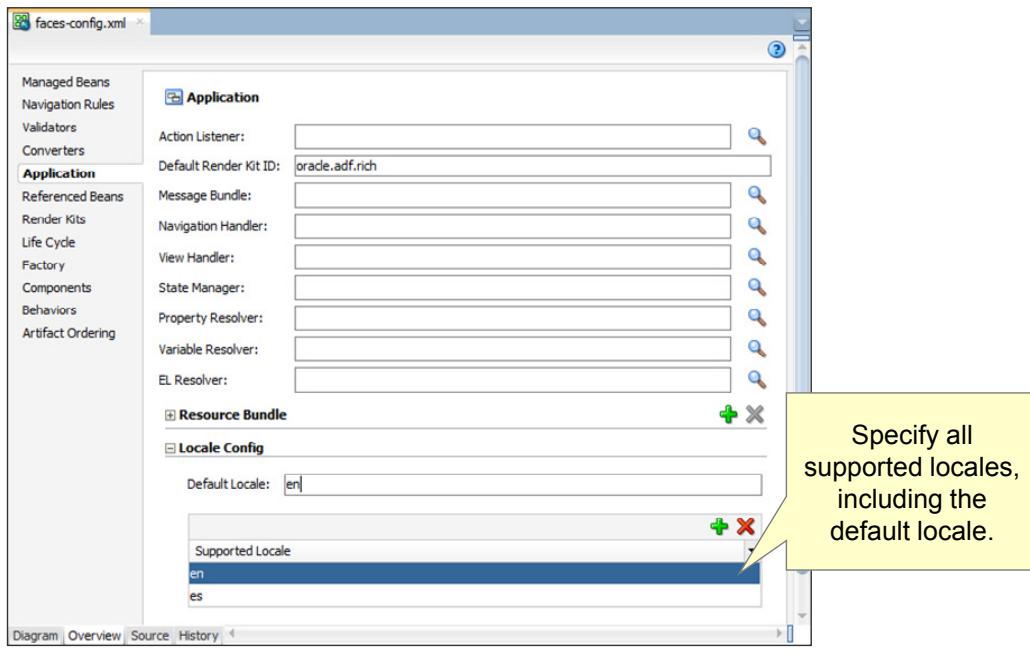
To localize the application, you create a version of the resource bundle for each locale where your application will be used. You can copy the resource bundle file to another file and append the file name with an underscore and the language code, such as _es for Spanish. For example, if the resource bundle file name is ModelBundle.properties, you would name the Spanish resource bundle ModelBundle_es.properties. Then you would translate the values in the file—for example, LastName_LABEL=Apellido.

The slide shows examples of resource bundles in the native language (English) and how the bundles might look after localization to the locale es_ES, which is the locale for the Spanish language spoken in Spain. To ensure that your translations encompass the culture of a region and not just the language, you would also want to provide translations for other regions where Spanish is spoken and where you do business. For example, you might want to provide resource bundles localized for es_AR, which is the locale for the Spanish language spoken in Argentina.

The example shows a properties files. For XLIFF bundles, you use the same process. For list bundles, you append the file name with an underscore and the language code, but you need to remember to refactor the class so that the class signature matches the file name.

Configuring the Application to Support Locales

Configure supported locales in `faces-config.xml`.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As a final step in localizing your application, you must configure the application to support specific locales:

1. In the Applications window, under the view controller project, expand the `WEB-INF` folder and double-click the `faces-config.xml` file to open it.
2. At the bottom of the `faces-config.xml` file, click the Overview tab, and then select the Application page.
3. In the Locale Config section, specify the default locale.
4. In the Supported Locale list, click the Add (green plus sign) icon to add an entry for each locale that you plan to support. Make sure that you include the default locale in this list.

Other Approaches

Other approaches to internationalizing an ADF application include:

- XLIFF files (.xlf)
- List resource bundles (.java)
- Database as the resource store



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Earlier you learned that JDeveloper supports three different formats for resource bundles: properties bundles (.properties), XLIFF bundles (.xlf), and list bundles (.java). You learned the basics of internationalizing your application by using properties bundles. Properties bundles are the most straightforward option because their simple format makes it easier for non-programmers (such as translators) to read and update the files. However, before deciding which format to use, you should be aware of some of the advantages of other approaches.

XLIFF files (.xlf extension) are similar to properties files, but they enable developers to maintain a properly formatted XML structure for specifying string resources. Although XLIFF is more verbose than the other two options, it is supported by some specialized tools that are used by translators and might be the right choice if your translators use those tools.

List bundles (.java files) are most commonly used when substitution parameters for resource string values do not provide enough flexibility and coding is required to determine resource values. List bundles are the only option available for managing non-string objects or for implementing bundle-related business logic. Using Java classes, however, adds complexity to the translation cycle because the updated resources need to be recompiled and redeployed. In addition, Java source files are not as human-readable as properties files and therefore might not be as easily translated by nontechnical translation specialists.

Another approach that is not explored in detail in this course is the option of storing resource strings and translations in the database. Applications with a database-centric architecture might be best suited for this approach. This approach is more difficult to implement because it requires you to write custom code. However, managing database-backed resource bundles can be a lot easier than managing text resources stored in files. For example, with resource bundles stored in a database, you can make the text resources updateable through a simple webpage that is used to update the text. You can apply changes without having to go through full redeployment of the application.

Summary

In this lesson, you should have learned how to:

- Describe the validation options in ADF BC applications and when to use them
- Decide when to use declarative validation and when to use “method” declarative validation
- Add validation to the user interface
- Write Groovy expressions in validation
- Internationalize messages and other translatable text by creating a resource bundle



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 7 Overview: Validating User Input

This practice covers the following topics:

- Using declarative validation
 - Comparison validator
 - Range validator
- Using the Oracle ADF Model Tester to test the validation
- Specifying a Groovy expression in the rule definition of a validation rule
- Using resource bundles to specify messages for failure handling



Copyright © 2014, 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 NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

8

Modifying Data Bindings Between the UI and the Data Model

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

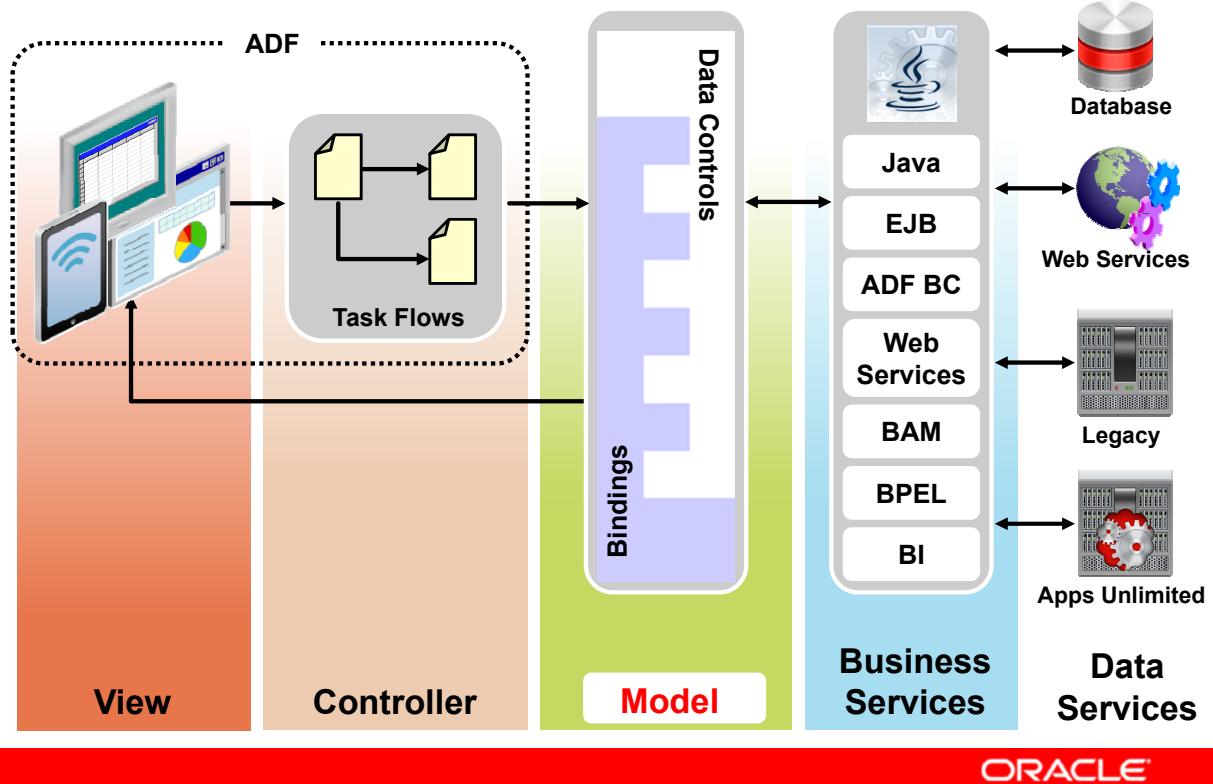
After completing this lesson, you should be able to:

- Describe the relationships among UI components, data bindings, data controls, and business services
- List and define the three major categories of data bindings
- Create and edit data bindings
- Use the Expression Builder to declaratively create EL expressions



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Modifying Data Bindings in the Model Layer

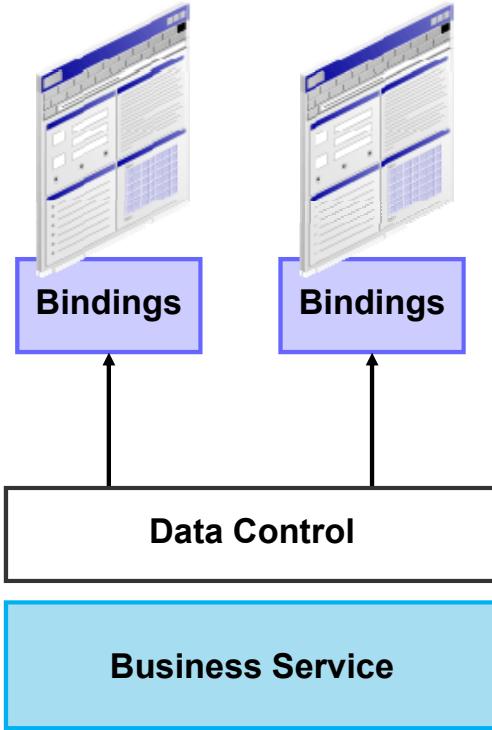


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In an earlier lesson, you learned about the Oracle ADF Model layer. In this lesson, you learn about the ADF Model layer in more detail. You learn how to create data controls and how to add and modify bindings.

Oracle ADF Model Layer: Review

- Data controls expose the public interface of a business service.
- Bindings are used to connect UI components to data controls.
- Data controls and bindings are defined by using XML metadata.
- The Data Controls window enables creation of UI components that are automatically data-bound.



ORACLE

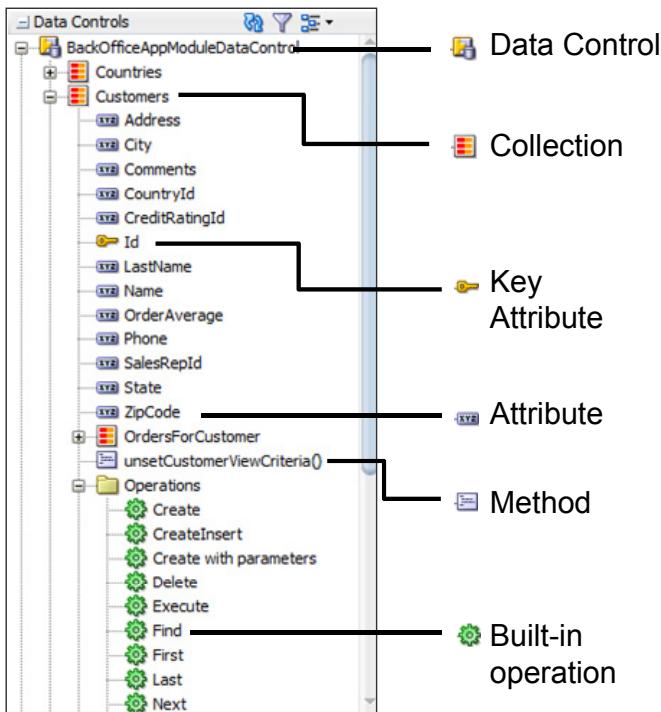
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You learned previously that ADF data controls expose the public interface of a business service—specifically, the business service's data collections, attributes, and methods. Data controls provide a consistent interface regardless of the underlying business service implementation. This means that UI developers are able to work in a way that is business service agnostic.

You learned that you bind UI components to data controls when you want to populate a page with data from your data model at run time. You also learned that the Data Controls window in JDeveloper provides a convenient mechanism for automatically creating data-bound UI components on a page. When you use this mechanism to create a UI component, JDeveloper automatically creates the code and objects needed to bind the component to the data control that you selected.

You now look more closely at ADF data controls and data bindings.

ADF Data Controls



Used to create...

(Not used to create anything; is a container for other objects)

Forms, tables, graphs, trees, range navigation components, and master-detail components

Label, text field, date, list of values, and selection list components

Same as key attribute

Command components and parameterized forms

Command components

ORACLE

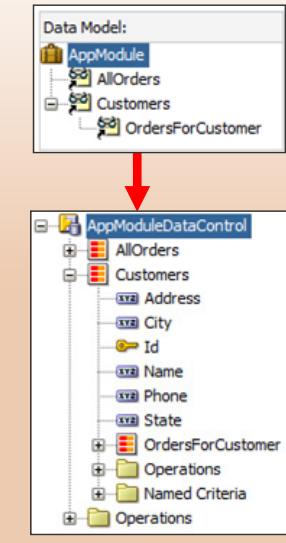
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Data Controls window shows the data controls that have been created for the application's business services. The data controls expose all the data objects, data collections, methods, and operations that are available for binding. A different icon is used for each type of data control object. Each root node in the Data Controls window represents a specific data control. Under each data control is a hierarchical list of objects, collections, methods, and operations. How this hierarchy appears in the Data Controls window depends on the type of business service represented by the data control and the way in which it was defined.

Creating ADF Data Controls

ADF Business Components

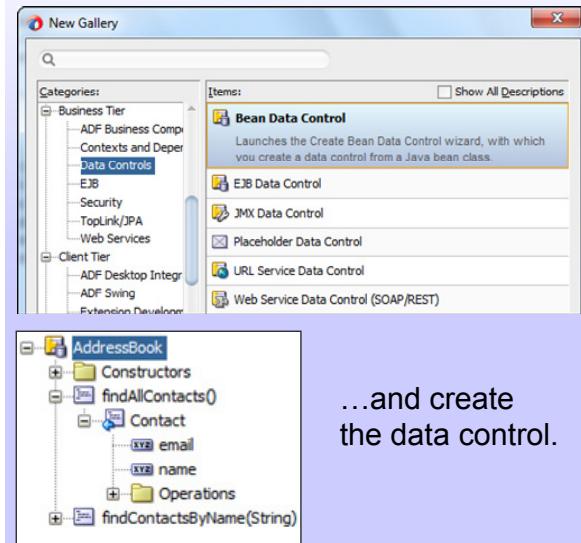
You create an application module that defines the data model...



...and the data control is available automatically.

Other Business Services

You select File > New > From Gallery > Business Tier > Data Controls...



...and create the data control.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you create an application module, it is “data control ready,” which means that a data control is available automatically for objects that you have added to the application module’s data model.

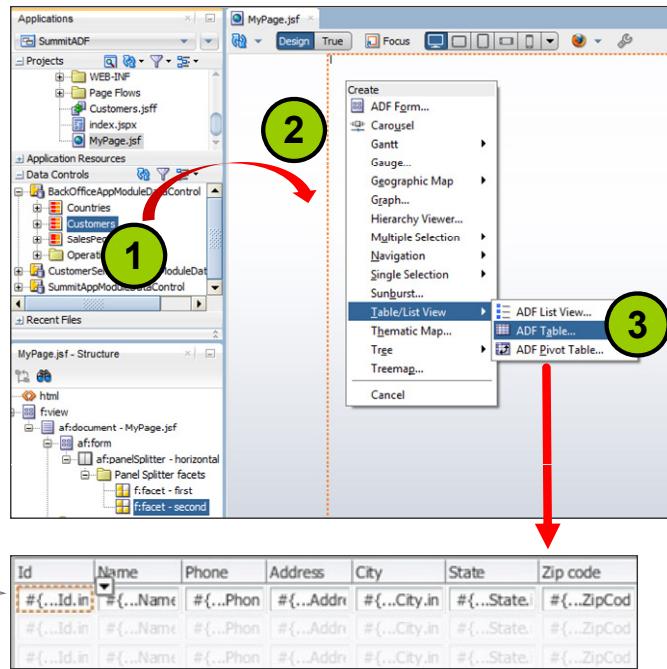
For other kinds of business services, though, you need to create the data control explicitly. For example, you can create data controls for web services, EJB session beans, or any Java class that is being used as an interface to some functionality. After you create the data control, you can drag it to a page to create a data-bound UI component in the same way that you would for ADF BC.

To create a data control, do one of the following:

- In the Applications window, right-click the item for which you want to create a data control, and select Create Data Control from the context menu.
- From the File menu, select New > From Gallery. In the New Gallery, expand Business Tier and select Data Controls. Select the kind of data control that you want to create, and then continue to define the data control in the wizard.

Using ADF Data Controls

1. In the Data Controls panel, select a data element.
2. Drag it to a page in the page editor or Structure window.
3. From the context menu, select the kind of component you want to create.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

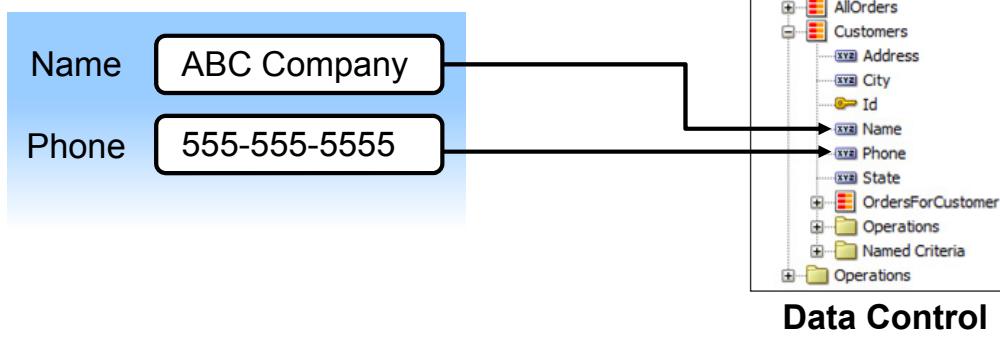
As you learned earlier, the Data Controls panel provides a convenient mechanism for automatically creating data-bound components on a page. When you drag a data element to a page, you are given a choice of the type of component to use to contain the data element, based on whatever is appropriate for that particular element. For example, dragging a collection 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.

After you select a component from the pop-up list, JDeveloper automatically creates the JSF page component code and the data-binding expressions needed to access the data control object. JDeveloper also defines the properties of the new UI component by using EL to refer to the bindings.

Tip: Whenever you update a business component, remember to click the Refresh button in the Data Controls panel to see your changes.

ADF Bindings

- Connect UI components on a page to the data and actions that are exposed through the data control
- Are created automatically when you drag elements from the Data Controls window to a page or panel
- Are described in a page-specific XML file called the page definition file, which is used at run time to instantiate a page's bindings



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The ADF binding layer is the glue that connects the UI components on the page to the data and actions exposed through the data control. The binding layer consists of metadata (generated for each page) and a set of generic framework classes. This extra layer of abstraction further decouples the web application from the technology that is used to implement the business service.

ADF bindings are created automatically when you add a UI component to a page by dragging elements from the Data Controls window to a page or panel. You can also create and edit bindings by using editors available in JDeveloper (you learn how to do this later in the lesson).

All the bindings for the UI components on a page are described in a page-specific XML file called the *page definition file*. The ADF Model layer uses this file at run time to instantiate the page's bindings. These bindings are held for the duration of the request in a map called the *binding container*.

Expression Language (EL) and Bindings

- Data-binding expressions are written in EL.
- They are evaluated at run time to determine the data to display.
- ADF EL expressions typically have the following form:
`# {bindings.BindingObject.propertyName}`

The diagram illustrates the relationship between a rendered UI component and its source code. At the top, a screenshot of a web browser shows a text input field with the placeholder "* Name ABC Company". Below this, the text "Component Rendered in HTML" is displayed. At the bottom, a block of XML source code is shown, enclosed in a red border. The source code defines an `<af:inputText>` component with attributes: `value="#{bindings.Name.inputValue}"`, `label="#{bindings.Name.hints.label}"`, `required="#{bindings.Name.hints.mandatory}"`, and `id="it1">`. To the right of the source code, a callout points to the redlined EL expressions in the attributes, with the text "Data-binding expressions are evaluated at run time.".

* Name ABC Company

Component Rendered in HTML

```
<af:inputText value="#{bindings.Name.inputValue}"
    label="#{bindings.Name.hints.label}"
    required="#{bindings.Name.hints.mandatory}"
    id="it1">
</af:inputText>
```

Data-binding expressions are evaluated at run time.

Source Code for Component

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

UI components use ADF data-binding expressions to access data bindings. The data-binding expressions are written in the Expression Language (EL) and evaluated at run time to determine what to display.

When you use the Data Controls panel to add a component, the data-binding expressions are created for you. The expressions are added to every component attribute that displays data from—or references the properties of—a binding object. You can add data-binding expressions to any component attribute that you want to populate with data from a binding object.

As you learned earlier, the EL is a scripting language used in JSF pages that enables the view layer (webpages) to communicate with application logic through simple syntax. A typical ADF data-binding EL expression uses the following syntax:

`# {bindings.BindingObject.propertyName}`

In this syntax:

- The `bindings` variable indicates that the binding object is located in the binding container of the current page
- `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. An EL expression can reference any binding object in the page definition file, including parameters, executables, and value bindings. You learn more about the page definition file in the next slide.

- The *propertyName* variable is the name of the ADF binding property that you are accessing to return a value from the ADF binding object. For example, the `inputValue` property returns the value of the first attribute to which the binding is associated. There are different binding properties for each type of binding object. For a description of all possible ADF binding properties, see <http://docs.oracle.com/middleware/1212/adf/ADFFD/appendixb.htm#BABDABIF>.

The example in the slide shows an ADF Faces input text component (a text field with a label) that displays the selected customer's name. The `value`, `label`, and `required` attributes of the text field are all set using EL expressions. The dot-separated expressions in the example all start with `bindings`, which is the binding variable that represents the binding context of the current page.

The EL expressions are evaluated at run time to return a value from the binding object:

- `value="#{bindings.Name.inputValue}"`: Sets the component's value by returning the `inputValue` property from the binding object
- `label="#{bindings.Name.hints.label}"`: Sets the component's label by returning the `label` property from the binding object.
Note: `hints` refers to the UI hints that are specified for the business component. (You learn about UI hints in a later lesson.)
- `required="#{bindings.Name.hints.mandatory}"`: `Required` is a boolean attribute that determines whether a value must be entered in the field. In this case, the `required` attribute evaluates to the `mandatory` property returned by the binding object. When the value of the `required` attribute is `mandatory`, the field is displayed in the UI with a required indicator (*) next to it, and the user must enter a value in the field.

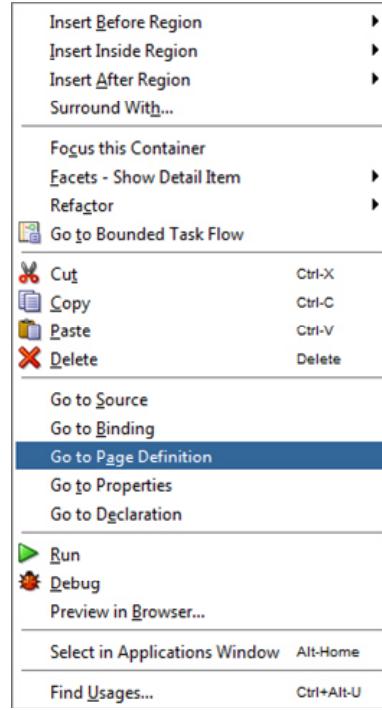
Viewing Data Bindings in the Page Definition File

The page definition file:

- Is created automatically when you add a data-bound 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 Applications window.
2. Select “Go to Page Definition.”



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

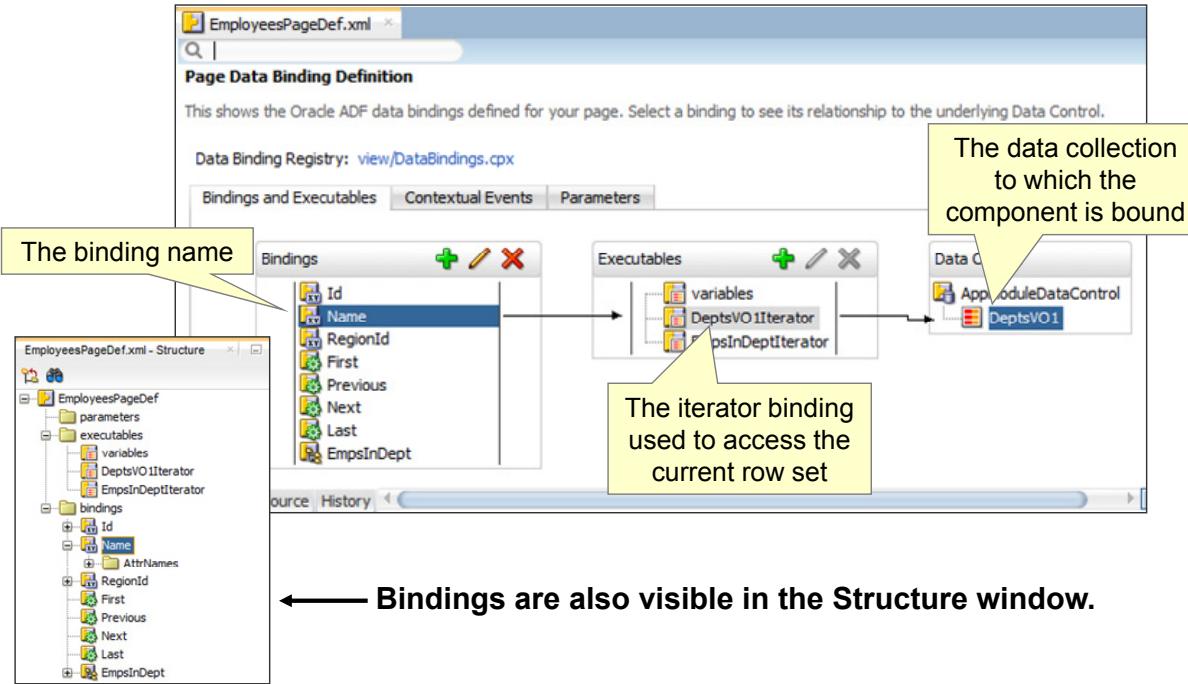
Information about data bindings is held not on the page itself but in a separate metadata file: <pagename>PageDef.xml. This file is created automatically when you first add a data binding to a page. Each time you add data-bound components to the page, JDeveloper adds appropriate declarative binding entries to the 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*. The container is accessible during each page request by using the EL expression # {bindings}, which 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 data-bound page.

To edit a page definition file, open it in the visual editor or code editor by right-clicking anywhere on a page (or page fragment) and selecting “Go to Page Definition” from the context menu. You can also view the bindings by clicking the Bindings tab while editing a JSF page.

In the Applications window, the page definition file is located in the Application Sources folder as <pagename>PageDef . xml.

Examining the Page Definition File



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

On the Overview tab of the page definition file, you can view and edit all the bindings that are defined for the page. In the Bindings list, select a binding to see the mapping between the binding, the executable that is used to access data values, and the data control. The executable in the middle column acts as the current record indicator to ensure that the binding points to the correct record or row set in the collection. The executable is usually an iterator that iterates over a collection.

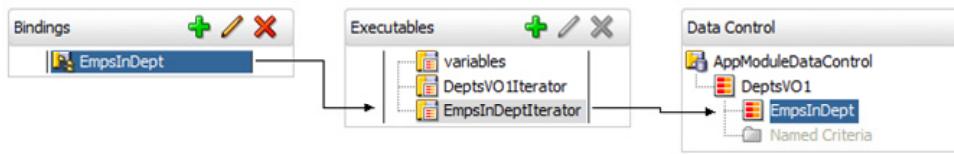
The binding in the example enables the application to access the current row of the DeptsVO1 data model object and display the current value of the Name attribute. Under Executables, notice that there is an iterator for each data collection.

To explore the relationship among bindings, executables, and data controls, you can also click the Source tab in the editor to see the XML source for the page definition file.

In the overview editor for the page definition file, you can add, edit, and delete data bindings. You can also see the structure of the bindings file in the Structure window, and you can click the Source tab to see the XML source for the file.

Categories of Data Bindings

- **Value bindings:** Connect UI components to attributes in a data collection—for example, attribute binding, tree binding, list binding, and table binding
- **Action bindings:** Invoke a method or operation
- **Iterator bindings:** Keep track of the current row in a data collection
 - Iterator
 - Method iterator
 - Accessor iterator
 - Variable iterator



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle ADF provides several categories of binding objects to support the attributes and operations exposed by the Oracle ADF data controls for a particular business object.

Value Bindings

Value bindings provide access to data. There is one value binding for each data-bound UI component on the page. The value binding that is used for a component depends on the UI component that is used to display the data, 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

Action bindings provide access to operations or methods defined by the business object. Action bindings are defined for command components such as buttons. 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.

Iterator Bindings

An iterator binding references an iterator for the data collection, which facilitates iterating over its data objects. It also manages currency and state for the data objects in the collection. An iterator binding does not actually access the data. Instead, it simply exposes the object that can access the data, and it specifies the current data object in the collection. Other bindings then refer to the iterator binding to return data for the current object or to perform an action on the object's data.

Note that the iterator binding is not an iterator. It is a binding to an iterator. Iterators are usually created for you so that you usually do not have to create them explicitly.

There are four types of iterator bindings:

- **Iterator:** Is used to iterate over a collection. When you drag a view object from the Data Controls window to a page, an iterator is created automatically.
- **Method iterator:** Is used to iterate over the results returned by a method
- **Accessor iterator:** In a master-detail relationship, is used to iterate over detail objects returned by accessors. An accessor iterator is created automatically when an accessor return is dragged to the page from the Data Controls window. 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 iterator.
- **Variable iterator:** Is used to iterate 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.

Editing Data Bindings

You can edit data bindings in the following places in JDeveloper:

- In the page definition file
- In the Properties window
- By rebinding an existing component to a different data control
- By binding an existing component to a data control



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

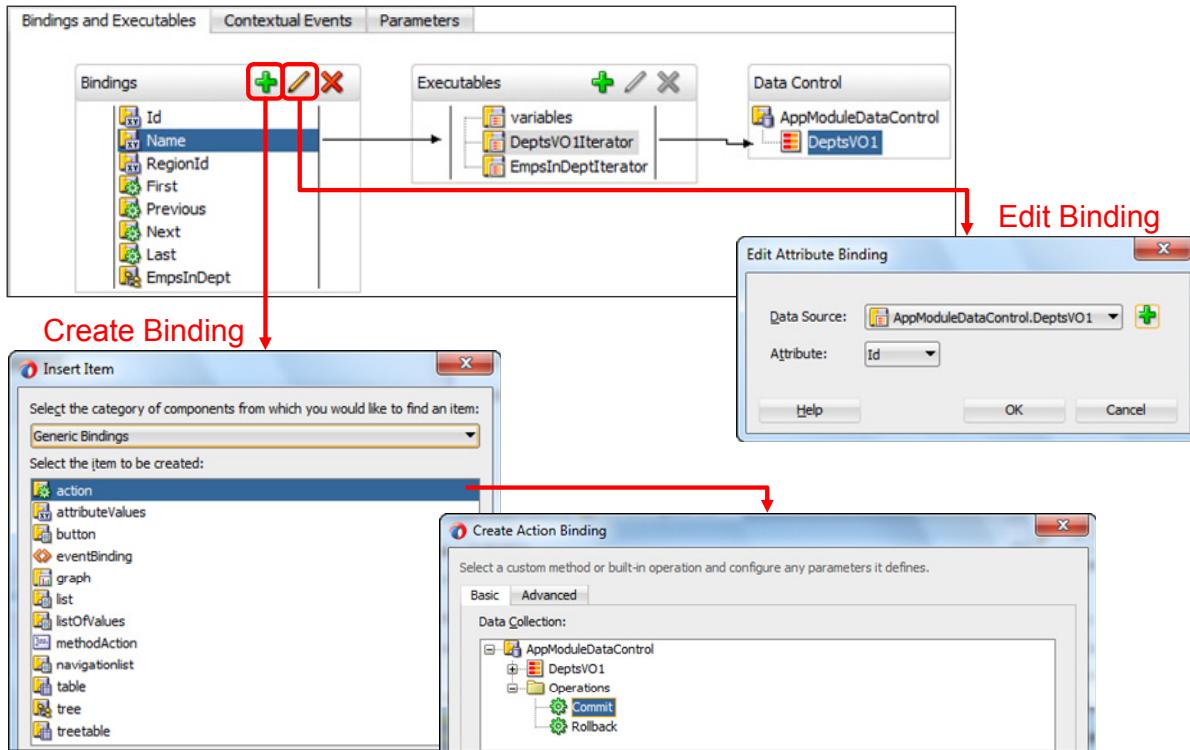
You learned previously that data bindings are created automatically when you drag a component from the Data Controls window to a page or panel. However, you might decide to build a page with unbound UI components (perhaps for UI prototyping) and then define the bindings later, or you might want to change the binding for an existing component.

You can also use JDeveloper to create new data bindings or edit existing bindings. You can:

- Create and edit bindings in the page definition file (on the Bindings tab of the editor, or by opening the page definition file)
- Select a component in the editor, and create or edit data-binding expressions in the Properties window. This enables you to use the Expression Builder to build an EL expression.
- Right-click a UI 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.”

The following slides show how to do this.

Editing in the Page Definition File



ORACLE

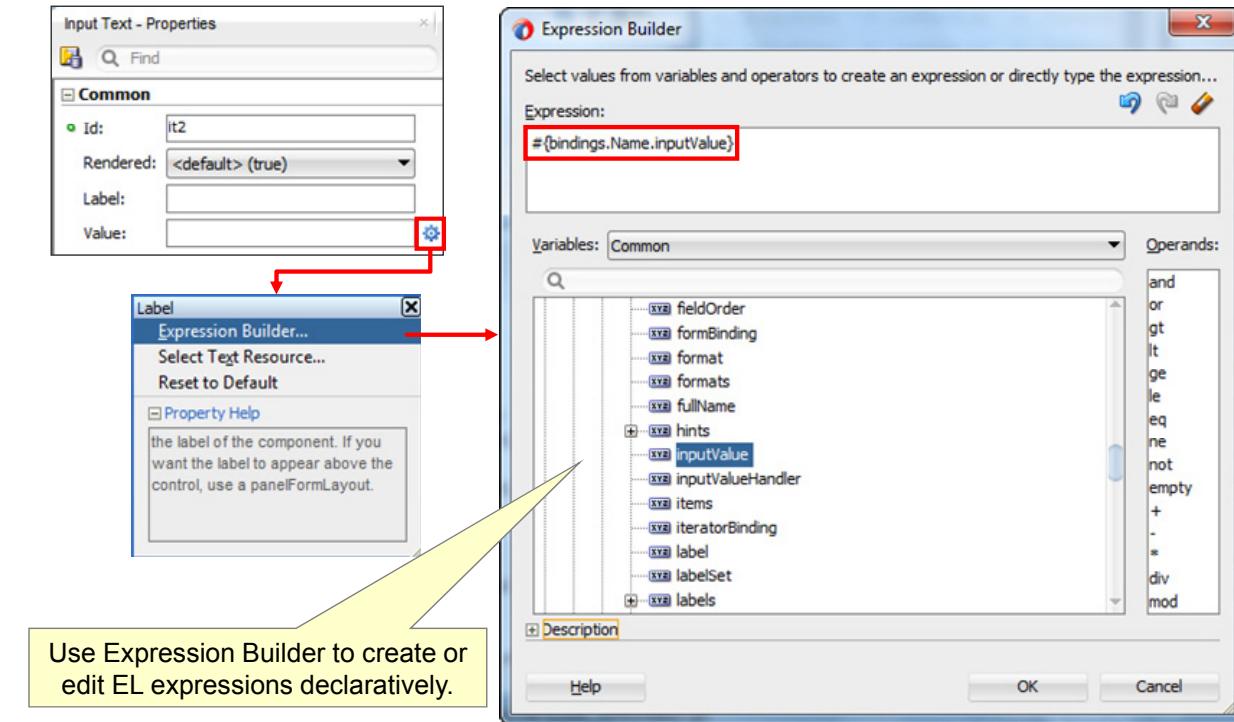
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the page definition file (on the “Bindings and Executables” tab), click the Create, Edit, or Delete icons in the Bindings area to modify the bindings for the page. To edit iterator bindings, click an icon in the Executables area.

- **To create a new binding:** Click the Create Control Binding icon or Create Executable Binding icon and select the kind of binding that you want to create. Specify the details of the binding. For help on specifying the details, click the Help button.
- **To edit an existing binding:** Select the binding and click the Edit Selected Element icon. Specify the changes that you want to make to the binding.
- **To delete a binding:** Select the binding and click the Delete Selected Elements icon. JDeveloper indicates any usages that exist for the binding. Click Show Usages to view the usages before confirming that you want to delete the binding.

Remember that you are editing the data bindings in the page definition file. You are not editing the data-binding expressions that are used by the UI components on the page.

Editing in the Properties Window



ORACLE

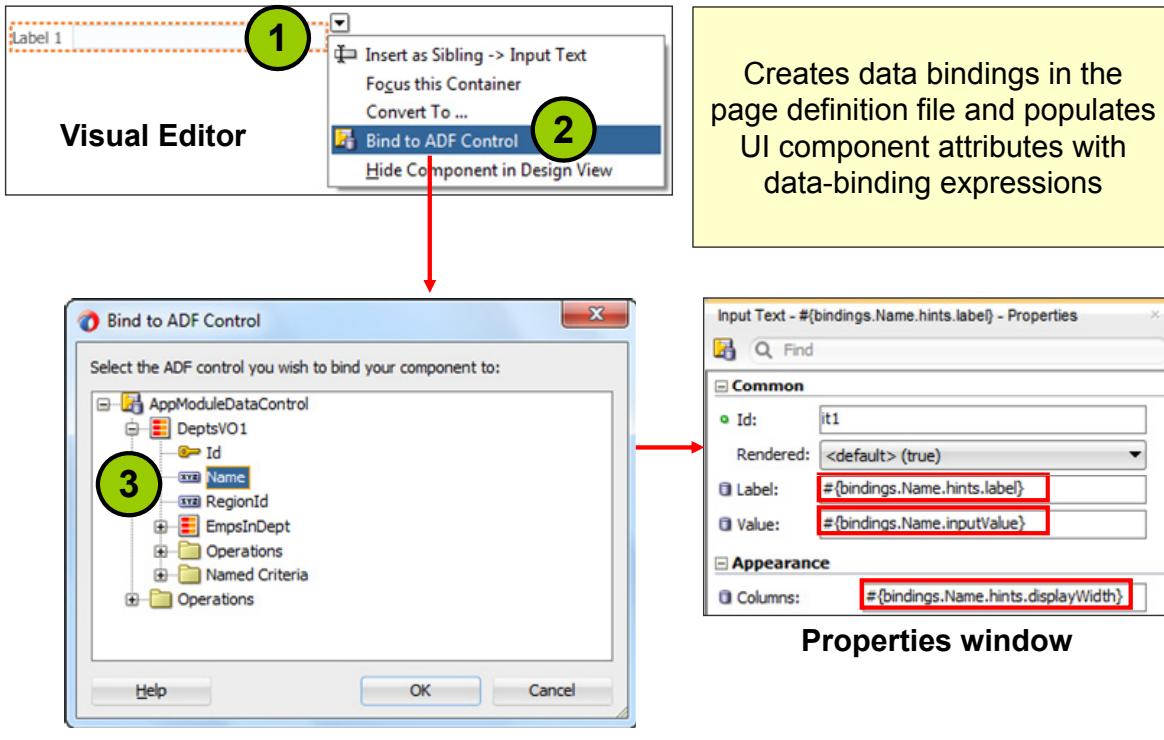
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the Properties window, you can create or edit the data-binding expression for a component by selecting the component in the page editor (or Structure window) and then clicking the Property Menu icon next to a property to run the Expression Builder. In the Expression Builder, you expand the ADF Bindings folder and select a binding to use for the component. Notice that the data binding must already exist in the ADF bindings for the page. The Expression Builder creates the data-binding expression; it does not create the data binding.

The example in the slide shows how to specify a data-binding expression for the `value` attribute of an input text component. The value of some UI components (such as an input text component) is determined at run time by the `value` attribute. Although a component can have static text as its value, the `value` attribute usually contains an EL expression that the run-time infrastructure evaluates to determine what data to display. Because any attribute of a UI 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.

Note: For simple expressions, rather than using the Expression Builder, you can begin typing the expression into a field in the Properties window, and then select from a list of options to auto-complete the expression.

Editing by Binding an Existing Component to a Data Control



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

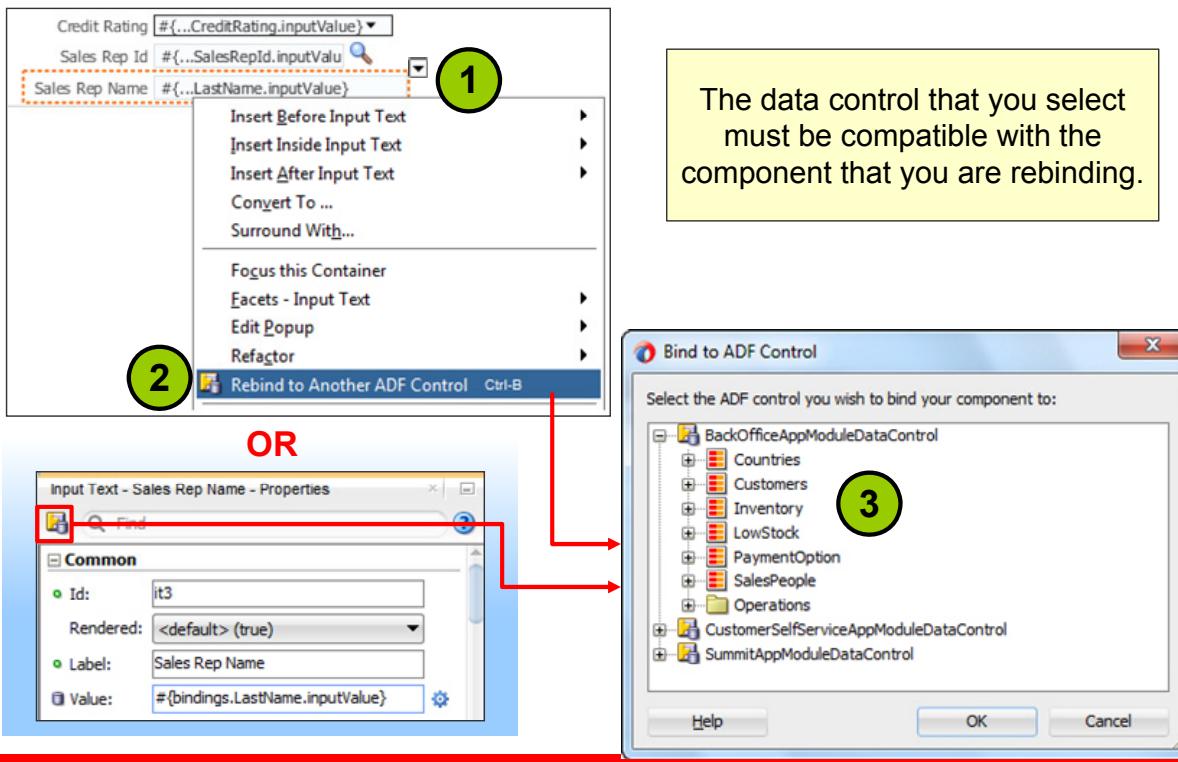
In some situations, you might want to create an unbound component and then bind it to an underlying data control later. For example, you might want to create unbound components when you are doing UI prototyping. Later, when you are ready to hook the controls up to data and executables, you can bind the components to data controls.

When you bind a component to a data control, JDeveloper creates the data bindings in the page definition file, and it populates the UI component attributes (such as Label, Value, and so on) with data-binding expressions.

To bind an unbound component to a data control:

1. In the visual editor or the Structure window, right-click the component (or click the down arrow next to the component in the visual editor).
2. Select “Bind to an ADF Control.”
3. Choose an ADF control to which to bind. The data control that you select must be compatible with the component that you are binding. For example, you must bind a text field to a control that returns a text value; you cannot bind it to a data control that invokes an operation.

Editing by Rebinding an Existing Component to a Different Data Control



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

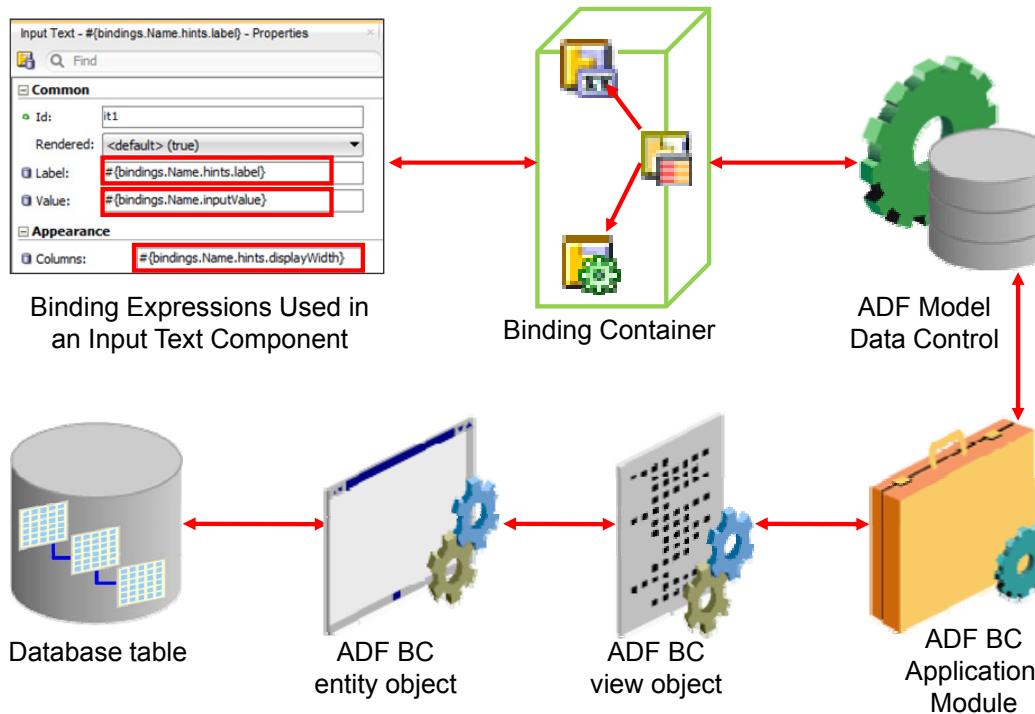
ORACLE

You can rebind a component to a different data control, and JDeveloper will update the data bindings in the page definition file and populate the UI component attributes (such as Label, Value, and so forth) with new data-binding expressions.

To rebind a component to a different data control:

1. In the visual editor or the Structure window, right-click the component (or click the down arrow next to the component in the visual editor).
2. Select “Rebind to Another ADF Control.” (Alternatively, you can select the component and click the Rebind button in the Properties window.)
3. Select a different ADF control to which to bind. The data control that you select must be compatible with the component that you are binding. For example, you must bind a text field to a control that returns a text value; you cannot bind it to a data control that invokes an operation.

How Bindings Work Behind the Scenes



ORACLE

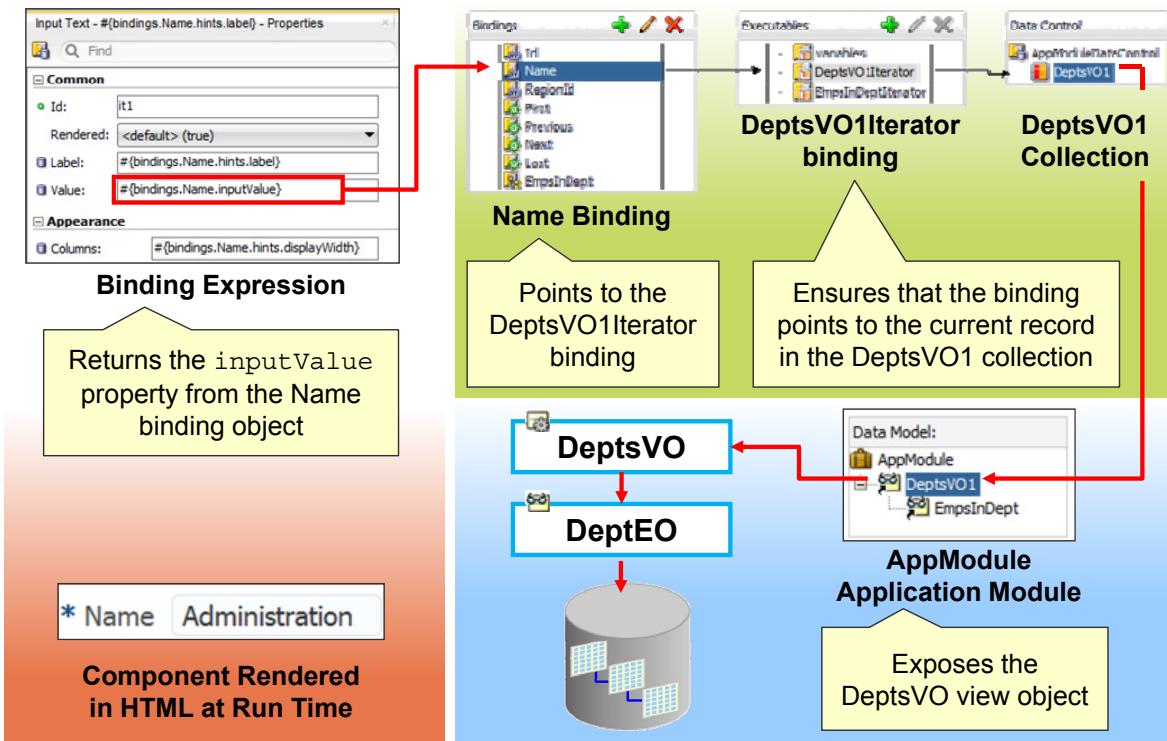
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As a developer, it is easy for you to create data-bound components. However, it is helpful for you to understand what goes on behind the scenes to enable this simplicity.

The 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 entity object data that is required by an application. The developer exposes the view object to an application in an ADF BC application module. When you create an application module, 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 entity object and, through that object, enabling access to the database table.

The next slide examines the concept of data binding in more detail, expanding on the top portion of the slide.

Example: Value Bindings for an Input Text Component



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in this slide traces a single value binding in a text component back through the model and services layers to the database where the data resides. To keep the example simple, other bindings used by the component are not examined. As you look through the slide, keep in mind that bindings are also used to connect other types of components (such as buttons) to executables that perform actions.

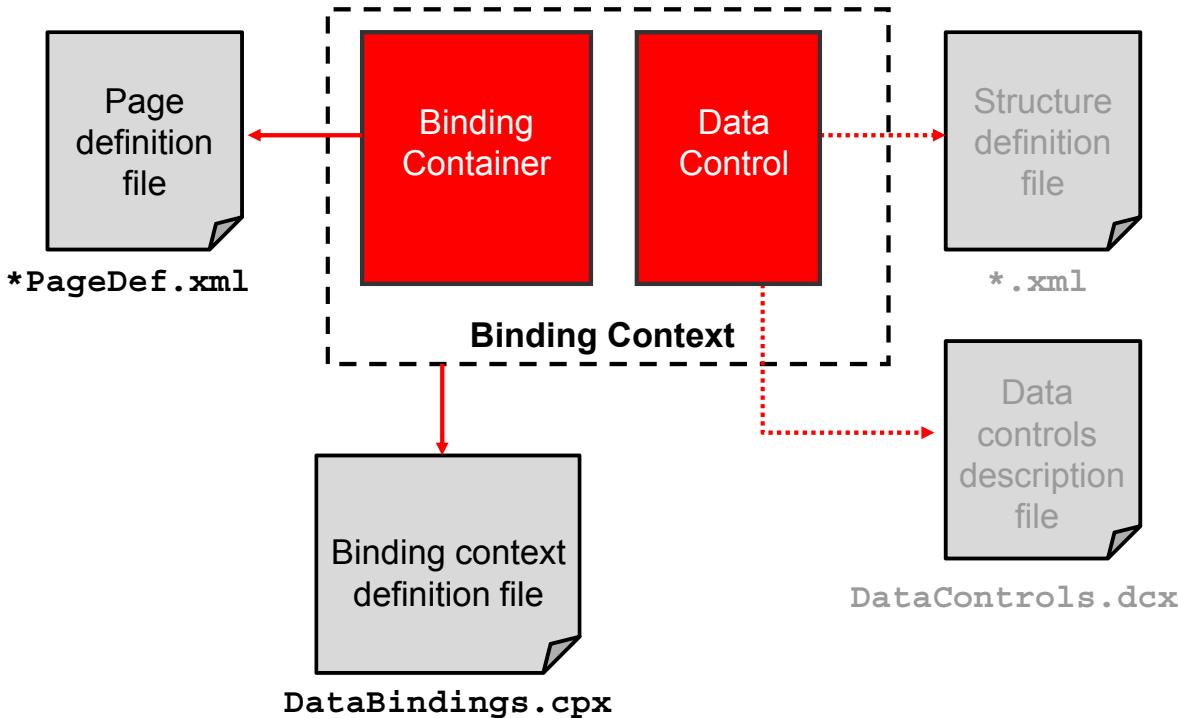
The example shows the following data-binding expression, which is used to return the value of the input text field:

```
{bindings.Name.inputValue}
```

This expression returns the `inputValue` property from the Name binding object. (The `inputValue` property resolves to the value of the first attribute to which the binding is associated.) Remember that each type of binding object has specific properties that expose data in the binding object. You can see the full list of binding object properties here: <http://docs.oracle.com/middleware/1212/adf/ADFFD/appendixb.htm>

The Name binding object points to an iterator binding called `DeptsVO1IteratorBinding`. The iterator binding is necessary to make sure the page displays the value from the current row. The iterator iterates over the rows exposed by the `DeptsVO1` collection in the data control. The data control exposes the data model defined in the application module, `AppModule`, which contains the `DeptsVO1` view object instance. `DeptsVO1` is an instance of `DeptsVO`, which presents an application-specific view of data from the `DeptEO` entity object. The entity object interacts with the database to retrieve data to display in the Name field.

Examining the Binding Context and Metadata Files



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You now take a closer look at the metadata files that are involved in data binding.

As you learned previously, the page definition file contains the binding definitions for a page. Typically there is one page definition file for each page. The page definition file is created automatically the first time you create a binding on a page. Page definition files provide design-time access to all the ADF bindings defined for a page. At run time, the binding objects defined by a page definition file are instantiated in a binding container, which is the run-time instance of the page definition file. The binding container together with the data controls give you the binding context.

Note: You work with the binding context later in the course when you learn how to programmatically access data bindings.

ADF Business Components is already data-control aware. Therefore, if you are using ADF BC, no extra files are created for data controls. However, if you create a data control from another type of business service, such as a POJO or web service, then potentially two files are created: a structure definition (.xml) file and a data controls description file named DataControls.dcx.

The first time you drag a data control to a page to create a data-bound component, another file (DataBindings.cpx) is created. This file contains the mapping between the pages that you create and the binding files (page definition files) that back those pages.

Summary

In this lesson, you should have learned how to:

- Describe the relationships among UI components, data bindings, data controls, and business services
- List and define the three major categories of data bindings
- Create and edit data bindings
- Use the Expression Builder to declaratively create EL expressions



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 8 Overview: Modifying Data Bindings

This practice covers the following topics:

- Creating and examining data bindings
- Examining metadata files
- Modifying bindings for an existing UI control
- Deleting a binding without deleting its reference on a page, and then observing the results



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Adding Functionality to Pages

9



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

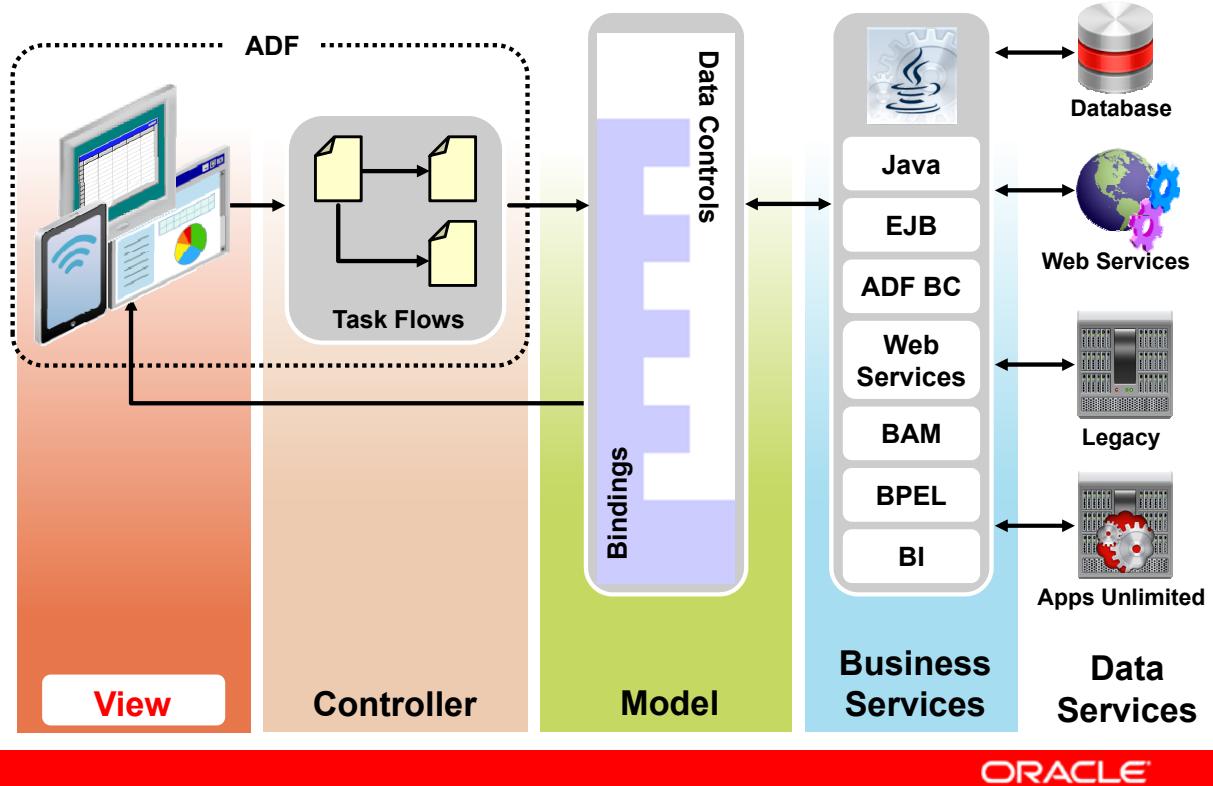
After completing this lesson, you should be able to:

- Display a selection list of values
- Display tabular data in tables
- Display hierarchical data in trees
- Define and use search forms and display the results
- Display data graphically
- Customize the UI programmatically by creating and configuring a backing bean



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Adding Functionality to Pages in the View Layer



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

In an earlier lesson, you learned how to use a few basic ADF Faces components on a page. In this lesson, you learn more about ADF Faces components. You learn how to use ADF Faces components to add functionality to the application, and you learn how to customize the UI programmatically by creating backing beans.

ADF Faces Rich Client Components

This lesson discusses the following types of components:

- General controls
- Text and selection
- Data views
- Menus and toolbars
- Layout
- Data visualization

ORACLE® ADF Faces Rich Client

View the ADF Faces Rich Client demo at:

<http://jdevadf.oracle.com/adf-richclient-demo/faces/index.jspx>

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are over a hundred ADF Faces Rich Client components from which to choose, many of which are discussed in this and later lessons. You can obtain a demo and tag reference for all the components at <http://jdevadf.oracle.com/adf-richclient-demo/faces/index.jspx>.

These components can be categorized as follows:

- **General controls:** For creating controls such as buttons, images, and links
- **Text and selection:** For creating controls that display, enter, or select values, such as text fields, radio buttons, check boxes, and drop-down lists
- **Data views:** For creating controls that display collections of complex data, such as tables, trees, query components, and carousels
- **Menus and toolbars:** For building menus and toolbars
- **Layout:** For arranging other components on a page using elements such as panels, decorative boxes, grids, tabs, accordions, and dashboards
- **Data visualization components:** For rendering dynamic charts, graphs, gauges, timelines, maps, and other graphics that provide a real-time view of the underlying data

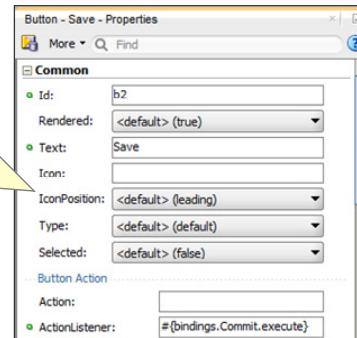
Defining General Controls



Use general controls for:

- Buttons
- Links [More...](#)
- Breadcrumbs
- Trains
- Images

Set properties to define the component's appearance and behavior.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You use general controls to display navigation controls, such as buttons and links, as well as images and icons. After adding a component to the page editor in JDeveloper, you specify properties that determine the appearance and behavior of the component. For example, when you define a button component, you must specify the text that displays on the button and an action that will be invoked when the button is clicked. The action might involve navigating to another page or calling an operation. In JDeveloper, you can access the tag reference for a component by pressing the F1 key from within the Components window.

You can quickly create controls that are bound to operations by selecting an operation from the Data Controls panel, dragging it to the page, and selecting the kind of component that you want to create. When you do this, JDeveloper automatically populates the ActionListener property with an action binding that invokes the operation.

You learn more about specifying navigation components later in this course.

Defining Text and Selection Components



Use text and selection components for:

- **Input text** * ID
- **Output text** This component supports styled text.
- **Radio buttons** Gender Male Female
- **Check boxes** Affiliated
- **Date selectors** Date
- **Number sliders**
- **Drop-down lists** Country 
- **Color selectors** 
- **Many others!**

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

You use text and selection components to create UI controls for displaying, entering, or selecting values. The slide shows only a subset of the text and selection components that are available. The input and selection components in this category accept user input in a variety of formats, including text, numbers, and dates. Values that are entered as input can be validated and converted before they are processed.

As with other ADF Faces components, you specify the behavior and appearance of these components declaratively by setting properties in the Properties window.

You can quickly create data-bound components by selecting an attribute or collection in the Data Controls panel, dragging it to the page, and selecting the kind of component that you want to create. When you do this, JDeveloper automatically generates the required bindings.

For example, if you drag an attribute to the page and choose to create an Input Text component, JDeveloper automatically populates the component properties, such as Label and Value, with data bindings that retrieve the label and value of the attribute in the current row of the collection.

If you drag a collection to the page and choose to create a Select One Choice list, JDeveloper prompts you to confirm the data source for the list and to select the attribute that will be displayed in the list.

You learn more about list bindings for selection components next.

Defining Lists for Selection Components

The list of values for a selection component can come from:

- A static list of values
- A dynamic list of values
- A model-driven list of values
- A managed bean



Use a model-driven LOV when you:

- ✓ Need to apply view criteria to an LOV
- ✓ Need to switch the LOV dynamically at run time
- ✓ Have data values (such as foreign keys) that are tightly integrated with the underlying model

ORACLE

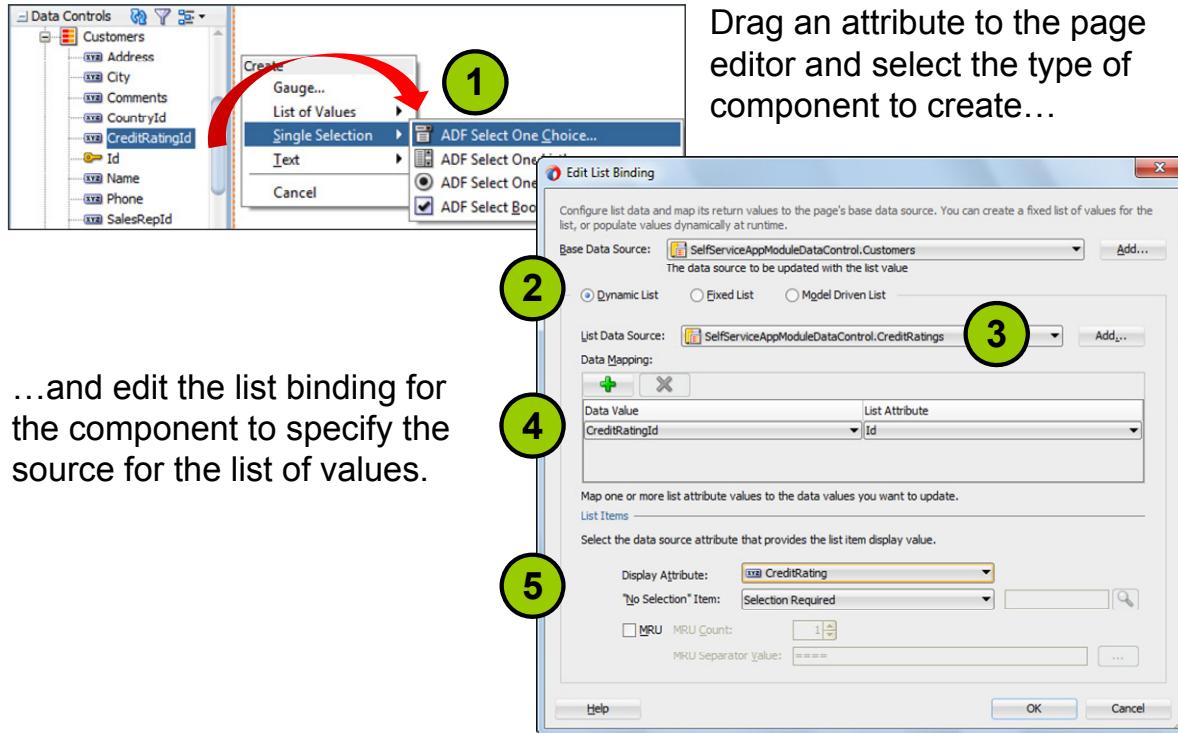
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you define lists of values for selection components, you can define:

- A static list of values that is hard-coded in the list binding of the selection component (see the next slide). Static lists that are defined in this way are not translatable.
- A dynamic list of values defined in the list binding for the selection component (see the next slide). This kind of list is most useful for master-detail lookup scenarios, where the value of the detail view's attribute exists as a corresponding value in the master view.
- A model-driven list of values on an attribute in a view object. You learned how to create a model-driven LOV in the lesson titled "Declaratively Customizing ADF Business Components." To use a model-driven LOV on a page, you simply drag the attribute that has the LOV associated with it to the page. By default, the page uses the UI control that you selected when you specified the UI hints for the LOV. You can override this setting and choose a different type of selection component. You should use a model-driven LOV when you want to:
 - Filter the list of values by applying view criteria to an LOV
 - Switch the LOV dynamically at run time
 - Display data values (such as foreign keys) that are tightly integrated with the underlying model

You can also retrieve a list of values from a managed bean.

Defining Lists for Selection Components



...and edit the list binding for the component to specify the source for the list of values.

Drag an attribute to the page editor and select the type of component to create...

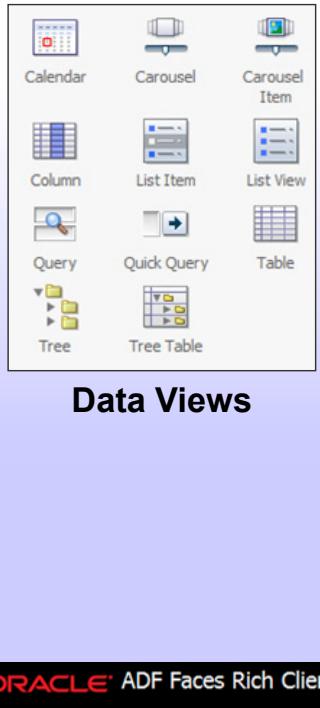
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

To create a selection component and bind it to a list of values:

1. Drag an attribute from the Data Controls window to the page editor, and select the type of component that you want to create. If you are using a model-driven LOV, select one of the List of Values components, and you are done. Otherwise, select a choice list or other type of selection component.
2. In the Edit List Binding dialog box, choose whether to create a dynamic list or a fixed list. If you are creating a fixed list of values, simply enter the list of items separated by line breaks, and you are done.
3. If you selected Dynamic List, click Add for the List Data Source and select the data collection that contains the attribute that will populate the values in the selection list.
4. In the Data Mapping section, make sure the Data Value column displays the attribute that will be updated when the user selects an item in the list. Also make sure that the List Attribute column contains the attribute that populates the values in the selection list.
5. For the Display Attribute, select the attribute that you want to display in the list. Choosing a display attribute such as CreditRating instead of CreditRatingId provides a more user-friendly label to display in the list of values. Instead of seeing a list of numbers, users will see a meaningful label.

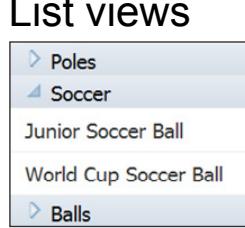
Defining Data Views



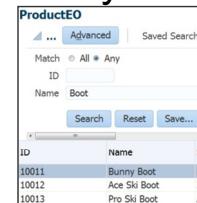
ORACLE ADF Faces Rich Client

Use data view components to define:

- Carousels
- Query forms
- List views
- Trees
- Tables
- Tree tables



ID	Name	Price
10011	Bunny Boot	150
10012	Ace Ski Boot	200
10013	Pro Ski Boot	410



Directory Name	Icon	Last Modified
My Files	/aff/folder_ena.png	
META-INF	/aff/folder_ena.png	06/18/2013 3:32 PM
WEB-INF	/aff/folder_ena.png	07/19/2013 11:32 AM
components	/aff/folder_ena.png	06/18/2013 3:32 PM
confusedCon	/aff/node_ena.png	06/18/2013 3:32 PM

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

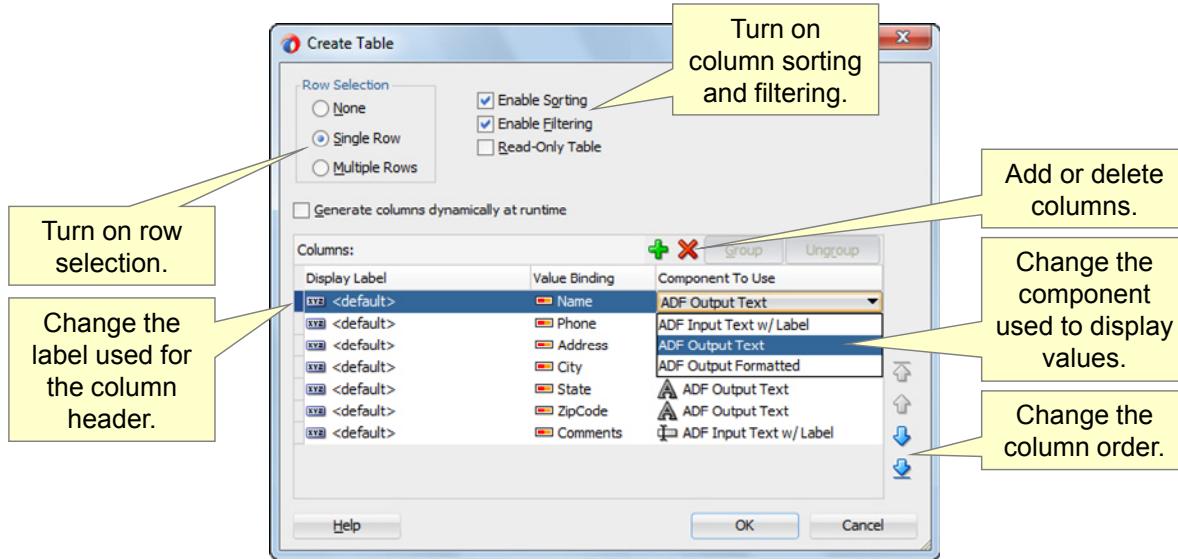
You use data view components to create controls that display collections of complex data, such as tables, trees, query components, and carousels.

You specify the behavior and appearance of data view components declaratively by using dialog boxes and the Properties window.

The easiest way to create a data view is by selecting a data collection in the Data Controls panel, dragging it to the page, and selecting the kind of component that you want to create. When you do this, JDeveloper usually displays a dialog box that enables you to specify details about the data view that you are building, such as the attributes to display and the settings that control the layout of the data view.

You learn how to define some of the more common data views (tables and trees) in the following slides.

Defining Tables



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can easily add a data-bound table to a page by using the ADF Faces Table component. The bindings are created for you automatically when you drag a collection from the Data Controls panel and choose to create it as an ADF table.

In the Create Table dialog box, you can specify basic properties of the table, such as:

- Whether the rows can be selected
- Ability to sort and filter columns
- Whether the values in the table are read-only or updateable

In the Create Table dialog box, you can add or delete columns, change the component that is used to display the values in a specific column, change the order of columns, and change the label that is used for the column header. You can also choose to group selected columns under an additional header.

Specifying Table Properties

Name	Address	City	State	Zip Code	Comments
Acme Sporting Go...	770 4th Ave	San Diego	CA	92101	
Athletes Anonymous	5801 College Ave	Oakland	CA	94618	Inconsistent payments
Baxter Bike	1001 Broxton Ave	Los Angeles	CA	90024	
Big John's Sports E...	4783 18th Street	San Francisco	CA	94117	Customer has a dependable credit record.
Bikes-n-More	735 S Figueroa St	Los Angeles	CA	90017	Places very large orders
Colma Bicycle	200 Colma Blvd	Colma	CA	94014	New Venture
Father Gym's	7007 Friars Rd	San Diego	CA	92108	
Futbol Mundial	11677 San Vicente...	Los Angeles	CA	90049	
Geary Blvd Store	2675 Geary Blvd	San Francisco	CA	94118	
Hooligans	700 Du Bois St	San Rafael	CA	94901	
Hot Stuff	25613 Dollar St	Hayward	CA	94544	
Perfect Purchase	631 San Felipe Road	Hollister	CA	95023-2803	
Ridearound	3700 Mandela Pkwy	Oakland	CA	94608	
Saucy Bikes	180 Donahue St	Sausalito	CA	94965	
Smooth Ride	1717 Harrison St	San Francisco	CA	94103	
Westwind Sports	531 Cowper St	Palo Alto	CA	94301	

Table properties:

```
columnStretching="last"
horizontalGridVisible="false"
verticalGridVisible="false"
rowBandingInterval="1"
```

Column properties:

```
sortable="true"
filterable="true"
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The ADF Faces Table component is similar to the standard JSF component, but provides additional features, including the ability to sort columns, filter data, reorder columns, and select a single row or multiple rows. After using the Create Table dialog box to specify the columns and general behavior of a table, you can further refine the table by setting properties.

The Properties window includes a full range of properties that control the look and feel, as well as the behavior, of the table.

Select the Table component (`af:table`) in either the Structure window or the page editor to modify property settings for the table itself, such as:

- The look and feel of the table
- Whether users will be able to select rows or reorder columns
- The bindings for the data that is to be displayed

Each Table component contains column components (`af:column`) that are nested below the Table component. Select a Column component and modify its properties to change the default look and feel, as well as the behavior, of the column. You can specify details such as:

- The alignment of values
- Whether values are allowed to wrap within cells
- Whether the column can be sorted

Use header and footer facets to specify column headers and footers.

To achieve the look and feel of the table shown in the slide, modify the following Table component properties from their default values:

- **ColumnStretching:** Set to `last` so that the last column in the table stretches to fill any remaining space.
- **HorizontalGridVisible:** Set to `false` to turn off the horizontal grid lines that appear by default in a table.
- **VerticalGridVisible:** Set to `false` to turn off the vertical grid lines that appear by default in a table.
- **RowBandingInterval:** Set to `1` to format every other cell in the table with a colored background.

At the column level, the following properties are set:

- **Sortable:** Set to `true` to enable users to sort the column by ascending and descending order.
- **Filterable:** Set to `true` to enable users to filter the rows displayed in the table by entering a string.

Examining Table Bindings

The screenshot illustrates the JDeveloper interface for examining table bindings. It shows the Bindings Tab, Structure Window, and the properties of a Table component.

- Bindings Tab:** Shows a tree structure of bindings. A binding for 'Customers' is selected, which points to an 'Executables' node containing 'variables', 'CustomersIterator', and 'CustomersQuery'. This node also points to a 'Data Control' panel.
- Data Control Panel:** Shows the 'SelfService AppModule Data Control' with nodes like CardTypes, Countries, CreditRatings, Customers, Images, PaymentTypes, ProductCategories, and Products. The 'Customers' node is highlighted.
- Structure Window:** Shows the structure of 'TestTables.jsf'. It includes an 'af:table' component with an ID of 't1'. The 'Value' property of 't1' is bound to '#{bindings.Customers.collectionModel}'. The 'Var' property is set to 'row'. The 'Output Text' component within the table has a 'Value' property of '#{row.Name}'.
- Table - t1 - Properties:** Shows the properties for the 't1' table component. The 'Value' is '#{bindings.Customers.collectionModel}', 'Var' is 'row', and 'VarStatus' is 'vs'.
- Output Text - #{row.Name} - Properties:** Shows the properties for the 'Output Text' component. The 'Id' is 'ot1', 'Rendered' is '<default> (true)', and 'Value' is '#{row.Name}'.
- Annotations:**
 - A callout points to the 'Value' property of the Table component with the text: "The binding expression returns data for the table."
 - A callout points to the 'Value' property of the Output Text component with the text: "The row variable holds the current row object."

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you drag a data collection from the Data Controls panel to the page editor, JDeveloper binds the table components to data by using an ADF tree binding. (If this seems counterintuitive, think of the table as simply a flattened hierarchy where the table node is the parent of all the column nodes nested below it.)

In the example, notice the binding expression for the Value property of the Table component:
`#{bindings.Customers.collectionModel}`

In this expression:

- `bindings` is a reference to the binding container for the current page
- `Customers` is the ID of the tree binding object in the page definition file. To facilitate iterating over the data objects in the collection, the `Customers` binding references the iterator binding for the `Customers` data collection
- `collectionModel` is a property of the binding object that returns the data for the table wrapped in an object

The Var property of the Table component specifies the name of the variable that holds the current row object during table rendering. By default, the value for this variable is defined as `row` and is accessible from expression language. Notice that the `row` variable is used in the Column child components (for example, the Output Text component) to reference the current row of data. The expression `#{row.Name}` returns the value of the `Name` attribute in the current row. During table rendering, this value is used to populate the `Name` column for each row.

How Table Data Is Rendered through Stamping

Each child component of a column is stamped once per row. As a result, there are no embedded stamped components.

```
<af:table var="row" value="#{bindings.Customers.collectionModel}">
  <af:column>
    <af:outputText value="#{row.FirstName}" />
  </af:column>
  <af:column>
    <af:outputText value="#{row.LastName}" />
  </af:column>
  <af:table var="row" value="#{bindings.Customers.collectionModel}">
    <af:column>
      <af:outputText value="#{row.FirstName}" />
    </af:column>
    <af:column>
      <af:outputText value="#{row.LastName}" />
    </af:column>
  </af:table>
</af:table>
```

Nested tables are not supported!



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The immediate children of an `af:table` component must all be `af:column` components. Each visible `af:column` component creates a separate column in the table.

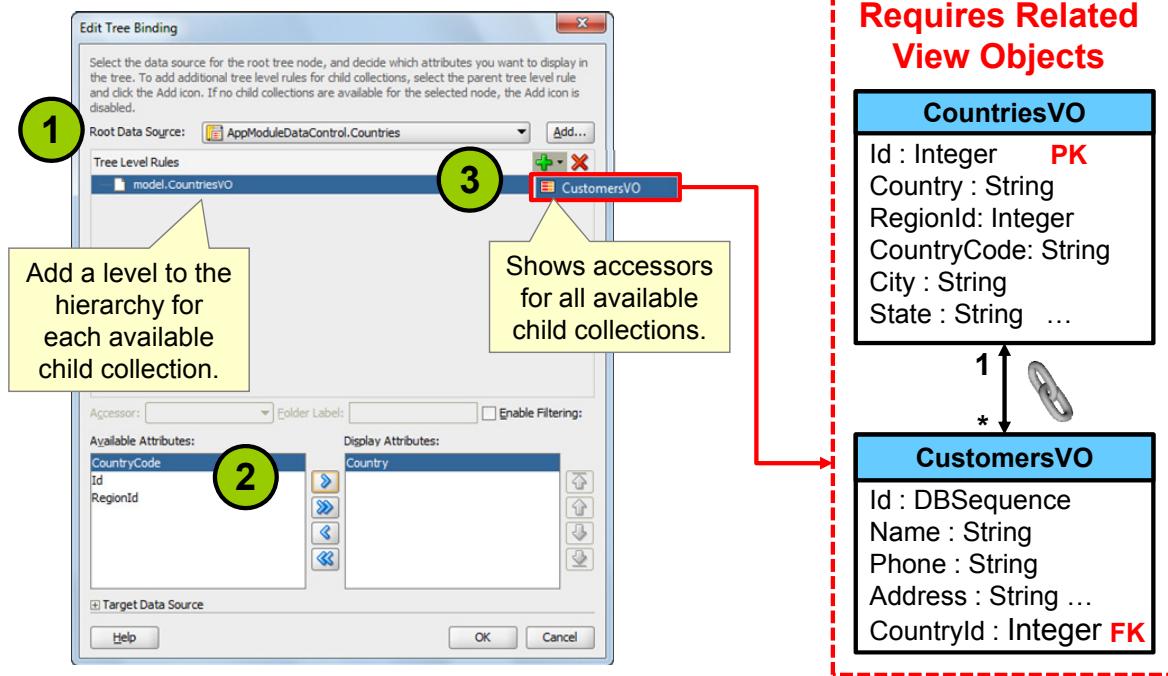
The child components of each column display the data for each row in that column. The column does not create child components per row; instead, each child is repeatedly rendered (stamped) once per row.

Because of this stamping behavior, no embedded stamped components are allowed. Any component that is pure output, with no behavior, will work without problems. For example, ADF Faces input components work properly. However, components that themselves support stamping (such as tables within tables) are not allowed. This restriction applies to any component that is rendered through stamping (such as Tree, TreeTable, and ListView).

As each row is stamped, the data for the current row is copied into a property that can be addressed using an EL expression. You specify the name to use for this property in the `var` property on the table. After the table has completed rendering, this property is removed or reverted to its previous value.

In the example in the slide, the data for each row is referenced by using the variable `row`, which identifies the data to be displayed in the table. Each column displays the data for each row as defined by a particular attribute.

Defining Trees



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The ADF Faces Tree component displays hierarchical data in which each tree node represents the data of a specific level in the hierarchy. The rendered hierarchy can have more than one root node, and each root node can have any number of child elements, which can appear under more than one parent node. The Tree component displays the data in a form that represents the structure, with each element indented to the appropriate level (to indicate its level in the hierarchy) and connected to its parent. Users can expand and collapse portions of the hierarchy. A common use case for the Tree component is a navigation tree.

To create a data-bound tree, you drag a collection from the Data Controls panel to the page editor and choose to create it as an ADF Tree. In the Edit Tree Binding dialog box, you configure the tree structure by defining rules for accessing the data for each level of the hierarchy:

1. In the Root Data Source list, verify that the data collection you want to use is selected. If it is not selected, click the Add button and select a different data collection. Notice that a tree level rule is created for the root node.
2. In the Available Attributes list, select the attributes that you want to display in the tree and shuttle them into the Display Attributes list. In the example, Country is selected because the tree will include a list of countries.

3. In the Tree Level Rules area, click the Add Rule button to add a rule for the child of the selected node. Choose the accessor for the child collection that you want to include in the hierarchy. In the example, you use the CustomersVO accessor (defined in CountriesVO) to access the child collection (Customers). Within the data model, there is a relationship defined via a view link between the Id attribute in CountriesVO and the CountryId attribute in CustomersVO. After you add the child collection to the example, the tree will include two levels: an expandable list of countries and, under each country, all customers who are located in that country.
4. Repeat steps 2 and 3 to add additional levels to the hierarchy. You can add a level to the hierarchy for each available child collection.

Synchronizing Another Part of the Page with the Tree Selection

In addition to synchronizing a detail view with a master view, you can synchronize other parts of a page with a tree selection. For each node definition (rule), you can specify an optional TargetIterator property and set it to an EL expression for an iterator binding in the current binding container. The expression is evaluated at run time when the user selects a row in the tree.

You specify the EL expression for the iterator binding in the Target Data Source section of the Edit Tree Binding dialog box. For more information, see the section about displaying master-detail data in *Developing Fusion Web Applications with Oracle Application Development Framework*.

Examining Tree Bindings

The screenshot illustrates the JDeveloper interface for examining tree bindings. At the top is the Bindings Tab, which shows a binding from a 'Countries' collection to a 'variables' node, which in turn points to an 'AppModuleDataControl' named 'Countries'. Below the Bindings Tab is the Structure Window, which displays the tree component's structure: it contains an 'af:tree' component ('t1') with facets for contextMenu, nodeStamp, outputText, and pathStamp. The 'Value' property of the tree component is set to '#{bindings.Countries.treeModel}'. The 'Var' property is set to 'node'. A red dashed arrow points from the 'node' variable in the tree component's properties to the 'Value' property of the 'af:outputText' component in the Structure Window. The 'af:outputText' component has its 'Value' property set to '#{node}'. A callout box states: 'The tree uses default nodeStamp rendering to display attributes as node labels.' To the right of the Structure Window is a tree view showing a hierarchy of countries: Brazil, Canada, Columbia, Czech Republic, Dominican Republic, Egypt, France (which is expanded to show Sportique, Germany, Hong Kong, India, Japan, Nigeria, Russian Federation, and USA).

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you drag a data collection from the Data Controls panel to the page editor and create a tree component, JDeveloper binds the tree component to data by using an ADF tree binding. The bindings that are created for a tree are similar to the bindings for a table.

In the example, notice the binding expression for the Value property of the Tree component:

```
# {bindings.Countries.collectionModel}
```

In this expression:

- `bindings` is a reference to the binding container for the current page
- `Countries` is the ID of the tree binding object in the page definition file. To facilitate iterating over the data objects in the collection, the `Countries` binding references the iterator binding for the `Countries` data collection
- `collectionModel` is a property of the binding object that returns the data for the tree wrapped in an object

The Var property of the Tree component specifies the name of the variable (`node`) that holds the current row object during tree rendering. During tree rendering, this variable is used to populate the tree and leaf nodes. Like tables, trees use a single instance of the UI components that are contained in the nodeStamp facet to render the nodes through stamping. Because of this stamping behavior, only certain types of components are supported as children within an ADF Faces tree. All components that have no behavior are supported, as are most components that implement the `ValueHolder` or `ActionSource` interfaces.

Keep in mind that the `node` variable is a reference to an object. If a tree binding specifies more than one display attribute for a specific level of the tree, the expression `#{{node}}` returns values for all display attributes. For example, imagine that you have a tree binding that marks two display attributes, `FirstName` and `LastName`, for a specific level of the tree. If you specify the value `#{{node}}`, the tree would display the first name and last name (for example, John Smith).

You can use an expression that refers to specific display attributes. For example, you could specify the following expression for the value of an output text component in a `nodeStamp` facet:

```
#{{node.LastName}}, #{{node.FirstName}}
```

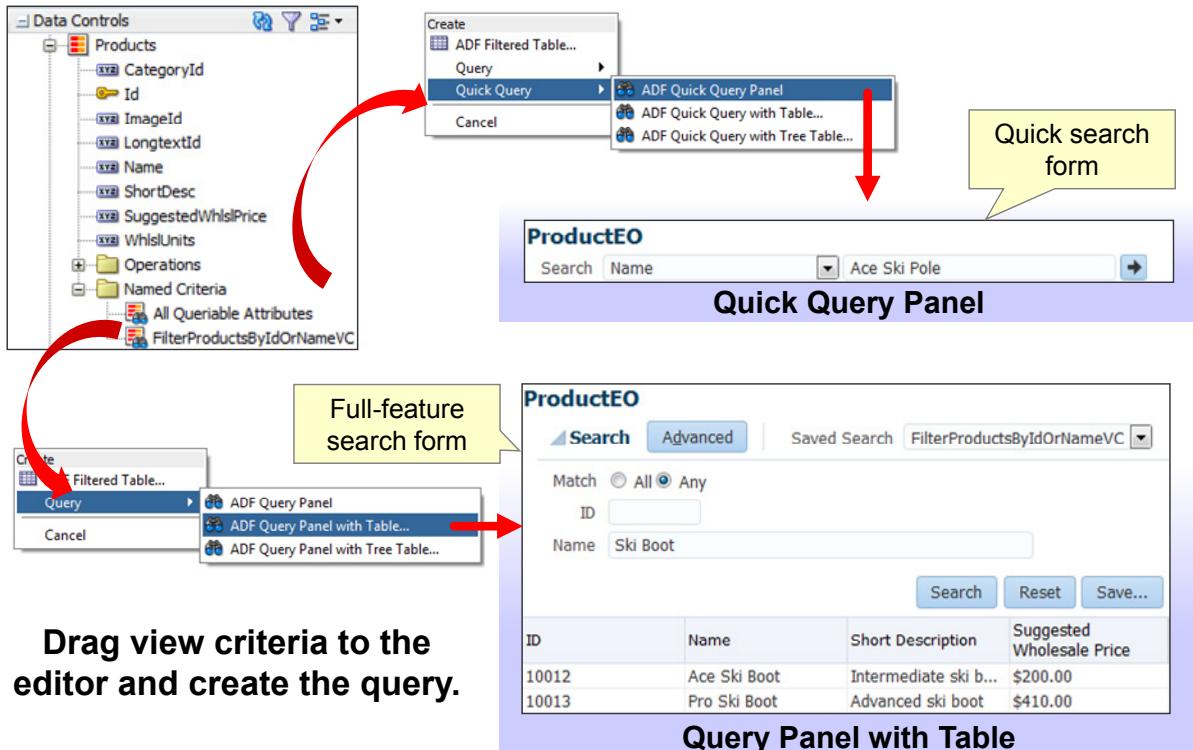
This expression would display something like Smith, John.

If you use this approach, remember that node stamping is used to render each level of the tree. You need to verify that the expression returns a value for every level in the tree. If the expression refers to an attribute that does not exist at a given level, the expression returns null. For example, if you want to define a tree that contains department names at the top level of the tree with the employees in each department nested below, you could use the following expression for the value of an output text component in a `nodeStamp` facet:

```
#{{node.departmentName}} #{{node.lastName}}.
```

For department nodes, the `lastName` attribute would evaluate to `null`, so only the department name would be printed. Likewise, for employee nodes, only the last name would be printed.

Defining Query Forms



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ADF Faces provides two different components for defining query forms declaratively:

- **Quick Query component:** Creates a quick search form that searches for a textual string against a selected criterion. The Quick Query component has a single search criteria input field. The user can select which attribute to search by selecting from a drop-down list.
- **Query component:** Creates a full-featured search form that can support multiple search criteria (dynamically adding and deleting user-defined criteria), selectable search operators, match all/any selections, seeded or saved searches, basic and advanced modes, and personalization of searches

To create a query form, drag a named view criteria item from the Data Controls panel onto a page. The named criteria represent all view criteria that have been defined for the data collection in the model. This includes named view criteria that you created explicitly on view objects. (Recall that you learned how to create named view criteria in an earlier lesson when you learned how to declaratively customize ADF Business Components.) The named criteria also include implicit named criteria called All Queriable Attributes that are created automatically for every data collection in the Data Controls panel.

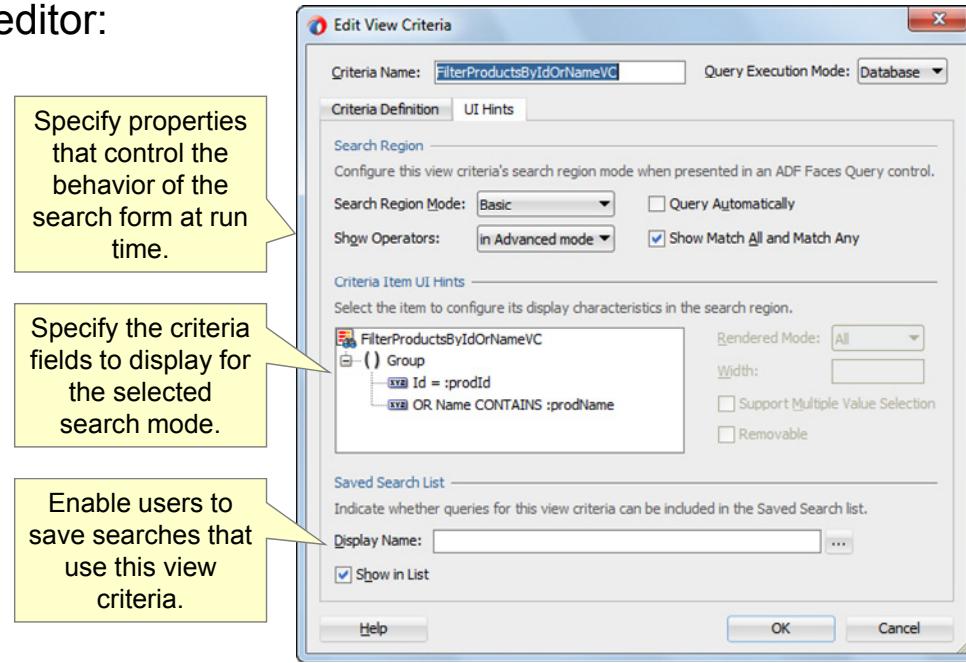
You can use the All Queriable Attributes named criteria to create either Query or Quick Query forms. However, named criteria from view objects can only be used to create Query forms. (The operators that are used in Quick Query forms are hard-coded and do not honor operators that are specified in the view criteria itself.)

When you drop a named view criteria onto a page and select the Query component, the view criteria is the basis for the initial search form. All other view criteria defined against that data collection appear in the Saved Search list. Users can then select any of the view criteria search forms as well as any saved searches.

When you drop the named criteria on the page, you can choose to create the query form only, or you can create a query form that includes a table or tree table for displaying the query results. When you choose to include a tree or tree table, JDeveloper automatically wires up the results table with the query panel. If you choose to create only the search form, you must create a component for displaying the results and populate the ResultsComponentId property of the Query component with the Id of the display component (you learn how to do that later).

More About Named View Criteria Used for Queries

View criteria are specified on the View Criteria tab of the view object editor:



ORACLE

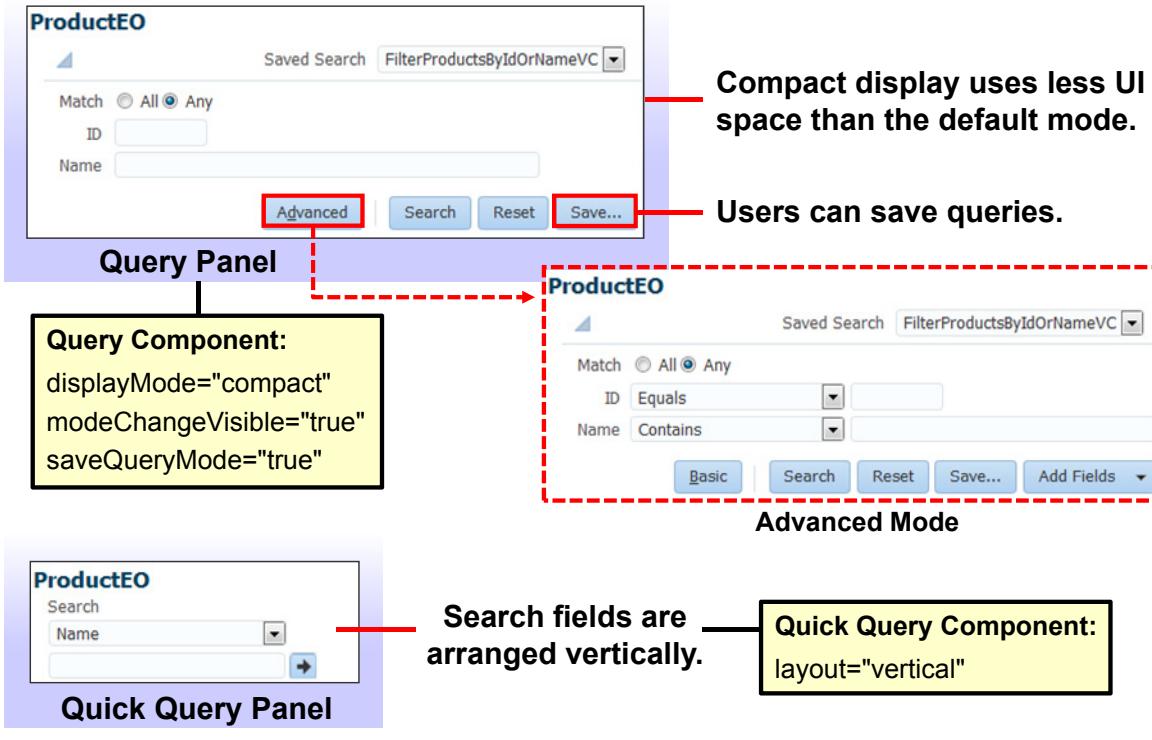
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you define named view criteria from a view object, the query search form uses the settings that are specified in the UI Hints for the view criteria. You specify view criteria on the View Criteria tab of the view object editor.

Within the UI hints, you can specify properties such as:

- The search mode to use for the query (Basic or Advanced). Basic mode has all features of Advanced mode, except it does not allow the end user to dynamically modify the displayed search criteria fields.
- When to show operators (in Basic mode, in Advanced mode, or never)
- Which criteria fields are displayed in the search form for specific search modes
- Whether the user can save queries in the Saved Search list

Specifying Query Panel Properties



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you define the view criteria in a view object, you specify the attributes and conjunctions that form the query statement and serve as a basis for the query search form. You can also specify some of the properties of the search form after you have dragged it to the page. These properties include changing the display mode, setting the Basic/Advanced mode, and specifying whether users can save and personalize searches. As with other ADF Faces components, you modify the property settings in the Properties window. Select the Query or Quick Query component (`af:query` or `af:quickQuery`) in either the Structure window or page editor and modify property settings for the query. The Query component offers a richer set of features than the Quick Query component, so its properties are more extensive.

To achieve the look and feel, as well as the behavior, of the query components shown in the slide, set the following properties for the Query component:

- **DisplayMode:** Set to `compact` to save space by turning off the header text, placing all buttons in the footer, removing borders around the header or toolbar, and so on.
- **ModeChangeVisible:** Set to `true` (the default) to render a button that enables users to toggle between Basic mode and Advanced mode. In Advanced mode, users can specify operators to use in the search, and they can add fields to the query. Also in Advanced mode, all attributes that are marked as queriable in the underlying view object can be added to the query. Depending on the size of the underlying table, you might want to disable this capability for performance reasons.

- **SaveQueryMode:** Set to `default` to allow users to save, delete, and personalize searches. When you set this property to `hidden`, the controls for saving searches and for selecting saved searches are not rendered.

For the Quick Query component, you can change the Layout property to `vertical` to arrange the fields in the search panel vertically.

Quick Query components can also be used as the starting point of a more complex transactional search that uses a query component. For example, the user can perform a quick query search on one attribute and, if successful, may want to continue to a more complex search. The Quick Query component supports this by having a built-in advanced link that is an `af:commandLink` component. You enable the link by setting the `Enabled` property of the `af:commandLink` component to `true`. You also need to implement logic that hides the Quick Query component and displays the Query component.

Specifying a Display Component for a Query Panel

The screenshot illustrates the configuration of a query panel. On the left, the 'UpdateProducts.jsf - Structure' view shows a hierarchical tree of components. A red box highlights the 'af:query' component under 'af:panelHeader - Search for a Product'. Another red box highlights the 'af:panelFormLayout' component under 'af:panelHeader - Search Result'. On the right, two property editor windows are shown. The top window is 'Query - Properties' for the 'af:query' component, with the 'ResultComponentId' field set to '::pf1'. The bottom window is 'Panel Form Layout - Properties' for the 'af:panelFormLayout' component, with the 'Id' field set to 'pf1'. A yellow callout box points from the text 'Query results are displayed in a form.' to the 'af:panelFormLayout' component in the structure view.

Set the value of ResultComponentId to the ID of the display component.

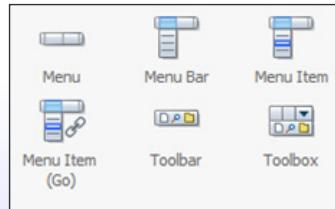
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you create a form that includes a table or tree table for displaying the query results, JDeveloper automatically wires up the results table with the query panel. If you choose to create only the search form and use a different type of component to display the results of the query, you must populate the ResultsComponentId property of the Query component with the ID of the display component.

The example shows a query panel that uses an ADF form (with navigation buttons) to display the query results. Notice that the ResultsComponentID property specifies the ID for the Panel Form Layout component.

Defining Menus and Toolbars

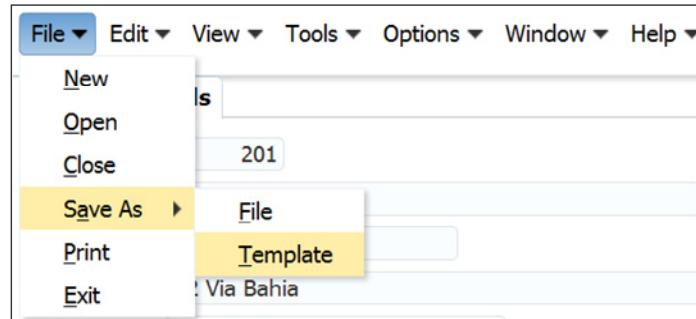


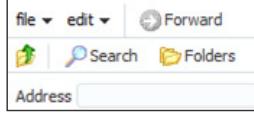
Menus and Toolbars

ORACLE ADF Faces Rich Client

Use menu and toolbar components for:

- **Menu bars** 
- **Menus and menu items in menu bars**



- **Toolbars** 
- **Toolboxes** 

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You use menu and toolbar components to build menus and toolbars for your application. The menus and toolbars can be as complicated as you need them to be. They can include icons, they can trigger actions (including navigation), they can invoke operations, and so on. You can even create toolboxes that include an assortment of menus and buttons. The menus can be nested within other menus to create cascading menus.

As with other ADF Faces components, you specify the behavior and appearance of menu and toolbar components declaratively by setting properties in the Properties window. For example, if the menu item invokes an operation, the `ActionListener` property would specify a binding to an operation, such as `#{bindings.Commit.execute}`. If the menu item invokes navigation, the `Action` property would specify a navigation outcome, such as `showDetail` (which takes the user to an Order Details page). The component properties also enable you to specify access keys and accelerators for defining keyboard shortcuts.

You learn more about using menus and toolbars for navigation later in the course.

Defining Layout Components



Use layout components to arrange other components on the page:

The image shows a user interface example using a "Panel Grid Layout" component. The layout is divided into two columns by a vertical dashed red line. The left column contains several input fields: "ID" (value: 201), "Name" (value: Unisports), "Phone" (value: 55-2066101), "Address" (value: 72 Via Bahia), "City" (value: Sao Paulo), and "State". The right column contains: "Country" (dropdown menu showing Brazil), "Zip Code" (input field), "Credit Rating" (input field with value 1), and "Sales Rep" (input field with value 12). The entire grid structure is labeled "Panel Grid Layout". Within the grid, specific cells are highlighted with dashed red boxes and labeled: "Grid Row" (the row containing the first two columns), "Grid Cell" (the first column of the first row), and "Grid Cell" (the second column of the first row).

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

You use layout components to arrange other components on the page. ADF Faces provides a wide assortment of layout components for arranging other components on a page by using elements like panels, decorative boxes, grids, tabs, accordions, and dashboards. The example in the slide shows a Panel Grid Layout component, which you use to lay out components in a grid composed of rows and cells.

You learn more about layout components later in the course.

Defining Data Visualization (DVT) Components

Data Visualization (more available)

ORACLE ADF Faces Rich Client

Use data visualization components for:

- Gantt charts and timelines
 - Hierarchical data
 - Maps
 - Many others!
- Gantt chart showing project timelines for four employees across four quarters of 2012.

Treemap visualization showing hierarchical data categorized by color-coded territories.

Bubble chart showing four data series with varying sizes and colors.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

You use data visualization (DVT) components to render hierarchy viewers, pivot tables, sunbursts, treemaps, dynamic charts, graphs, gauges, timelines, geographic and thematic maps, and other graphics that provide a real-time view of the underlying data.

You specify the behavior and appearance of DVT components declaratively by using dialog boxes and the Properties window.

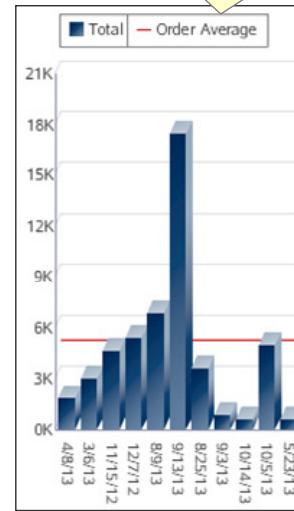
You can quickly create DVT components by selecting a collection in the Data Controls panel, dragging it to the page, and selecting the kind of component that you want to create. When you do this, JDeveloper presents you with a dialog box for defining the details of the component and then generates the required bindings based on the settings that you specify.

Shaping Data for DVT components

To shape data for a DVT component:

- Use the default view object to display basic data in a chart component.
No special shaping is required.
- Use transient attributes to calculate aggregate values.
- Create read-only view objects that include group by and order by clauses to shape the data for graphical display.

Transient attribute uses Groovy to calculate the order average:
`OrdVO.avg ("Total")`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When creating DVT components, you must first determine the shape of the data that you want to present graphically.

For basic DVT components, such as a chart that displays basic data, you can often use the default view object generated for an entity object (rather than doing anything special to shape the data). The only condition is that at least one of the attributes selected in the query must be numeric.

If you need to calculate an aggregate value, but you are using values from more than one view object (or if the DVT component does not aggregate values for you), you can create a transient attribute and use a Groovy expression to calculate the value. Use the transient attribute when you define the bindings for the DVT component. The example in the slide shows a DVT component that uses a transient attribute called OrderAverage in the CustomerVO view object to calculate the average total of all orders for the selected customer.

When the DVT component requires data that is grouped or ordered in a specific way, create read-only view objects that contain the group by and order by clauses to shape the data specifically for graphical display.

ADF Faces Resources

Learn more about ADF Faces components:

- *ADF Faces Tag Reference*
<http://docs.oracle.com/middleware/1212/adf/TROAF/index.html>
- *ADF Faces Data Visualization Tools Reference*
<http://docs.oracle.com/middleware/1212/adf/DVTTR/index.html>
- ADF Faces Rich Client demonstrations
<http://jdevadf.oracle.com/adf-richclient-demo/faces/index.jspx>
- *Developing Web User Interfaces with Oracle ADF Faces*
<http://docs.oracle.com/middleware/1212/adf/ADFUI/index.html>



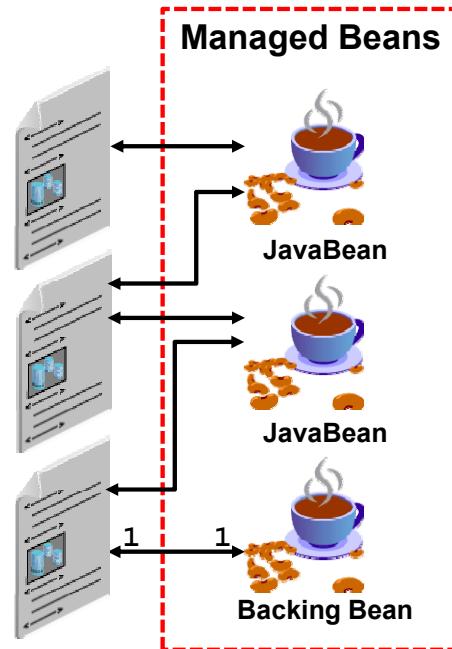
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Adding Application Code to Managed Beans and Backing Beans

Use managed beans and backing beans to add application-specific code:

- Managed beans are POJOs that are registered with the JSF container.
- A backing bean is a special use case of a managed bean that is associated with a specific page.

Rule: Implement functionality in code *only* if it cannot be implemented declaratively.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To ensure a clean separation between presentation and business logic, most of the code for an ADF application exists in the Business Services layer. However, there will be situations when you need to write code to address a specific UI requirement. When that happens, you add the UI-specific code to a managed bean.

Review of Managed Beans and Backing Beans

Here is a brief review of the key concepts that you learned earlier about managed beans and backing beans.

Managed beans are Java objects that encapsulate application-specific code and data. Managed beans are often used to validate data, handle events, store data between pages, and perform navigation. Managed beans are simply Plain Old Java Objects (POJOs) that are registered with the JSF container and include a no-argument constructor; managed beans do not inherit from a specific base class. Managed beans have a scope that determines how long they are held in memory.

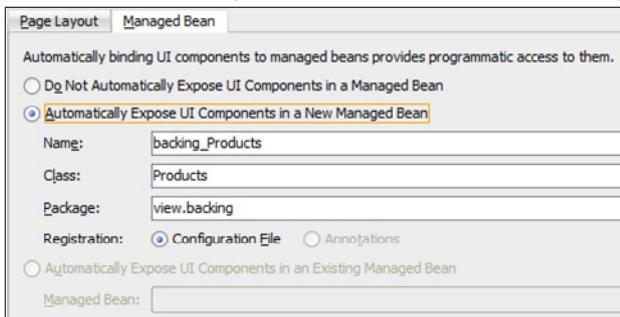
A *Backing bean* is a special use case of a managed bean that is associated with a specific page; the bean “backs” the page. For example, a backing bean might contain logic that disables or enables a field based on specific criteria.

The general rule for adding code to a managed bean is that you should implement functionality in code only if you cannot implement the functionality declaratively.

Creating a Backing Bean as a Managed Bean

You can create a backing bean for a page by:

- Automatically exposing the UI components in a backing bean when you create a new page



Not recommended:
Exposes getters and
setters for all UI
components

- Auto-binding an existing page to a managed bean (Design > Page Properties > Component Binding tab)
- Creating a backing bean when you bind the action property of a component to a method in a backing bean

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

By default, JDeveloper does not create a backing bean for each new JSF page. However, you can choose to do this when you use the Create JSF Page Wizard to create a new page. On the Managed Bean tab, you can choose to automatically expose UI components in a new or existing managed bean. If you select one of these options, JDeveloper creates a backing bean (or uses an existing one). From that point forward, whenever you drag a component to the page, JDeveloper inserts a bean property for each component and uses the binding attribute to bind component instances to those properties, allowing the bean to accept and return component instances.

Unless you are creating a simple page (for example, a login page), automatically exposing UI components in a backing bean is not generally recommended because the backing bean exposes getters and setters for all UI components on the page. This can produce unnecessary overhead if the backing beans are used with large and complex pages (unless you manually delete all the component references that are not used).

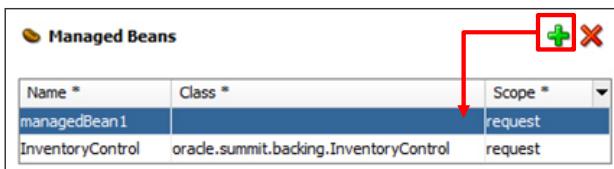
For an existing page, you can chose to auto-bind the page to a managed bean. With the JSF page open, select Design > Page Properties and then click the Component Binding tab. On the Component Binding tab, select the Auto Bind check box and then select (or create) the managed bean. Using this approach has the same drawbacks as the previous approach because it exposes the getters and setters for all UI components on the page.

The most straightforward way to create a backing bean for a page (without exposing every single UI component) is to create a backing bean when you bind the action property of a UI component to a method. With this approach, you can generate a backing bean and expose only the UI component that you are working with (you learn how to do this later in the lesson).

When you use JDeveloper to create a backing bean, it automatically configures the backing bean as a managed bean and registers the bean with the JSF container. Next, you learn how to create and register a managed bean that is not associated with a specific page.

Registering a Bean Class as a Managed Bean

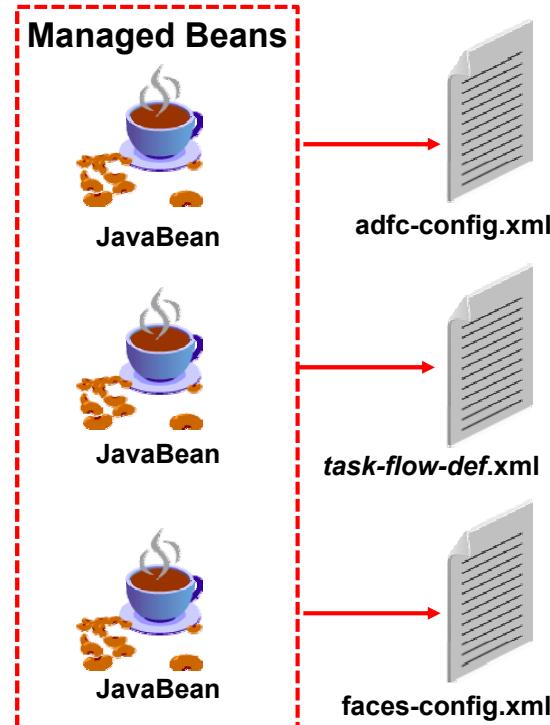
Configure the bean as managed in the overview editor:



Managed Bean tab

Where do I configure the bean?

- In **adfc-config** for applications that use ADF Model and ADF Controller
- In the **task flow definition** for beans that are used only within the task flow
- In **faces-config** for standard, non-ADF JSF applications only



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

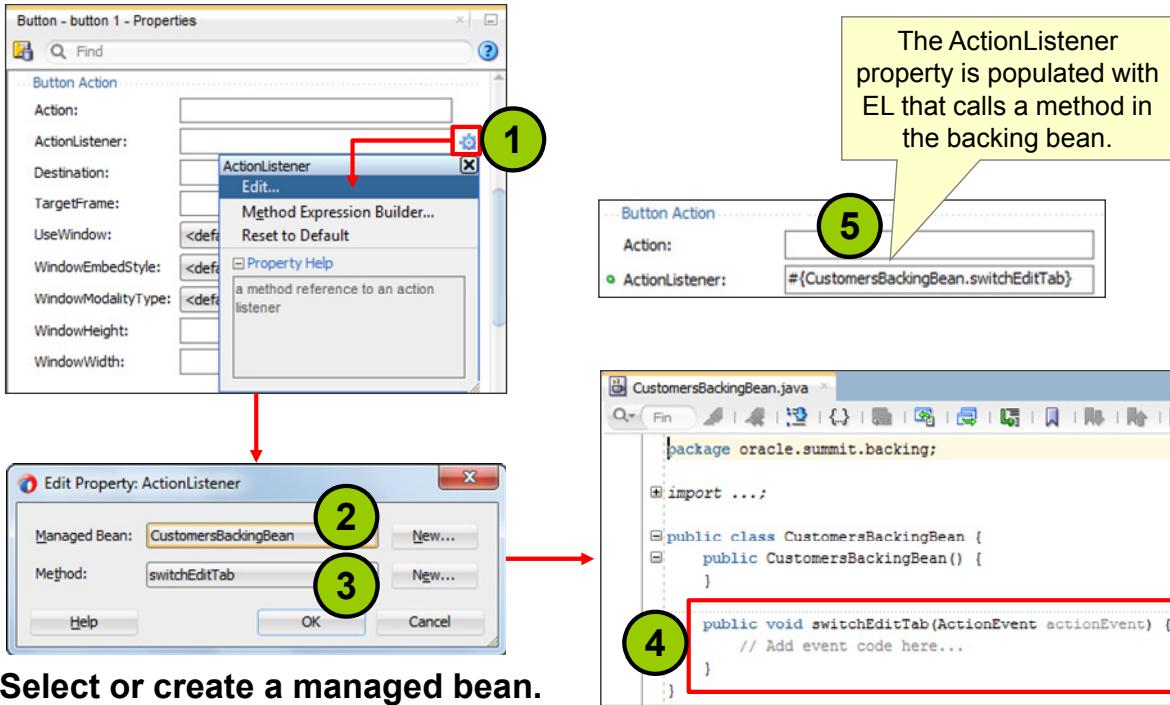
When you use JDeveloper to create a backing bean for a specific page, JDeveloper automatically registers the Java class as a managed bean.

You can register existing Java classes as managed beans by configuring them manually in the `adfc-config.xml` file, the task flow definition file, or the `faces.config.xml` file. In a standard, non-ADF JSF application, managed beans are registered in the `faces-config.xml` file. If you plan to use the ADF Model data binding layer and ADF Controller, you need to register managed beans in the `adfc-config.xml` file or a task flow definition file. Managed beans registered in task flow definition files are visible only to activities that execute in the same task flow.

To register a managed bean, open the configuration or task flow definition file, and select the Overview tab at the bottom of the editor. On the Managed Beans navigation tab, click the Add (green plus sign) icon next to the Managed Beans table. Notice that JDeveloper adds a row for the managed bean to the table. In the Properties window, specify the bean name, the class name, and the memory scope for the bean.

Note: Memory scopes are covered later in the course.

Calling a Managed Bean from a JSF Page



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You have already learned how to bind a command component, such as a button or a link, to an action declaratively by populating the Action property with EL, such as `# {bindings.Commit.execute}`.

If you need to bind a command component to a method in a managed bean, or to multiple action bindings, you can use the ActionListener property to specify a method to call when the user clicks the button or link. Likewise, you can use the ValueChangeListener property on input text components to specify a method to call when the user changes a value in a field.

To call a managed bean from a component:

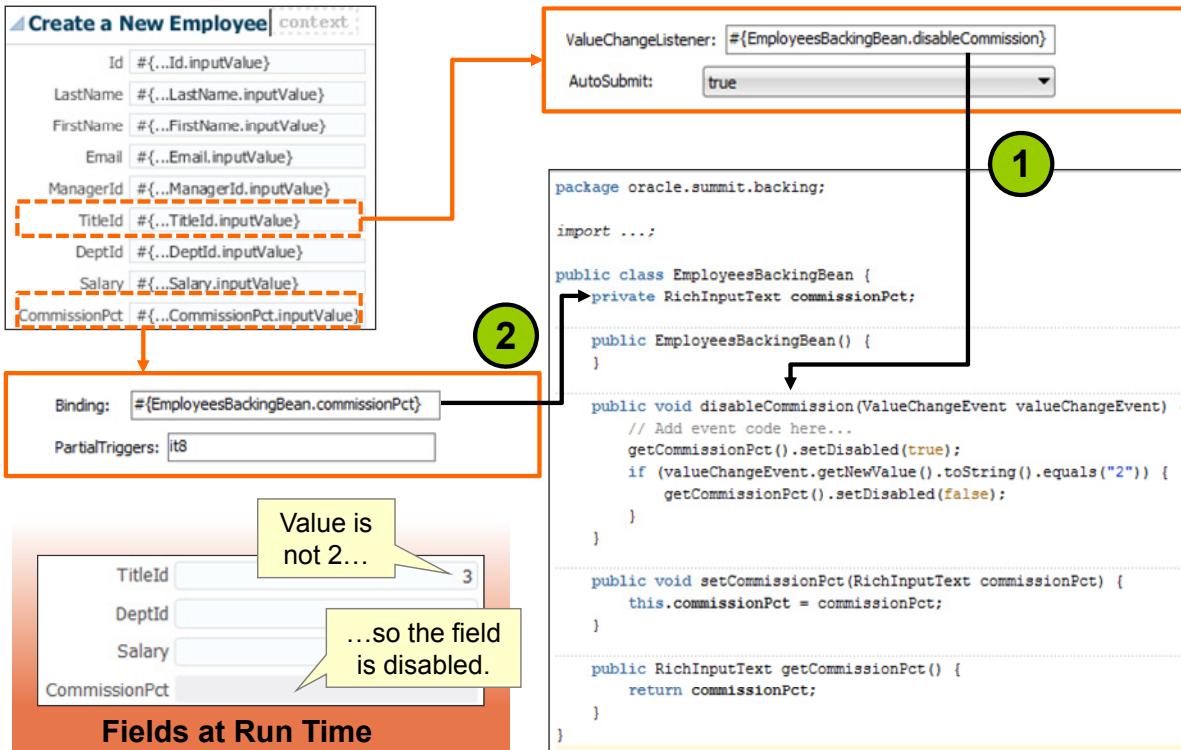
1. Click the Property Menu icon next to the component listener property in the Properties window (for example, the icon next to ActionListener) and select Edit.
2. In the Managed Bean list, either select an existing managed bean or click the New button to create a new managed bean.
3. In the Method list, either select a method or click new and create a method. This method is called when the user triggers an action event on the component.
4. If you created a new managed bean and chose to generate the class file, notice that JDeveloper creates the file with skeleton code for the method that you created.

5. Also notice that the listener component property (ActionListener, in this example) is populated with an EL expression that resolves to the method in the backing bean that you created.

When a user clicks the button in the example, the event code defined in the switchEditTab method executes. The framework is responsible for instantiating an instance of the bean, which remains in memory for a specific period of the processing life cycle as defined by its scope.

There are other types of component listeners that are available for binding UI components to methods in managed beans. You learn about those listeners later in the lesson about responding to application events.

Example: Enabling and Disabling Components Programmatically



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, you learn how to enable and disable a component programmatically by calling a method in a backing bean. Suppose that the application includes a form for creating a new employee. You want to automatically disable the commission field in the form if the new employee is not a sales representative.

First, you go to the ValueChangeListener property of the TitleId field and create a managed bean called EmployeesBackingBean that contains a method called disableCommission. The disableCommission method checks to see if the value of the TitleId field is equal to 2 (the ID for sales representatives). If the TitleId is equal to 2, the field is enabled. If the TitleId is not equal to 2, the field is disabled.

Next, you expose the commissionPct field in the managed bean so that the backing bean can access the component and disable it. To do this, you go to the binding property of CommissionPct field and select Edit from the property menu to expose the component (along with getters and setters) in the backing bean.

To get this logic to work, you also need to set the AutoSubmit property of the TitleId field to true (to submit the value change), and you need to set the PartialTrigger property of the CommissionPct field to the ID of the TitleId field (it8 in the example).

This example shows how to disable components programmatically. However, there is a flaw in the example. For this particular use case, you can implement the same behavior without writing code in a managed bean. The next example shows you how to do that.

Example: Enabling and Disabling Components Declaratively

The diagram illustrates the configuration of a form component and its runtime behavior. On the left, the 'Create a New Employee' form is shown with various fields: Id, LastName, FirstName, Email, ManagerId, TitleId, DeptId, Salary, and CommissionPct. The CommissionPct field is highlighted with a dashed red border. On the right, the component's configuration is displayed, showing the following properties:

- AutoSubmit:** true
- Disabled:** #{bindings.TitleId.inputValue!=2}
- PartialTriggers:** it8

A callout box indicates that the CommissionPct field is disabled if EL evaluates to true. Below this, a text box states: "Setting the Disabled property of the CommissionPct field achieves the same behavior as the previous example, but without the overhead of a backing bean." On the far right, the form is shown at run time. The CommissionPct field is grayed out and has a tooltip: "Value is not 2...". A speech bubble says "...so the field is disabled".

Fields at Run Time

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.



You can achieve the same behavior as the previous example by using an expression in the Disabled property of the CommissionPct field that returns true if the value of the TitleId field is not equal to 2 (the ID for salesrRepresentatives):

```
# {bindings.TitleId.inputValue!=2}
```

This is a much simpler approach to enabling and disabling fields in the UI, and it is a perfect example of when you should use functionality available in the framework instead of developing your own logic. However, if you have a more complicated use case where the logic for disabling a field is more complex than simply checking the value of another attribute, using a backing bean might be the best approach.

Summary

In this lesson, you should have learned how to:

- Display a selection list of values
- Display tabular data in tables
- Display hierarchical data in trees
- Define and use search forms and display the results
- Display data graphically
- Customize the UI programmatically by creating and configuring a backing bean



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 9 Overview: Adding Functionality to Pages

This practice covers the creation of:

- A search page
- A category tree
- A graph
- An input form
- A sortable table



Copyright © 2014, 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 NTC - Nucleo de Tecnologia e Conhecimento em Informatica LTDA use only

10

Adding Advanced Features to Task Flows and Page Navigation

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

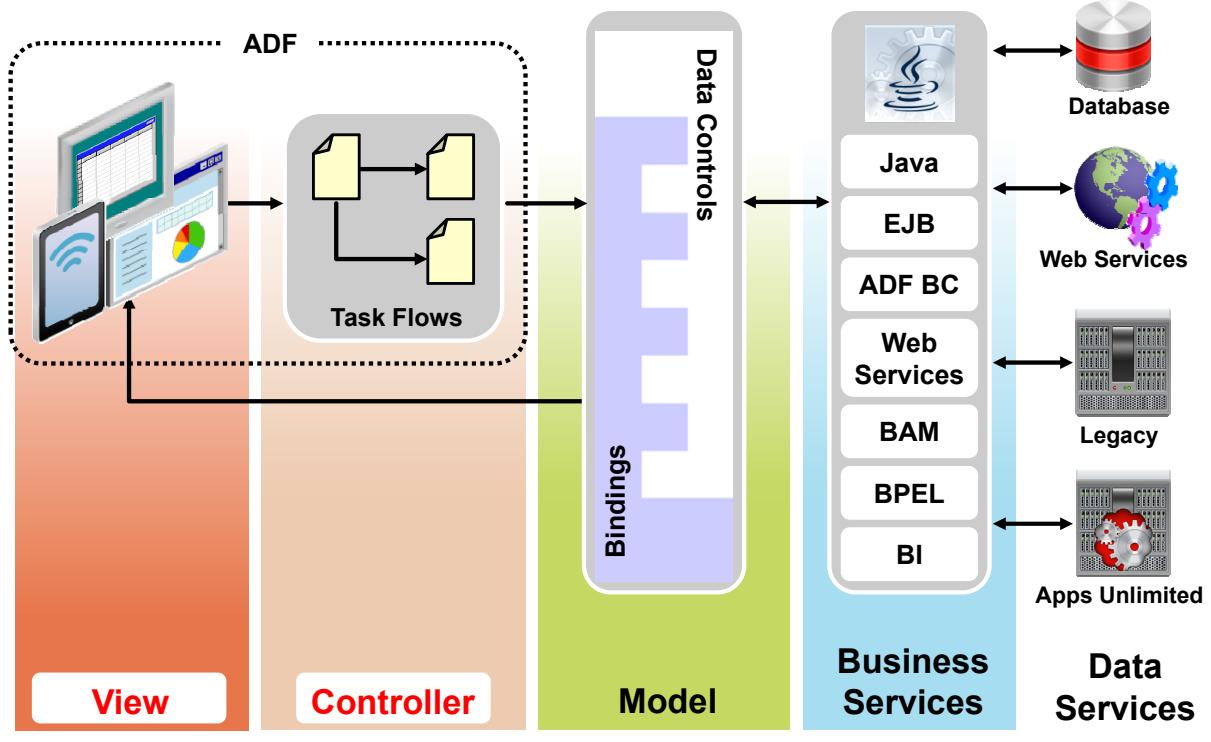
After completing this lesson, you should be able to:

- Describe the difference between bounded and unbounded task flows
- Create routers for conditional navigation
- Call methods and other task flows
- Create menu items, menu bars, pop-up menus, context menus, and navigation panes
- Define breadcrumbs and trains
- Create a page fragment and use it in a bounded task flow
- Use a bounded task flow as a region



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Adding Advanced Features to Task Flows and Navigation



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In an earlier lesson, you learned about the fundamentals of building the Controller layer. You learned about the ADF Controller and how it extends the capabilities of the JSF Controller. You also learned how to use task flows to define control flow in an application, and you learned about some basic navigation components (such as links and buttons) that are used to trigger navigation between pages.

In this lesson, you learn how to add complexity to page navigation by using bounded task flows, calling other task flows, adding regions, and using ADF Faces Rich Client components to create sophisticated navigation controls such as menus, navigation panes, breadcrumbs, and trains.

More about Task Flows

ADF task flows are logical units of page flows that:

- Offer advantages over JSF page flows because ADF task flows:
 - Can be broken into a series of modular flows that call one another
 - Are not limited to page navigation. They can call methods, other task flows, or activities that perform other tasks.
 - Are reusable
 - Are closely aligned with the concept of business processes
 - Provide a shared memory scope
- Can be either unbounded or bounded



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As you learned earlier, ADF task flows provide a modular approach for defining control flow in an application. Instead of representing an application as a single large JSF page flow, you can break it up into a collection of reusable task flows. In each task flow, you identify application activities, which are the work units that must be performed for the application to complete. An activity represents a piece of work that can be performed when running the task flow.

ADF task flows offer the following advantages over standard JSF page flows:

- Unlike JSF page flows, which represent the entire application in a single navigation file, ADF task flows can be divided into a series of modular flows that call one another.
- ADF task flows are not limited to page navigation. They can call methods, other task flows, or activities that perform other tasks, such as conditional navigation.
- ADF task flows are reusable in the same application or an entirely different application.
- ADF task flows are closely aligned with the concept of business processes.
- Whereas JSF page flows provide no shared memory scope between multiple requests (except for session scope), ADF task flows provide a shared memory scope that enables data to be passed between activities in the task flow. (You learn more about memory scopes in a later lesson.)

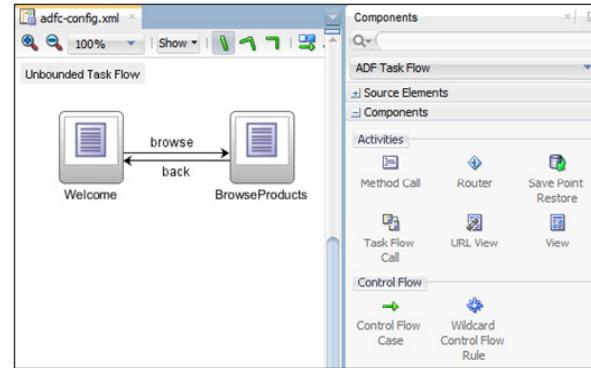
The ADF controller supports two types of task flows: unbounded and bounded.

Unbounded ADF Task Flows

Unbounded task flows often serve as the entry point to an application and have the following characteristics:

- First entry on task flow stack—the outermost task flow
- Represent the top level of the application and therefore define the user's entry point (allow multiple entry points)
- Have view activities that can be accessed via a URL
- Contain view activities that support browser bookmarks and reentry

Double-click
adfc-config.xml
to edit the task flow.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An *unbounded task flow* is a set of activities, control flow rules, and managed beans that interact to enable a user to complete a task. The unbounded task flow consists of all activities and control flows in an application that are not included in a bounded task flow. Although there can be many bounded task flows in an application, there is only one unbounded task flow, which is assembled at run time by combining one or more `adfc-config.xml` files. The set of files that is combined to produce the unbounded task flow is referred to as the application's *bootstrap configuration files*.

The unbounded task flow represents the top level of the application and therefore defines the user's entry point (an entry point is a view activity that can be directly requested by a browser). Any view activity in the unbounded task flow can be accessed via a URL, enabling users to bypass defined flows between activities.

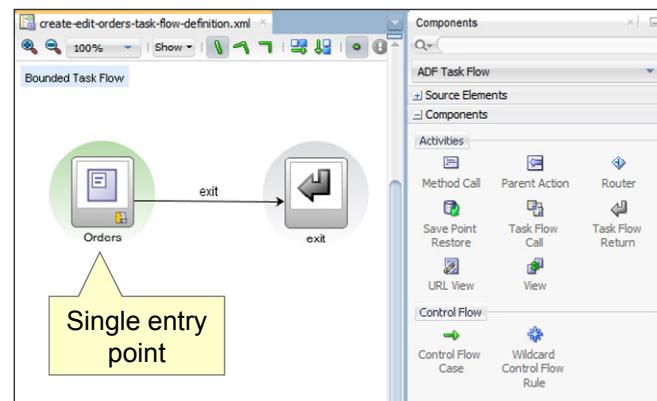
As a user navigates within an application, a task flow stack is maintained to keep track of the calls from task flow to task flow. An unbounded task flow always logically exists as the first entry on the task flow stack, but would simply be empty if no unbounded task flow is defined.

An unbounded task flow does not contain the well-defined boundary or single entry point of a bounded task flow. Typically, unbounded and bounded task flows are used together, with an unbounded task flow pointing to one or more entry points to the application.

Bounded Task Flows

Bounded task flows are named, modular blocks of task flow functionality that have the following characteristics:

- Single entry point and one or more defined exit points
- Strict navigation path defined by the developer
- Well-defined transaction boundary
- Declarative transaction management
- Declarative Back button support
- Acceptance of input parameters and return values



ORACLE

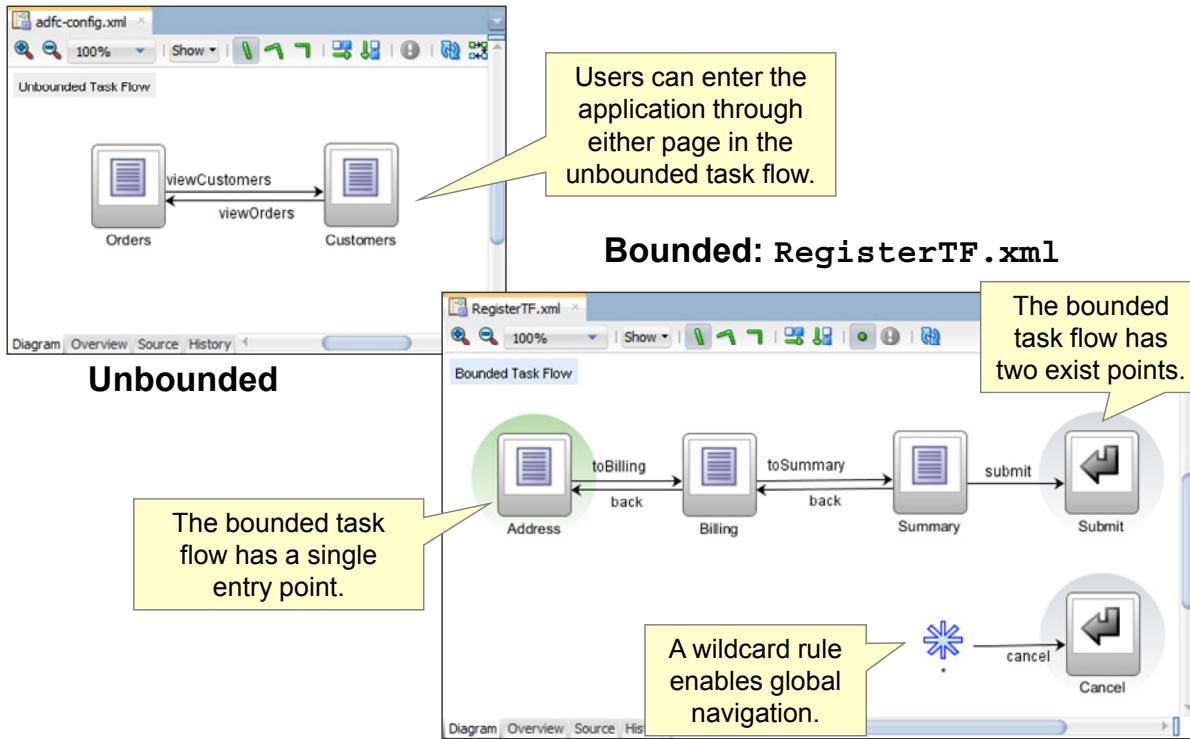
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A bounded task flow is a specialized form of task flow that, in contrast to the unbounded task flow, has a single entry point and zero or more exit points. It contains its own set of private control flow rules, activities, and managed beans. A bounded task flow allows reuse, parameters, transaction management, and reentry, and it can render within an ADF region in a JSF page. There can be many bounded task flows in an application. Each bounded task flow represents a reusable application flow that can be referenced from any other task flow. A bounded task flow is the only type of task flow that can be used as a region on a page.

Bounded task flows have the following characteristics:

- Single entry point and one or more well-defined exit points
- Strict navigation path defined by the developer (Users cannot bypass activities in the flow.)
- Well-defined boundary with a transaction beginning and end
- Declarative support for transaction management
 - Ability to begin a new transaction upon bounded task flow entry
 - Ability to commit or roll back upon bounded task flow exit
- Declarative support for Back button navigation
- Ability to pass input parameters from the bounded task flow caller and to return values back to the caller upon exit

Task Flow Example



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows:

- **An unbounded task flow:** Consists of two pages. There is no specific entry or exit point in this task flow, in which users can enter the application through either page. The unbounded task flow defines the “top-level” flow.
- **The RegisterTF task flow:** A bounded task flow with a single entry and two exit points. The task flow also includes a global control flow rule. Although the example shows a task flow that contains a linear series of steps, a bounded task flow (like any task flow) may also contain branches.

A typical application is a combination of an unbounded task flow and one or more bounded task flows. The unbounded task flow can contain task flow call activities that call the bounded task flows from within the top-level unbounded task flow. This design has various advantages over putting everything in one diagram:

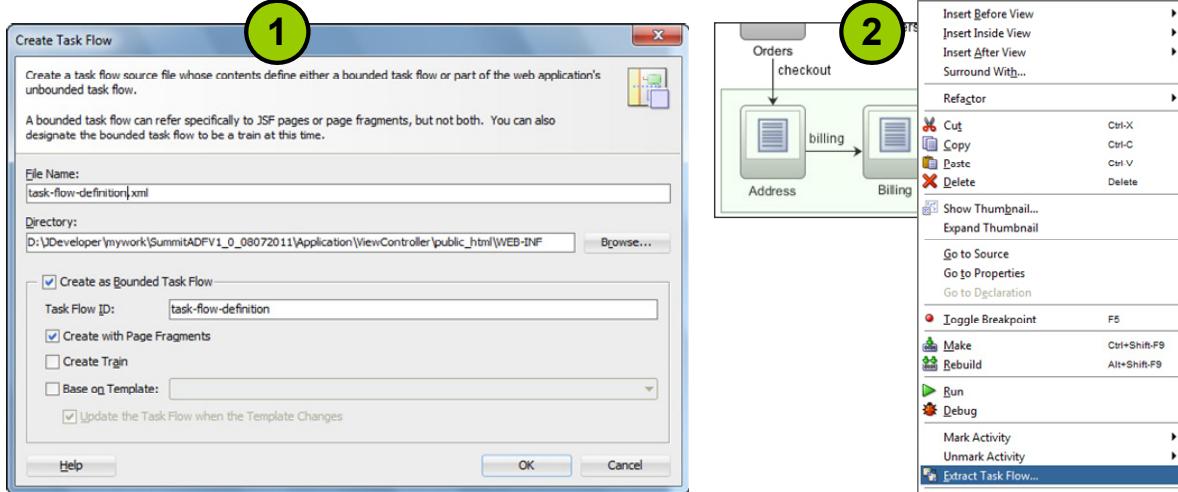
- It breaks the diagram into modules and makes it more readable.
- The individual modules (bounded task flows) make it easier for multiple developers to work on pieces of the application.
- Bounded task flows can be reused by other application developers.

Every Fusion web application must contain an unbounded task flow, even if the unbounded task flow is empty.

Review: Creating Task Flows

You can create a task flow by doing either of the following:

1. Using the Create Task Flow dialog box
- OR
2. Extracting part of an existing task flow (to a bounded flow)



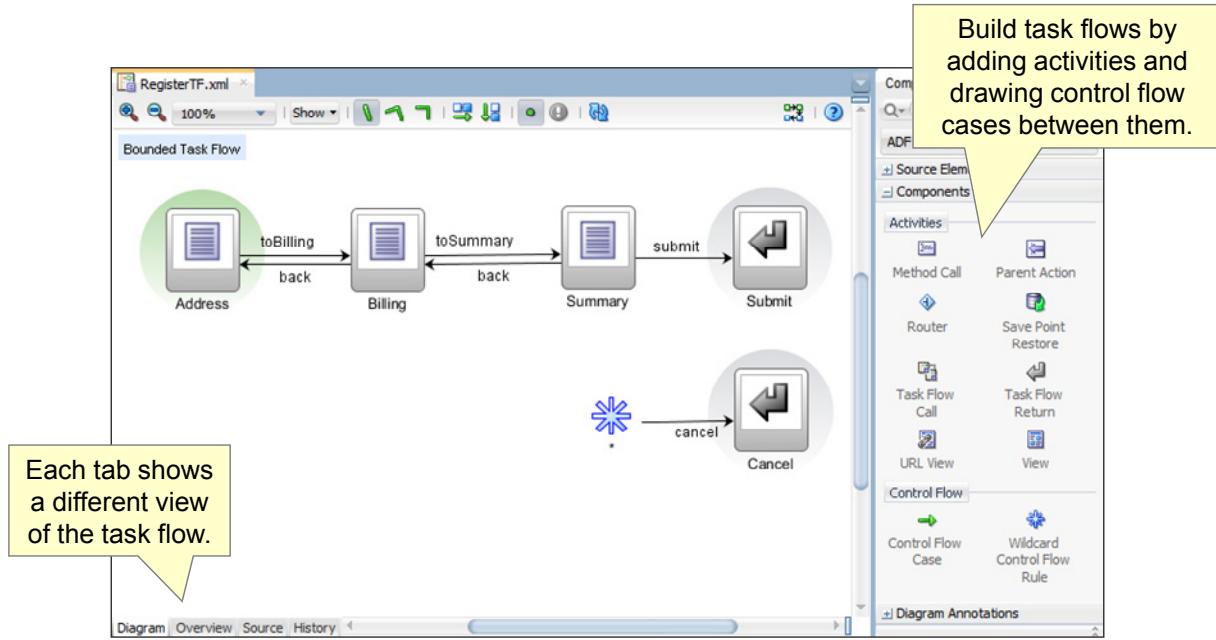
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In an earlier lesson, you learned how to create a task flow by using the Create Task Flow dialog box (right-click a ViewController project and select New > Task Flow). When you create a task flow in this way, you can choose to create an unbounded or bounded task flow, to create the task flow as a train, or to base the task flow on a template.

Another way of creating a task flow (assuming that you are creating a bounded task flow) is to extract part of an existing flow as a new bounded task flow. Using this approach enables you to model the overall page flow of your application and then extract parts of the page flow into bounded task flows.

Working with the Task Flow Editor



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You use the task flow editor in JDeveloper to create and edit task flows. The following four tabs are at the bottom of the editor and enable you to see different views of the task flow. All of the tabs remain coordinated as you edit the file.

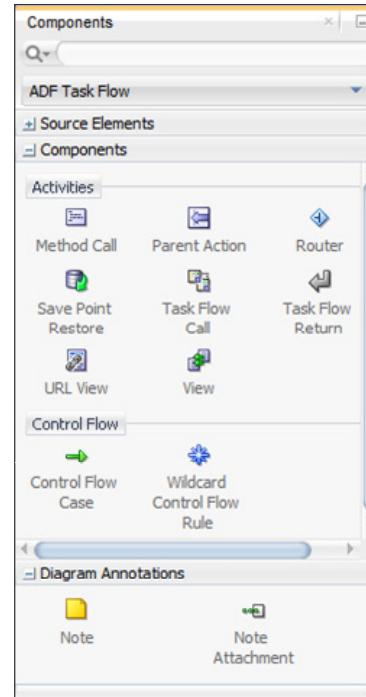
- **Diagram:** Provides a visual view that enables you to drag source elements, components, or diagram notations from the Components window to define the task flow
- **Overview:** Contains panels that enable you to configure the task flow, including a Managed Beans tab for registering the managed beans used in the task flow and a Parameters tab for specifying input parameter definitions and return values
- **Source:** Displays the XML source code for the task flow, which you can edit directly
- **History:** Shows a history of changes

As you learned earlier, you build task flows by adding activities to the task flow diagram and then drawing control flow cases between the activities.

Task Flow Activities

Some commonly used activities for building task flows:

- View activity
- Router activity
- Method call activity
- Task flow call activity
- Task flow return activity
- URL view activity



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are different types of task flow activities available for building task flows, including the following:

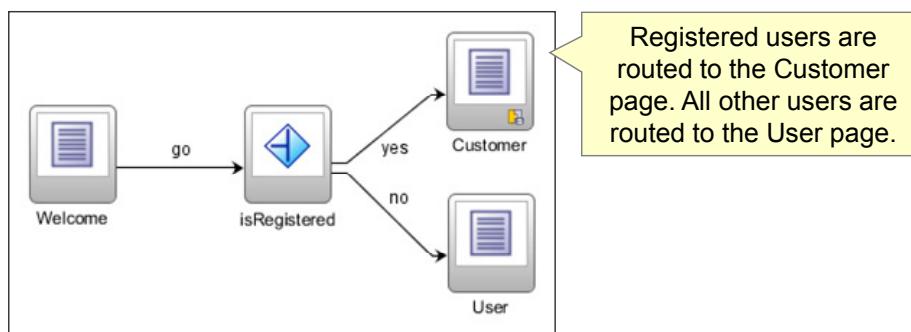
- **View activity:** Displays a page or page fragment
- **Router activity:** Evaluates a declarative expression to produce a control flow outcome
- **Method call activity:** Invokes application logic (Java method) from within the task flow
- **Task flow call activity:** Calls an bounded task flow
- **Task flow return activity:** Defines the exit point of a bounded task flow
- **URL view activity:** Redirects the view port to any URL addressable resource

You have already built some simple task flows by using view activities. In the following slides, you learn how to enhance task flows by including some of the other types of activities that are available for building task flows.

Adding Conditional Navigation to a Task Flow

Router activities:

- Use expressions that evaluate to true or false
- Define navigation conditionally based on the result of the expression



ORACLE

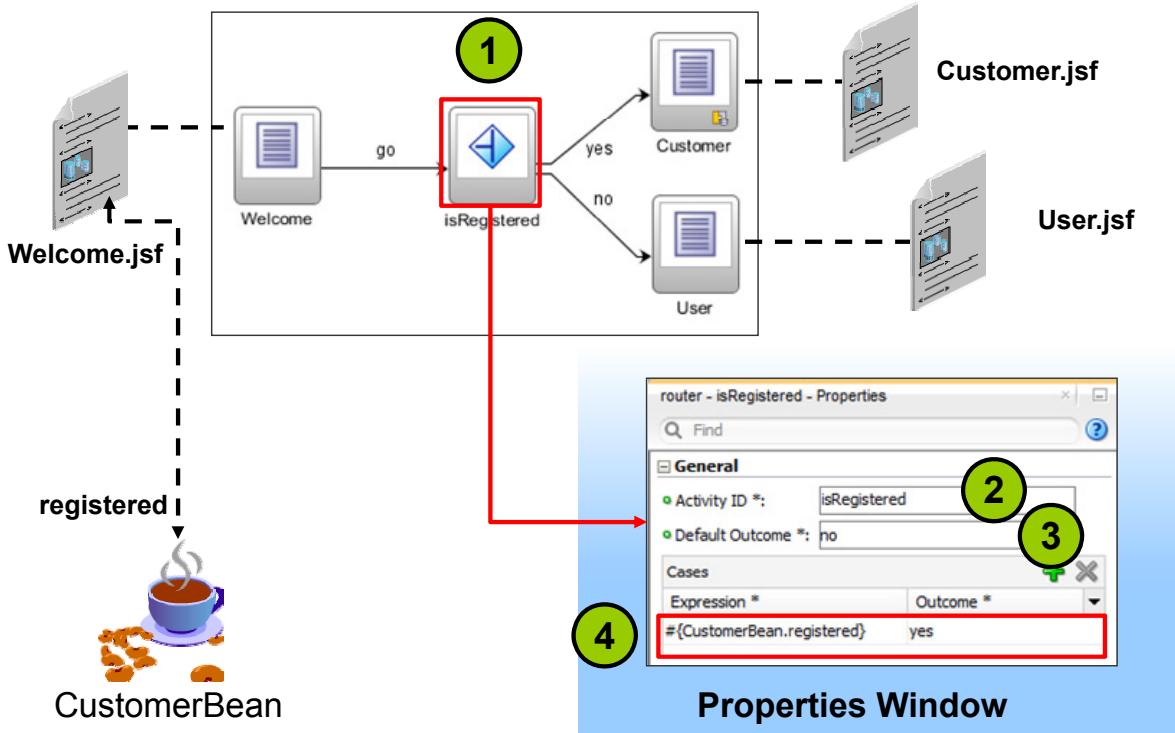
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the router activity in a task flow to declaratively route control to activities based on logic specified in an EL expression. As shown in the example in the slide, a router has multiple control flow cases that lead from the router to other activities.

Each control flow case corresponds to a different outcome. In the properties for the router component, you specify one or more expressions that resolve to either true or false. For each expression, you specify an outcome that corresponds to a control flow case. If the expression resolves to true, control passes to the control flow case that is associated with the specified outcome.

For example, suppose you want to route registered customers to a specific page. You could use a router activity (such as `isRegistered`, as in this example) to route registered users to a Customer page and all other users to a User page. In the next slide, you learn how to define the router activity.

Defining Router Activities



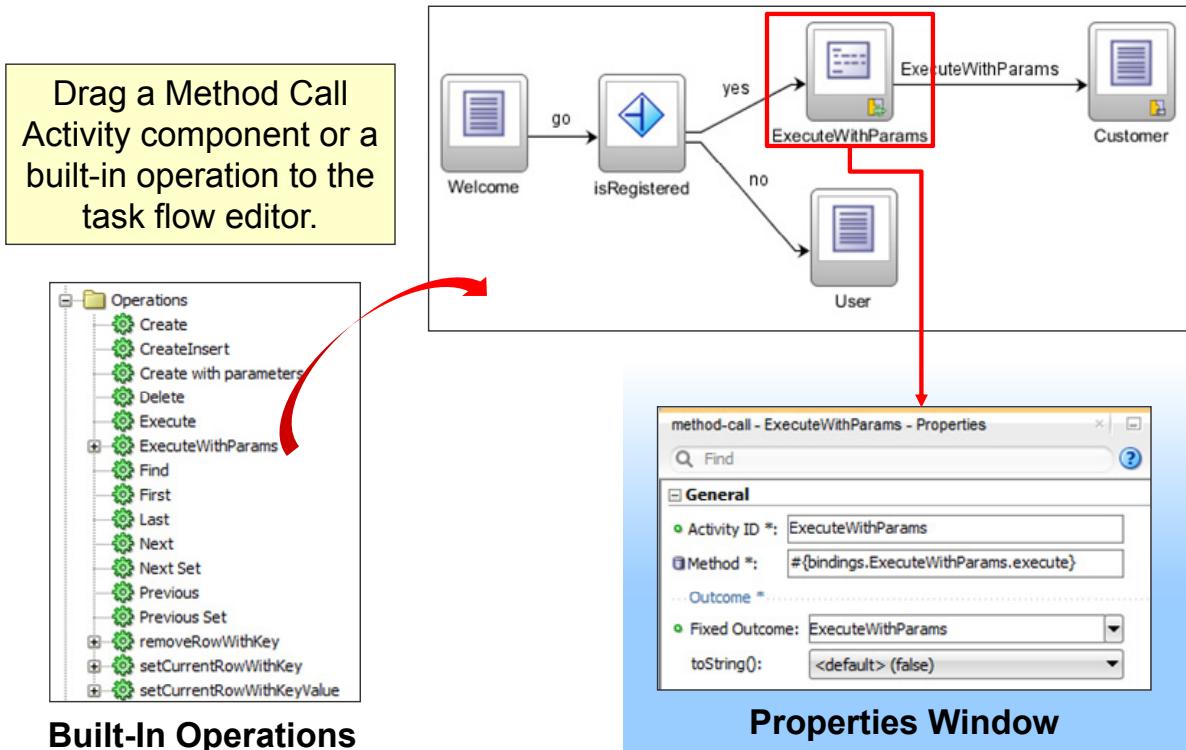
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To define a control flow that uses a router activity:

1. In the Components window, drag the router activity to the task flow diagram and create all the control flow cases and activities to which the router leads.
2. Select the router activity and, in the Properties window, enter an activity ID (for example, isRegistered). The activity ID is an identifier that is used to reference the router activity within the metadata.
3. In the Properties window, enter a default outcome. This outcome is returned if none of the cases for the router activity evaluates to true, or if no cases are specified. If no case is specified for the default outcome, an error occurs if the default outcome is returned.
4. In the Cases area of the Properties window, specify an expression and associated outcome for each of the router's cases. A case is a condition that, when evaluated to true, returns an outcome. For each case, you must enter:
 - **Expression:** An EL expression that evaluates to true or false
 - **Outcome:** The value that is returned by the router activity if the EL expression evaluates to true. In the example, if the value of the registered property is true, the control flow case associated with the "yes" outcome executes. Otherwise, the control flow case associated with the "no" outcome executes.

Calling a Method from a Task Flow



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A method call activity enables you to call a custom or built-in method that invokes application logic from anywhere within an application's control flow. You can specify methods to perform tasks such as initialization before displaying a page, cleanup after exiting a page, exception handling, and so forth. You can pass parameters to the method and control values that are returned by the method call. You create a method call activity by adding a Method Call Activity component to a task flow or by dragging an operation from a data control to the task flow editor.

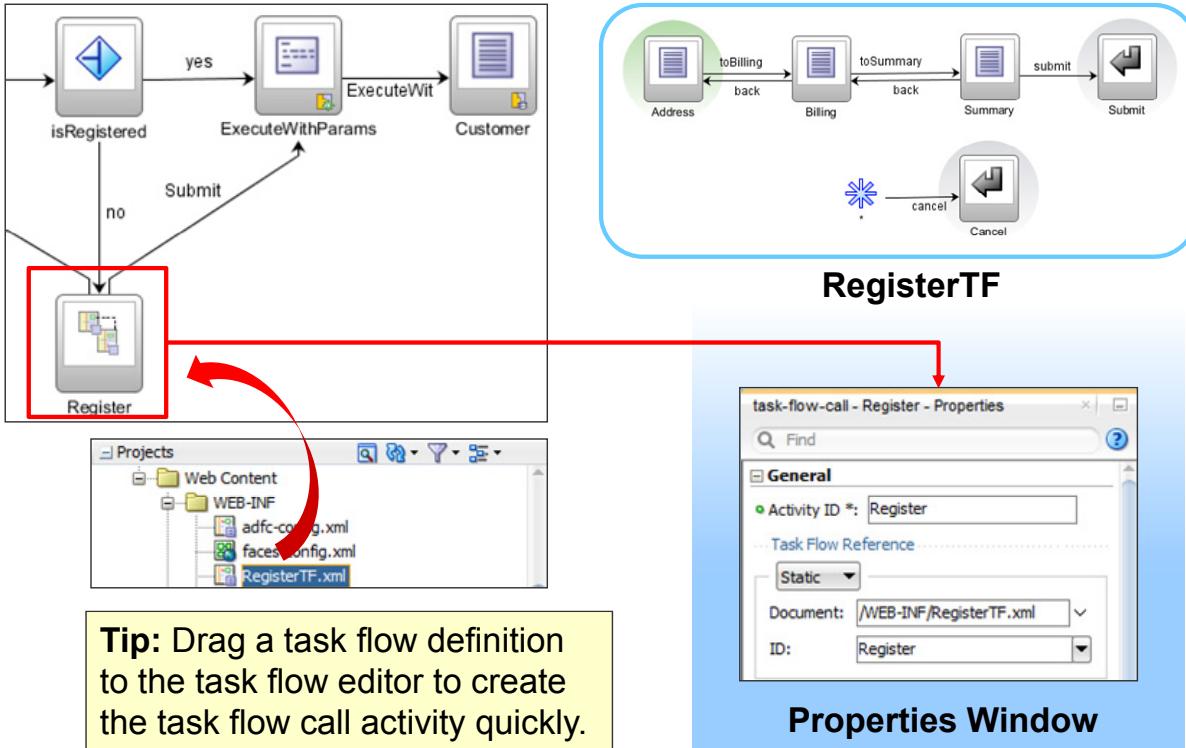
The example in the slide shows a method call activity that uses an ExecuteWithParams built-in operation. The method call activity is used to initialize the task flow by setting some view object bind variable values based on task flow parameters passed in from the caller.

To configure a method call activity, you specify values for the following properties in the Properties window:

- **Method:** Enter an EL expression for a method. The expression can call a method in a managed bean, or a built-in operation, as shown in the example:
#{bindings.ExecuteWithParams.execute}
- **Fixed Outcome:** Select the outcome that is returned by the method on successful completion (for example, ExecuteWithParams).

- **Parameters (not shown in slide):** If the method accepts parameters, specify an EL expression indicating where the value for the parameter will be retrieved (for example, `#{LoginBean.id}`). If you use a built-in operation, you do not specify parameter settings here. Instead, you set the parameters when you drag the operation to the task flow editor, and the parameters are stored in the `ExecuteWithParams` binding.
- **Return value (not shown in slide):** If the method returns a value, enter an EL expression indicating where to store the method return value (for example, `#{pageFlowScope.return}`). Appropriate boxing is performed to handle Java primitive types.

Calling Other Task Flows from a Task Flow



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

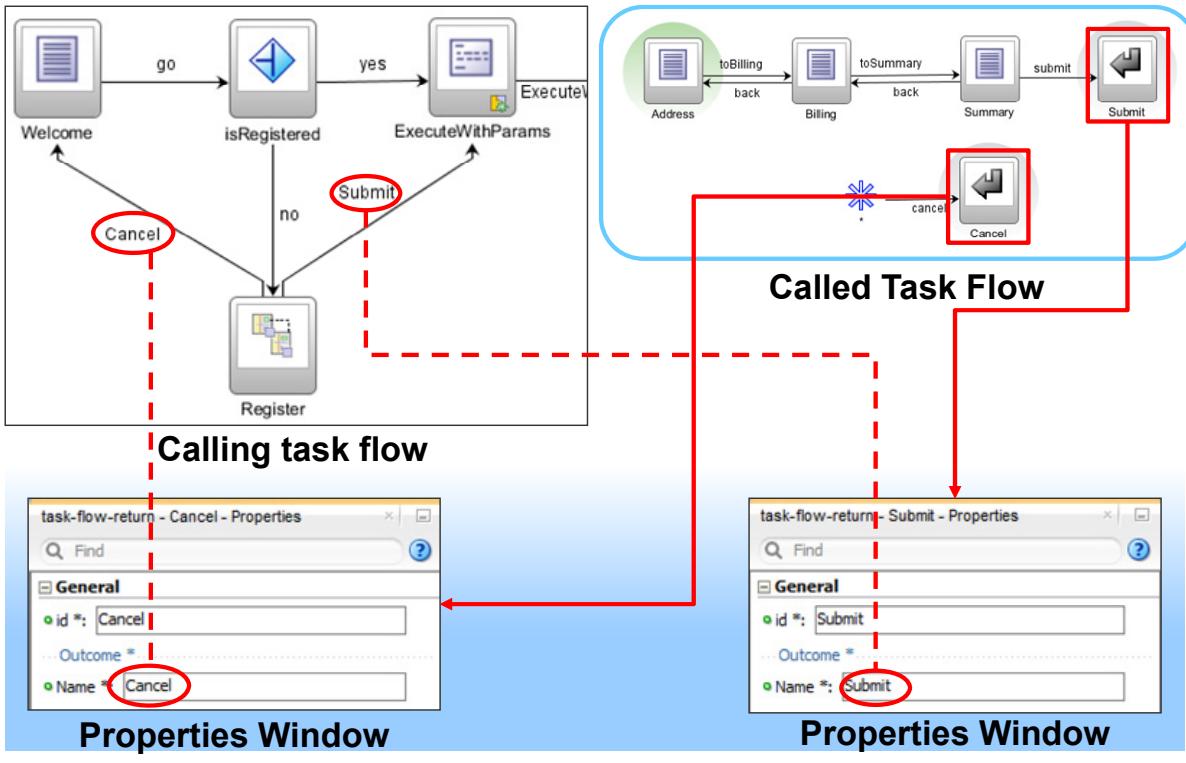
You can use a task flow call activity to call a bounded task flow from either an unbounded task flow or a bounded task flow. The bounded task flow can be located within the same application or a different application. The called bounded task flow executes its default activity first. There is no limit to the number of bounded task flows that can be called. For example, a called bounded task flow can call another bounded task flow, which can call another, and so on. You can pass parameters to the called activities, and control returns to the calling task flow when the called activity is completed.

The example in the slide shows part of a task flow that contains a task flow call activity. The task flow call activity calls a bounded task flow called RegisterTF, which takes the user through the steps required to register as a customer.

You create a task flow call activity either by dragging a Task Flow Call Activity component to the task flow editor and specifying the location of the bounded task flow definition file, or by dragging the task flow definition file from the Projects panel to the task flow editor.

In the Properties window, under Task Flow Reference, you specify the relative path to the XML file that defines the bounded task (this is set for you if you drag the task flow definition to the editor). In the Parameters section (not shown), you can specify input parameters to pass to the task flow and return values returned from the task flow. Passing parameters is covered in more detail in the next lesson.

Returning from a Task Flow



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you define a task flow return activity, you need to specify the outcome that is returned to the calling task flow. You can have only one outcome per task flow return. You can have as many task flow returns as you require, including a task flow return for a wildcard control flow rule.

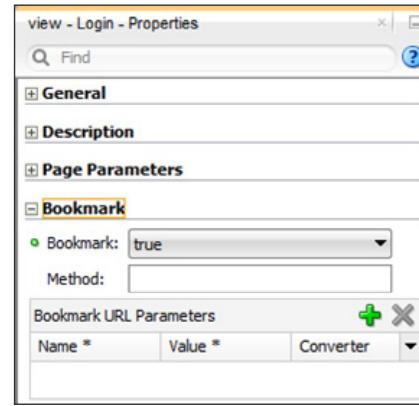
The calling flow must have a control flow rule that corresponds to the outcome that you name in the task flow return activity. If a bounded task flow has its own transaction scope, the return activity also controls whether a commit or rollback happens when the task flow returns to its caller. (You learn how to specify transaction options later in the course.)

The example in the slide shows a task flow that calls the RegisterTF task flow. The called task flow has two return activities, Submit and Cancel. When the called task flow is exited through the Submit activity, navigation proceeds to the ExecuteWithParams method call activity. When the called task flow is exited through the Cancel activity, navigation returns to the Welcome page.

Making View Activities in Unbounded Task Flows Bookmarkable (or Redirecting)

Saving a view activity as a bookmark is available only in unbounded task flows. You can:

- Designate bookmarks in the Properties window at design time
- (Optional) Specify:
 - URL parameters
 - Method to invoke before view is rendered
- Designate bookmarks at run time with the `ViewBookmarkable()` method
- Use the redirect option for a view activity instead of making it bookmarkable



ORACLE

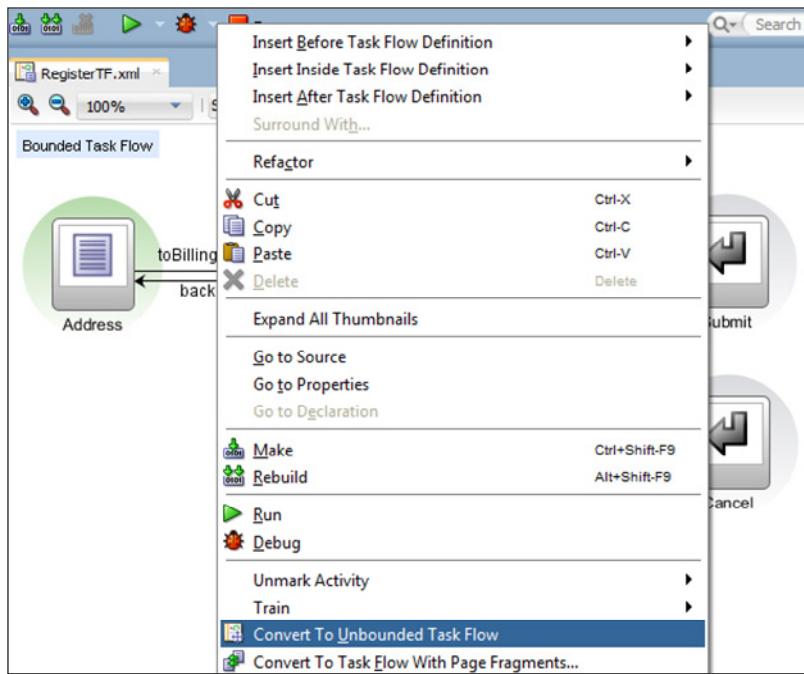
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The URL that is displayed in the browser for a JSF page might not include all the information that is required to render the page. For example, the bookmarked URL might not contain information (such as a customer ID) that enables dynamic content on the page to be reproduced. Therefore, you need to configure special handling to make view activities bookmarkable.

To configure a view activity in an unbounded task flow to be bookmarkable, select the view activity in the task flow diagram. Then, in the Properties window, set the `Bookmark` property to `true`. After you designate a view activity as bookmarkable, you can specify one or more URL parameters that enable the ADF controller to construct the bookmark URL with any request parameters that are required to rebuild the page. For the value, you specify an EL expression that, when evaluated, specifies a bookmark URL parameter value (for example, `# {pageFlowScope.customerID}`). If the object represented by the parameter is not a string, you must specify a converter method to translate the object into a `String` representation and the incoming request parameter value back to the original object. For example, an object of type `Date` adds the `String` representation of the date to the request parameter, which is converted back into a `Date` object when dereferencing the bookmark.

You can also specify an optional method that is invoked after updating the application model with submitted URL parameter values and before rendering the view activity. You can use this method to retrieve additional information based on URL parameter key values.

Converting Task Flows Between Bounded and Unbounded



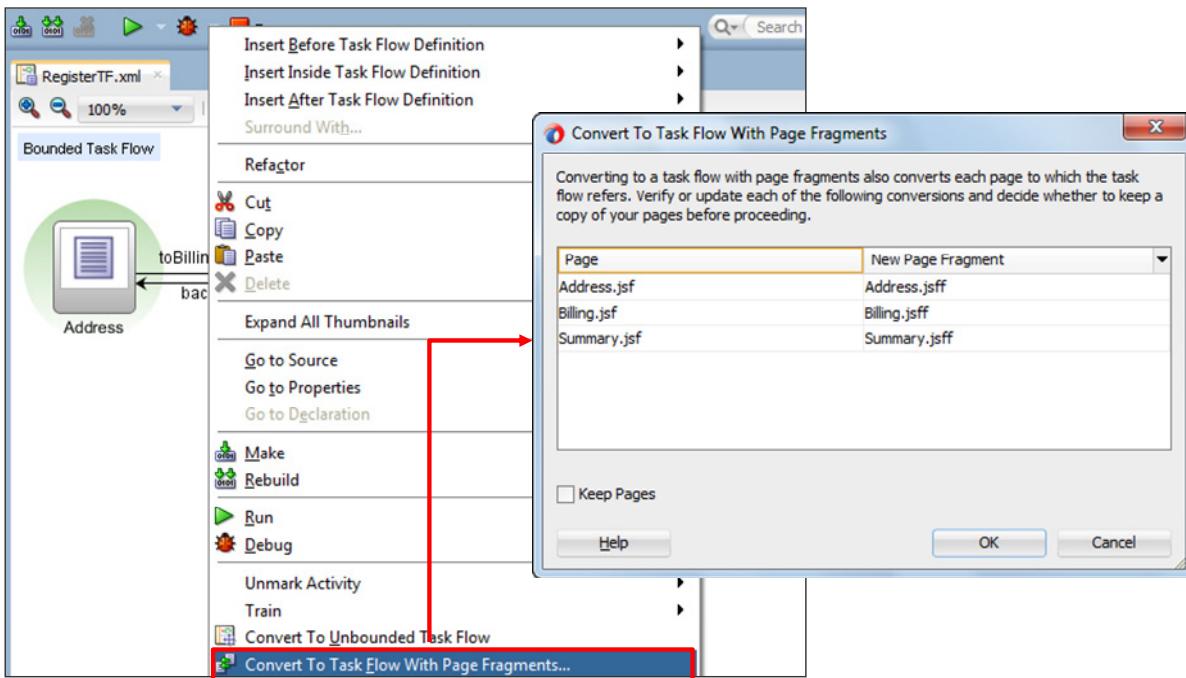
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

JDeveloper makes it easy to convert bounded task flows to unbounded ones. You right-click in the task flow diagram and select Convert To Unbounded Task Flow. For example, in the beginning, you might find it easier to work with unbounded task flows because you can access the pages in any order. Later, you can convert to a bounded task flow and define a strict navigation path.

When you convert a bounded task flow that uses fragments, the fragments are converted to pages.

Converting a Bounded Task Flow to Use Page Fragments



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As a reusability feature of Oracle ADF, page fragments enable you to break a JSF page into several smaller fragments. The smaller fragments are easier to maintain, and they can be reused on multiple pages.

Page fragments are typically used in bounded task flows. You can add a bounded task flow with page fragments to a page as a region. This approach enables you to reuse task flows and create pages that include multiple areas, with each area containing a specific flow.

To convert a bounded task flow to use page fragments instead of pages, right-click within the bounded task flow diagram and select Convert To Task Flow With Page Fragments.

You learn more about page fragments in the lesson about building reusable pages.

Using Bounded Task Flows

You can use a bounded task flow:

- In a larger application flow by using a task flow call activity
- As a region on a page
- Within a modal dialog box that is launched for a page



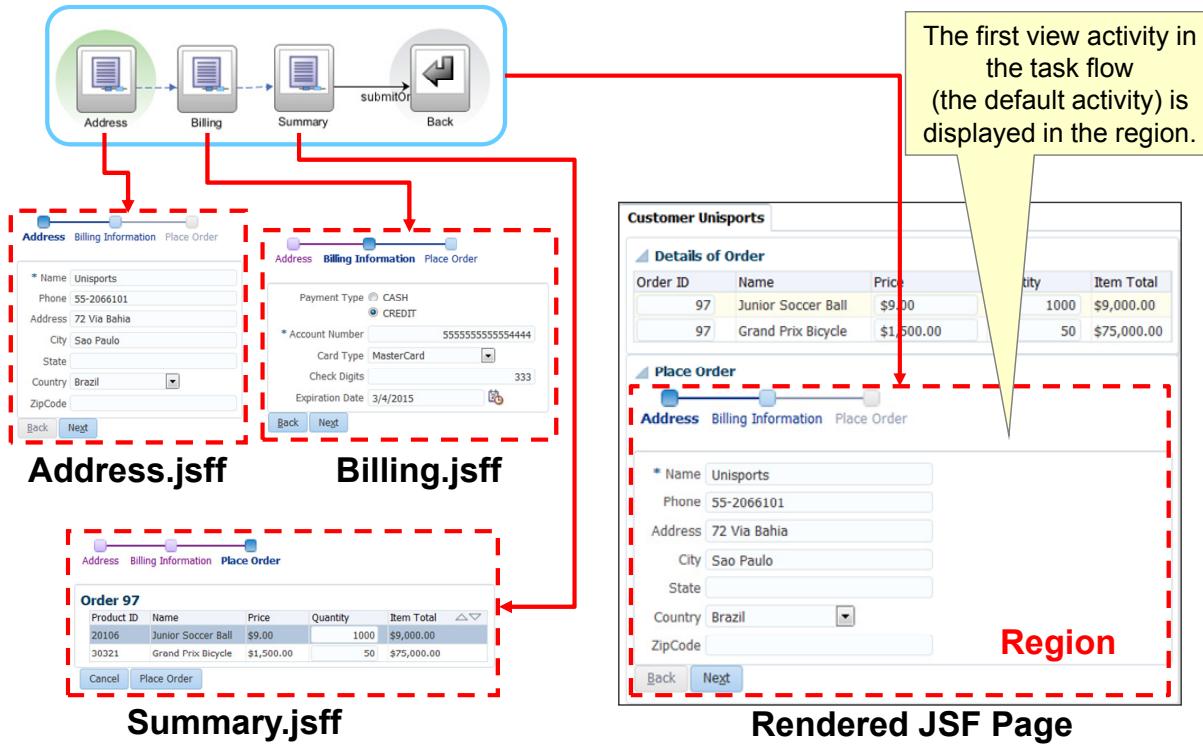
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use a bounded task flow in an application in various ways:

- As a set of pages and other activities in a larger application flow
- As a region, providing navigation between page fragments on a single containing page
- In a modal dialog box that is launched for a page

You have already learned how to call a bounded task flow from a larger application flow by using a task flow call activity. In the following slides, you learn how to include bounded task flows on regions within pages.

Using Regions on a Page



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

A region is a powerful feature in Oracle ADF that can be used within a page or page fragment. A region is a JSF component that represents a task flow as a smaller part of a larger page. You can use a region to wrap a task flow for display on a page that has other content.

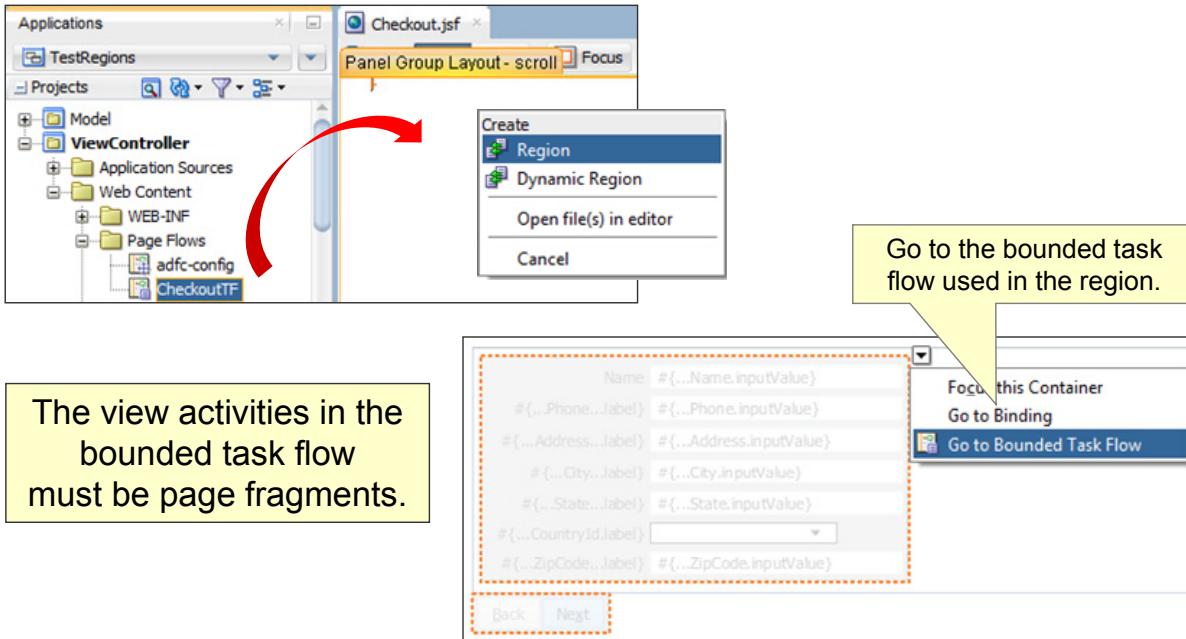
A region is similar to a portlet in functionality and user behavior, but regions are used when the functionality is in a local context, whereas portlets are used when exposing the functionality for external usage. Because a region runs locally in the local context, it can share information and transaction boundaries with the rest of the items on the page, and it does not have the overhead that a remote portlet does. Any task flow can also be wrapped as a portlet for external usage but should be used as a region for internal use cases.

Building well-defined, parameterized task flows that can be used as regions on pages is a good way to provide reusable and highly maintainable segments of functionality across your application.

The example in the slide shows a special type of bounded task flow called a *train*. You learn about creating trains later in this lesson. Of course, regions are not restricted to trains; you can use a region to wrap any bounded task flow that is composed of page fragments.

Defining a Bounded Task Flow as a Region

Drag a bounded task flow to the page as a region:



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can drag a bounded task flow that includes page fragments to a page and create the task flow as a region on that page. You can choose to create a simple region (choose Region) or a region that you can switch dynamically at run time (choose Dynamic Region). When you create a dynamic region, you also need to specify logic in a managed bean that sets the name of the dynamic region that is displayed.

The default activity of the task flow appears in the editor and on the page when it is first run. You can edit the bounded task flow by selecting the menu at the top-right corner of the region and choosing "Go to Bounded Task Flow."

At run time, you can use page navigation (described later in this lesson) to navigate through the task flow. As you do this, the region refreshes without redrawing the entire page (this is called *partial page refresh*).

Creating Navigation Components

ADF Faces components used for navigation:

- Buttons 
- Links [More...](#)
- Breadcrumbs Soccer Equipment > Balls
- Trains 
- Menus 
- Toolbars 
- Navigation panes 



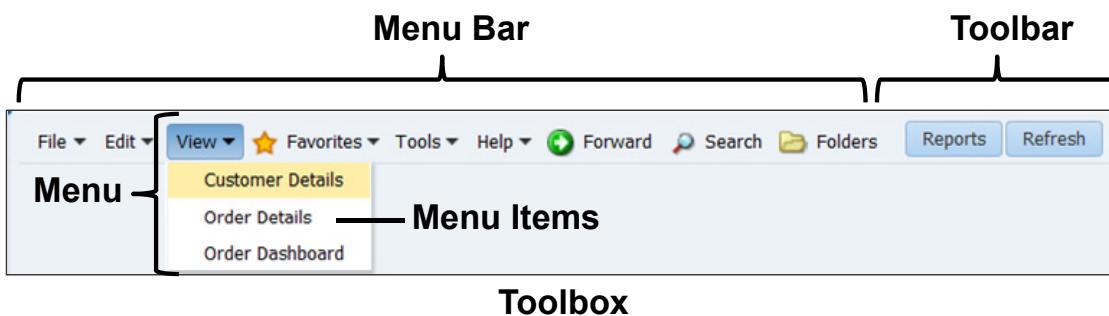
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Earlier you learned about some of the ADF Faces components that you can use for navigation, such as buttons, links, breadcrumbs, trains, menus, and toolbars. You now learn how to bind components to actions so that the components can be used for navigation.

Note: This lesson does not cover simple navigation components (such as buttons and links) because you already learned how to specify navigation for those types of components when you learned about task flows previously in the course.

Navigation Menus and Toolbars

- You can group menu bars and toolbars in a toolbox.
- Menu bars contain menus, which provide the top-level menu.
- Menu items provide the vertical (drop-down) selections.
- Menu items and toolbar buttons can perform navigation.
- Toolbars contain other components, such as buttons.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

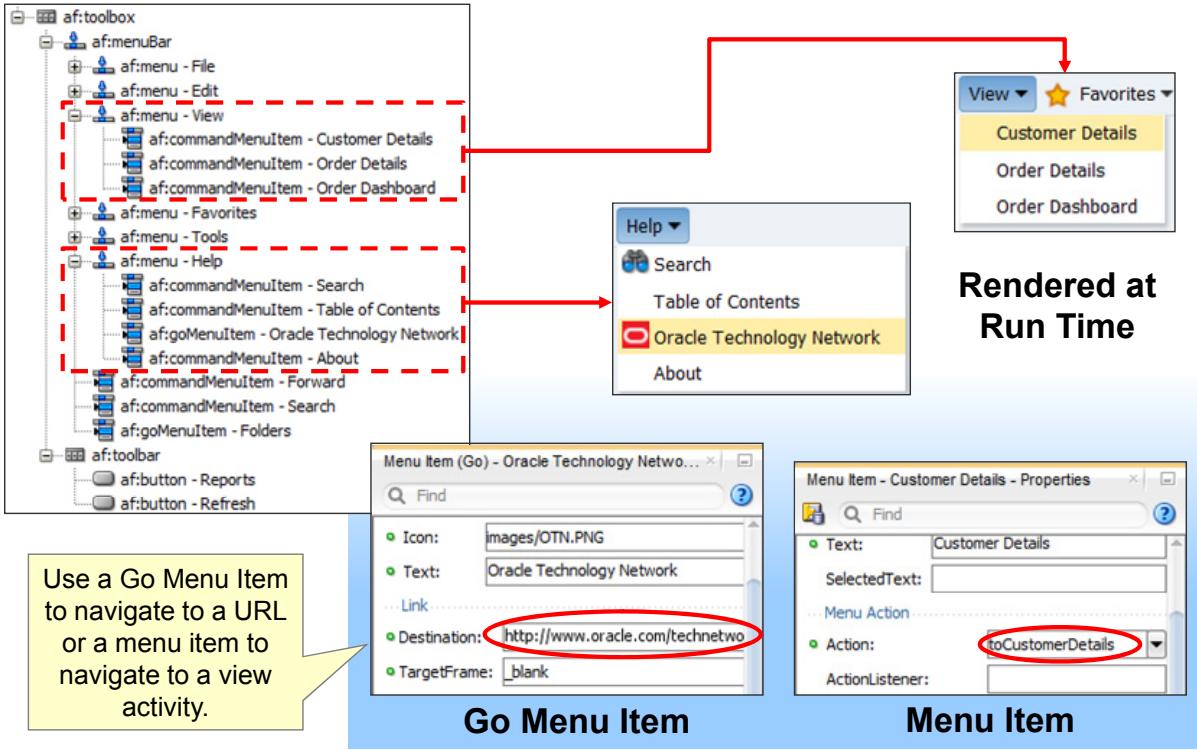
ADF Faces provides menu bars and toolbars that you can add to a page. Menu items and toolbar buttons enable users to perform changes to selected objects or to navigate somewhere else in the application. A toolbox can be used to group menus and multiple toolbars together. The example in the slide contains both a menu bar and a toolbar.

Both menu items and toolbar buttons have the capability to show selection. They can navigate to other pages in the application and perform any logic needed for navigation in the same way that other command components do.

You use the Menu Bar component to render a bar that contains the menu bar items (such as File, Edit, View, and so forth in the example). Each item on a menu bar is rendered by a menu component, which holds a vertical menu. Each vertical menu consists of a list of menu items that can invoke an operation on the application. You can nest a menu component inside another menu component to create a cascading menu.

You can include both menu bars and toolbars in a toolbox to display components that otherwise are not supported in menu bars, such as drop-down lists, buttons, and links.

Defining Navigation Menus



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

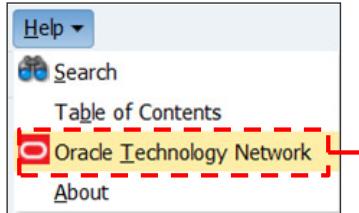
You create menus in a menu bar by nesting menus and menu items.

To create a menu bar:

1. Drag a Menu Bar component from the Components window to the JSF page.
2. Drag the desired number of Menu components to the menu bar.
3. Drag a menu item from the Components window. You can drag the menu item to a menu or directly to the Panel Menu Bar, which renders like a button on the menu bar.
4. Within each menu component, drag Menu Item components from the Components window to insert a series of Menu Item components that define the items in the vertical menu.
5. Set the properties of the menu components in the Properties window.
 - **Text:** The label for the button. Storing text in a resource bundle is recommended.
 - **Accelerator:** The keystroke to activate this menu item's command when the item is selected. ADF Faces converts the keystroke and displays a text version of the keystroke (for example, Ctrl + O) next to the menu item label.
 - **Icon:** Enter the URI of the image file you want to display.

- **Type:** Specify a type for this menu item. When a menu item type is specified, ADF Faces adds a visual indicator (such as a check mark) and a toggle behavior to the menu item. At run time, when the user selects a menu item with a specified type (other than default), ADF Faces toggles the visual indicator or menu item label.
- **Selected:** Set to `true` to select this menu item. By default, a menu item is not selected. The selected attribute is supported for check, radio, and antonym type menu items only.
- **SelectedText:** Set the alternate label to display for this menu item when the menu item is selected. The type attribute for the menu item must be set to antonym.
- **Action:** Use an EL expression that evaluates to an action method in an object (such as a managed bean) to be invoked when this menu item is activated by the user. The expression must evaluate to a public method that takes no parameters and returns a `java.lang.Object`. If you want to cause navigation in response to the action generated by the Menu Item component, instead of entering an EL expression, enter a static action outcome value as the value for the action attribute. You then need to either set the PartialSubmit property to `false` or use a redirect.
- **ActionListener:** Specify the expression that refers to an action listener method that is notified when this menu item is activated by the user. This method can be used instead of a method bound to the action attribute, allowing the action attribute to handle navigation only. The expression must evaluate to a public method that takes an `ActionEvent` parameter, with a return type of `void`.

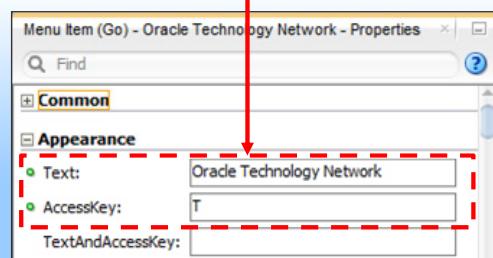
Defining Access (Shortcut) Keys



Access key is set to T.

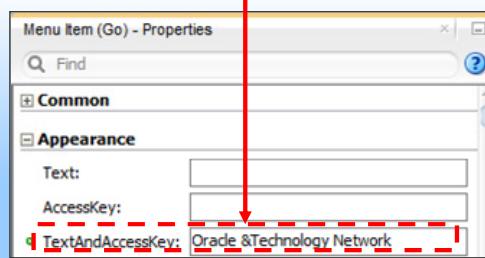
Tip:

Use LabelAndAccessKey and ValueAndAccessKey to define access keys for input components.



Specify Text and an AccessKey.

OR



Specify TextAndAccessKey.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can define access keys (also known as *mnemonics* or *shortcut keys*) for a component by entering the mnemonic character in the AccessKey property in the Properties window. The example in the slide shows two different ways to set the access key for the Oracle Technology Network menu item to the letter *T*.

Use one of the following attributes to specify a keyboard character for an ADF Faces input, command, or go component:

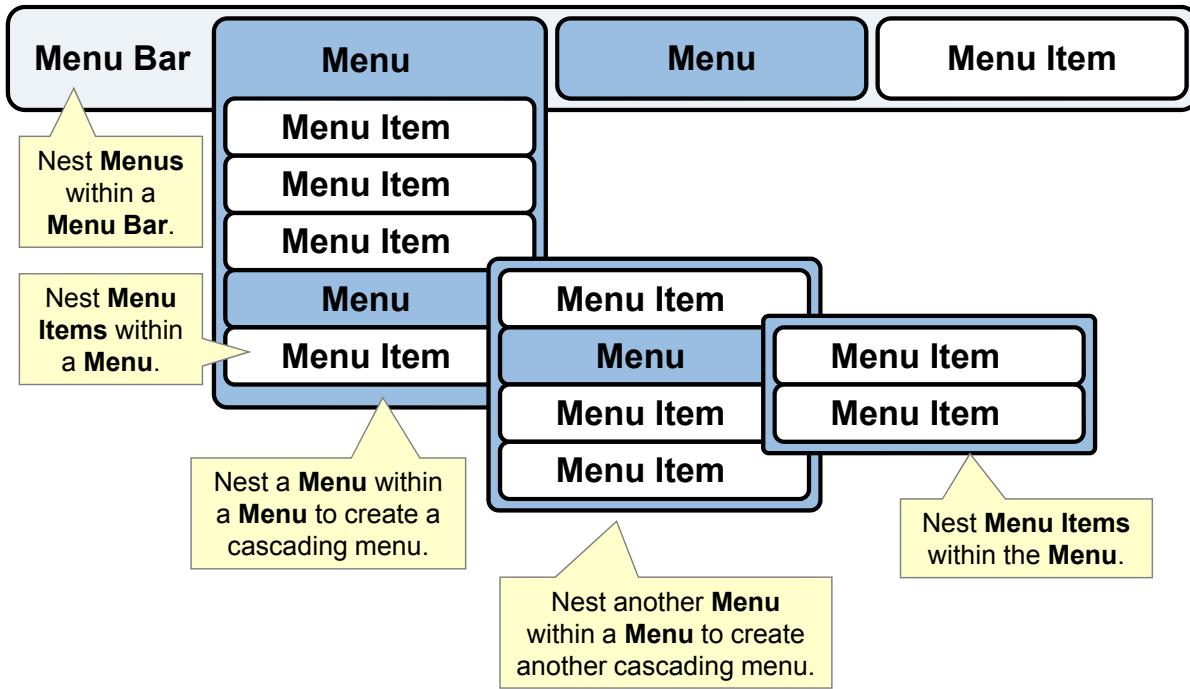
- **AccessKey:** Sets the mnemonic character used to gain quick access to the component. For buttons and links, the character specified by this attribute must exist in the text attribute of the component. Otherwise, ADF Faces does not display the visual indication that the component has an access key.
- **TextAndAccessKey:** Sets both the text and the mnemonic character for a component by using the ampersand (&) character. In the source editor, use & for an ampersand. In the Properties window, use only &. The ampersand should precede the character that is used as an access key.
- **LabelAndAccessKey:** Sets both the label attribute and the access key on an input component by using conventional ampersand notation
- **ValueAndAccessKey:** Sets both the value attribute and the access key by using conventional ampersand notation

Reusing Access Keys

You can bind the same access key to several components. If the same access key appears in multiple locations on the same page, the rendering agent cycles among the components accessed by the same key. That is, each time the access key is pressed, focus moves from component to component. When the last component is reached, focus returns to the first component.

Depending on the browser, if the same access key is assigned to two or more Go components on a page, the browser may activate the first component instead of cycling through the components that are accessed by the same key.

Defining Cascading Menus



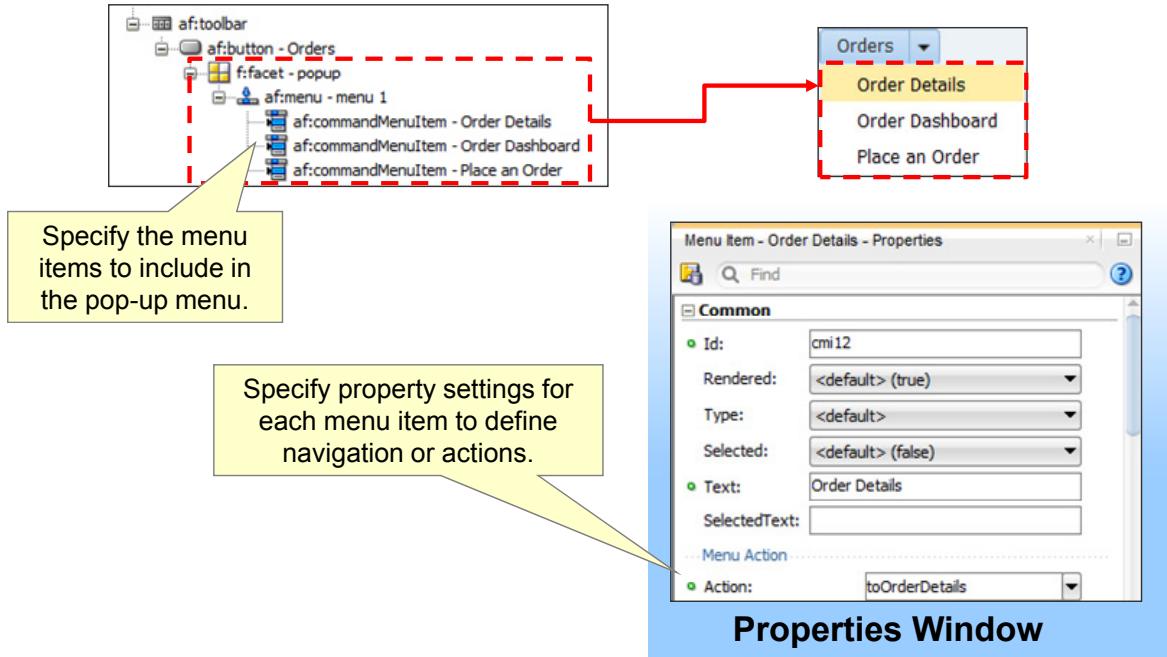
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Menu components are useful for creating groups of common commands. You can create cascading menus by nesting menu components, along with menu items, under other menu components. You can create multiple levels of cascading menus, but keep in mind that more than two levels of nesting can be confusing to end users. Notice that you can also add menu items directly to the menu bar

Creating Pop-Up Menus

Add a menu to the pop-up facet of a button:



ORACLE

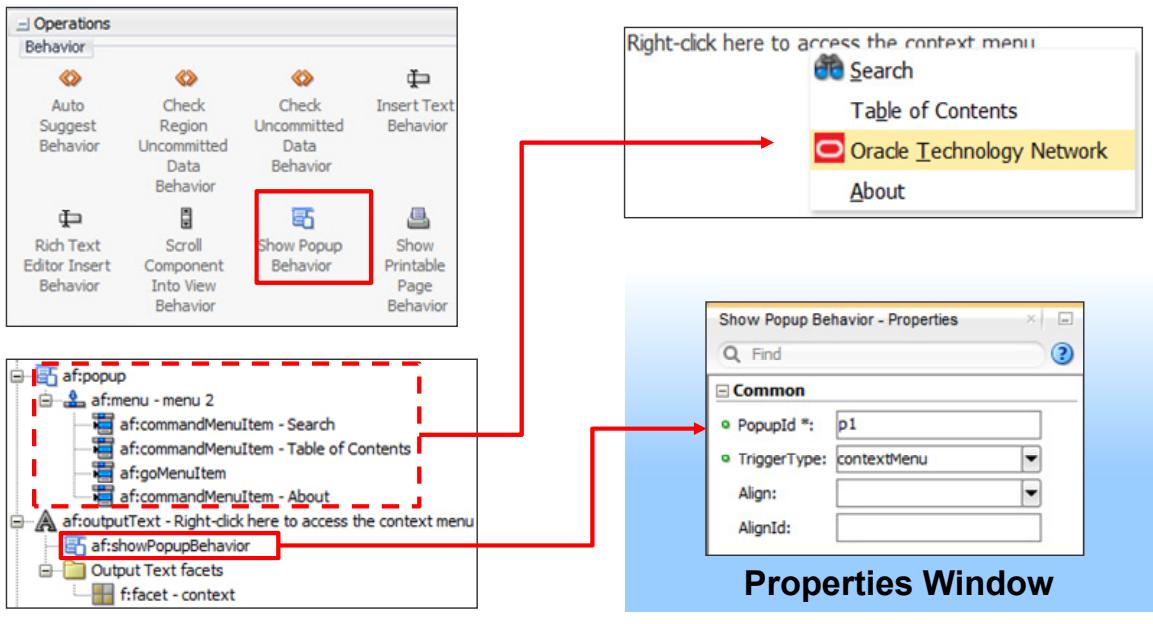
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can also have the menu launch in a pop-up window that displays vertically. To create a pop-up menu, perform the following steps:

1. Drag a button component to a toolbar on the JSF page and set the Text property to specify the text that appears on the button. Use the Icon property to set the image that appears on the button.
2. In the Structure window, expand the af:button node and drag a Menu component to the popup facet.
3. Drag to the Menu component as many Menu Item components as needed to define the items in the vertical menu. Set properties on the Menu Items as needed.

Creating Context Menus

To create a context menu, use the Show Popup Behavior operation from the Components window:



ORACLE

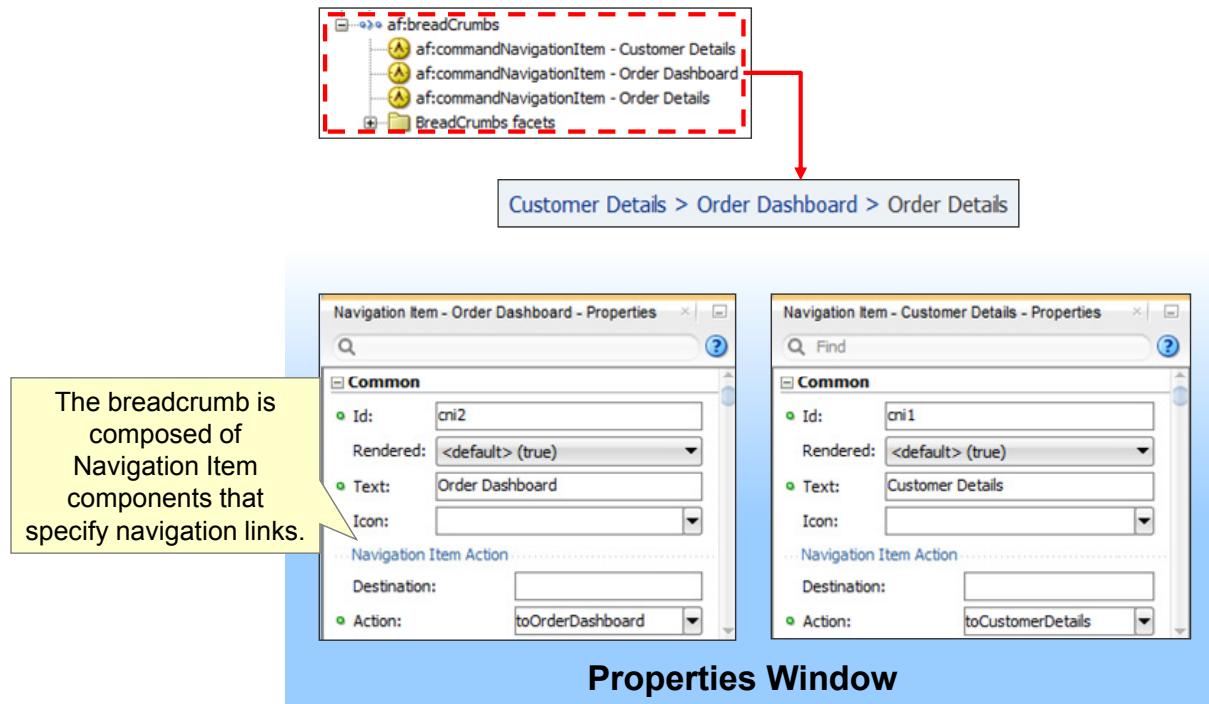
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ADF Faces client behavior tags provide declarative solutions to common client operations that you would otherwise have to write yourself using JavaScript, and register on components as client listeners. ADF Faces provides a number of client behavior tags to simplify the development of rich client interfaces. The Show Popup Behavior operation is an example of a client behavior tag that you can use to display context menus in your application.

To create a context menu for a component:

1. Create a pop-up menu that consists of a Popup component containing a nested Menu component and one or more nested Menu Item components.
2. Give the pop-up menu an ID. In the example, the ID of the popup menu is p1.
3. Create a component to use the context menu (the example uses an Output Text component).
4. Drag the Show Popup Behavior operation to the component and set properties:
 - **PopupId:** Specify the ID of the Popup component that will display in the pop-up.
 - **TriggerType:** Select contextMenu. Other trigger types include various mouse events, such as mouse down and mouse hover, and double click.
 - **AlignId and Align:** Optionally specify the ID of the component to align the pop-up contents with and specify an alignment position that is relative to the component identified by AlignId.

Creating Breadcrumbs

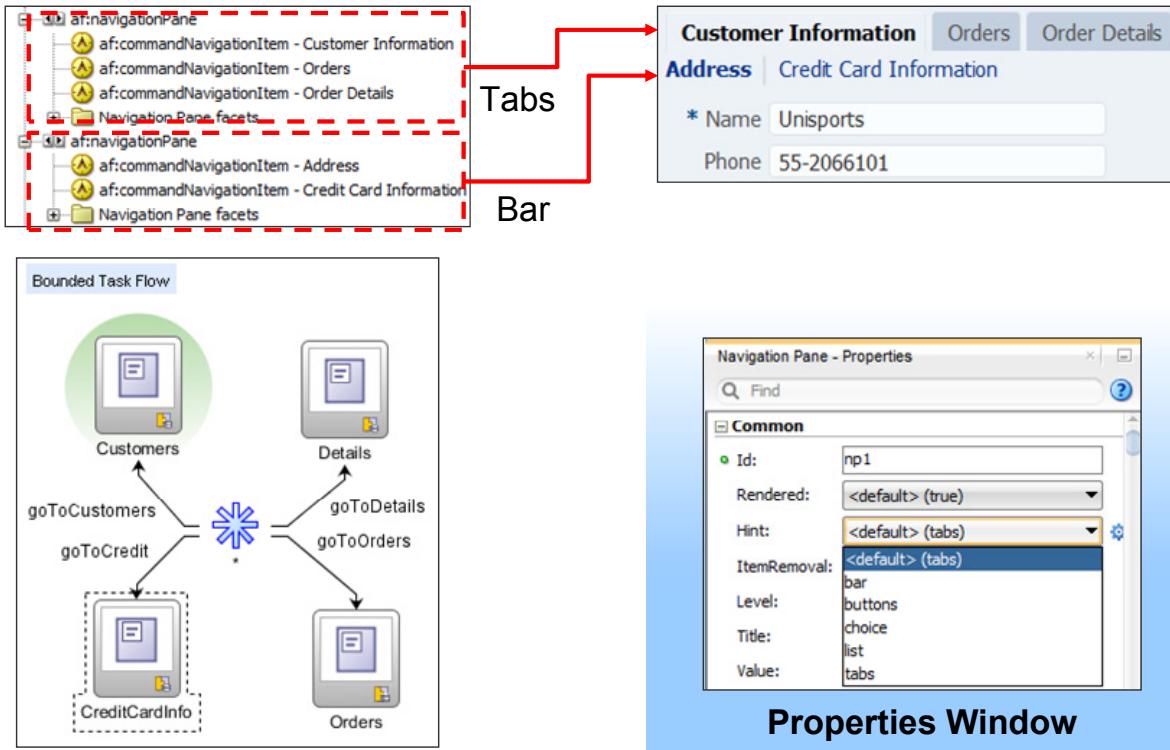


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

You use the BreadCrumb component to display a path of links for navigation. You can explicitly define the links by adding Navigation Item component for each link in the breadcrumb trail (as shown in the example). You need to add the breadcrumbs to each page in the path, or use a page fragment or page template to include the breadcrumbs on every page. You learn about using page fragments and page templates in the lesson about building reusability into pages.

Creating a Navigation Pane



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

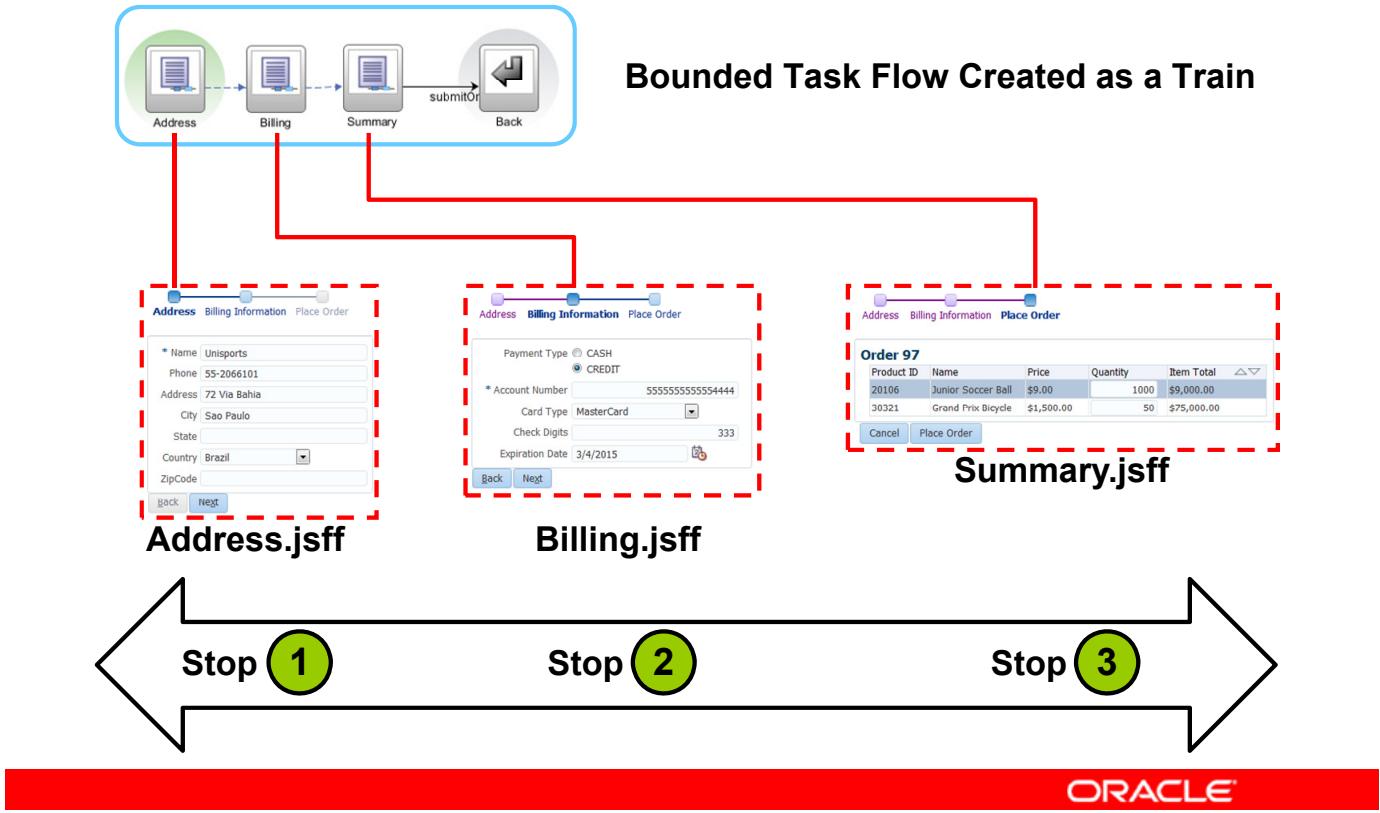
The Navigation Pane component is useful for creating multiple levels of navigation. For example, in the slide, there are two levels of navigation. The top level uses tabs (Customer Information, Orders, and Order Details) to navigate between page fragments within a region on a page. The second level of navigation use a bar on the Customer Information page to toggle between showing address information and showing credit card information.

To create a navigation pane, you define one global navigation rule that can access each page in the hierarchy. Then you drag a Navigation Pane component to a page and specify a hint that determines the kind of navigation control to use (bar, buttons, choice, and so forth). You then add Navigation Item components to the navigation pane. For each navigation item, you set the Action to a String outcome corresponding to one of the outcomes in the global navigation rule. You need to create the navigation pane on every page involved in navigation, making sure that you include only the navigation items that are relevant for that particular page where it exists in the navigation hierarchy. So, in the example, the Customer Information page is the only page that includes a navigation bar that toggles between the address and credit card information. You can reuse navigation panes on pages by using a page fragment or a page template.

Depending on the Hint attribute of the navigation pane, the navigation items can be rendered as:

- **Bar:** Displays the navigation links separated by a bar
- **Buttons**
- **Choice:** Displays navigation items in a pop-up list when the associated icon is clicked. (You must include a value for the navigation pane's icon attribute.)
- **List:** Bulleted list
- **Tabs**

Using Trains



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

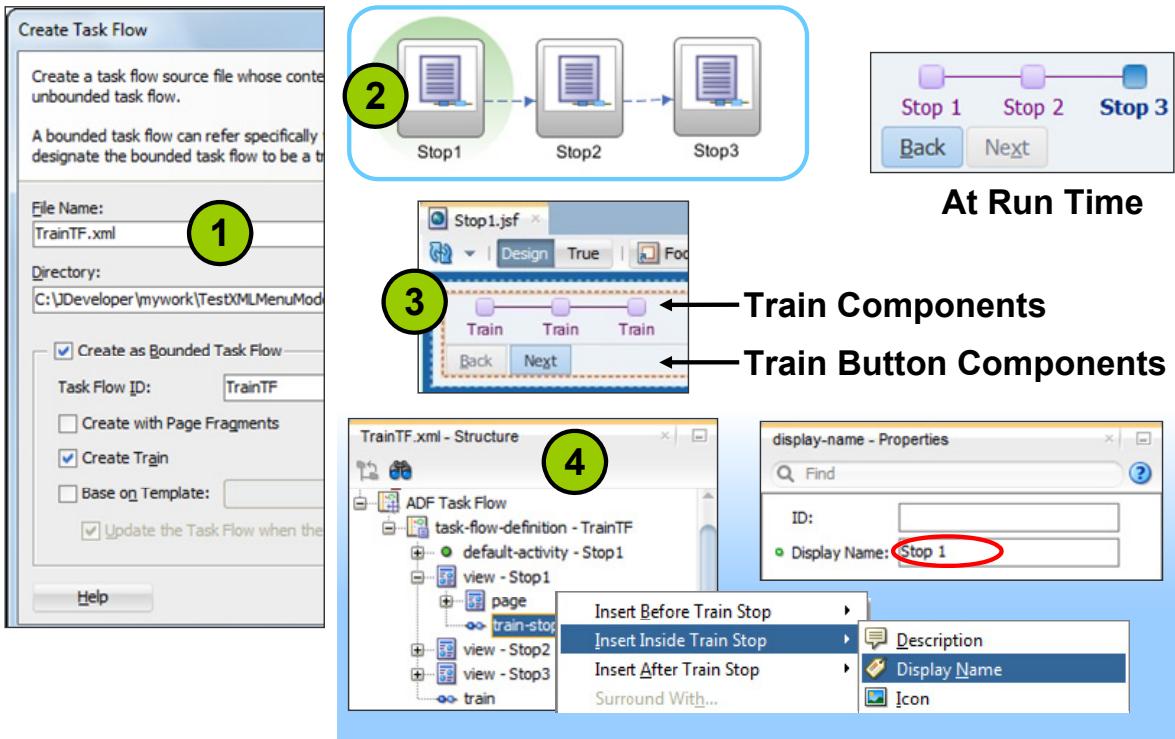
A train is a specialized type of bounded task flow that enables you to lead users through a series of pages that you want them to visit in a particular order. A train consists of a series of train stops, with each train stop corresponding to one step (or one page) in a multistep process. Users navigate the train stops by clicking an image or label, which causes a new page to display. Typically, train stops must be visited in sequence. Train stops can also be configured so that end users do not have to visit all the stops in sequence.

Trains offer some conveniences over coding your own bounded task flow, including:

- Several different styles of train stops to choose from.
- Navigation controls that include visual indicators (bold fonts, colored fonts, images, and so forth) to show the user's progress through the steps, including an indication of all visited stops, the current stop, the next stop (enabled), and grayed-out stops.

Earlier, you saw an example of a train that was used to lead users through the checkout process. The train task flow is composed of three page fragments: Address.jsff, Billing.jsff, and Summary.jsff. The user can either click the train navigation components at the top of each page, or use the Next and Back buttons to navigate through the sequence. Notice that the Place Order icon is grayed out in Stop 1. This prevents the user from jumping ahead and placing the order before visiting the billing page. Also, in Stop 2, notice in the navigation control that the visited stops appear in purple and that the Place Order stop is now active.

Creating Trains



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You create trains by defining a bounded task flow that uses the train model and by adding Train components to the pages in the task flow.

To create a train, perform the following steps:

1. Create a bounded task flow, being sure to select the Create Train check box.
2. Drag View components to the task flow—each View component becomes a train stop. You can add other components to the task flow as needed, but page flow between train stops is defined automatically.
3. Double-click each view to create the page (or page fragment), and then drag a Train component to the page. By default, JDeveloper binds the Train component to the task flow's train model.
 - Drag a Train Button Bar component to the page, which also, by default, binds to the task flow's train model.
 - Add whatever content you want to appear on the page.
 - Copy the Train components to all other view activities in the train task flow.

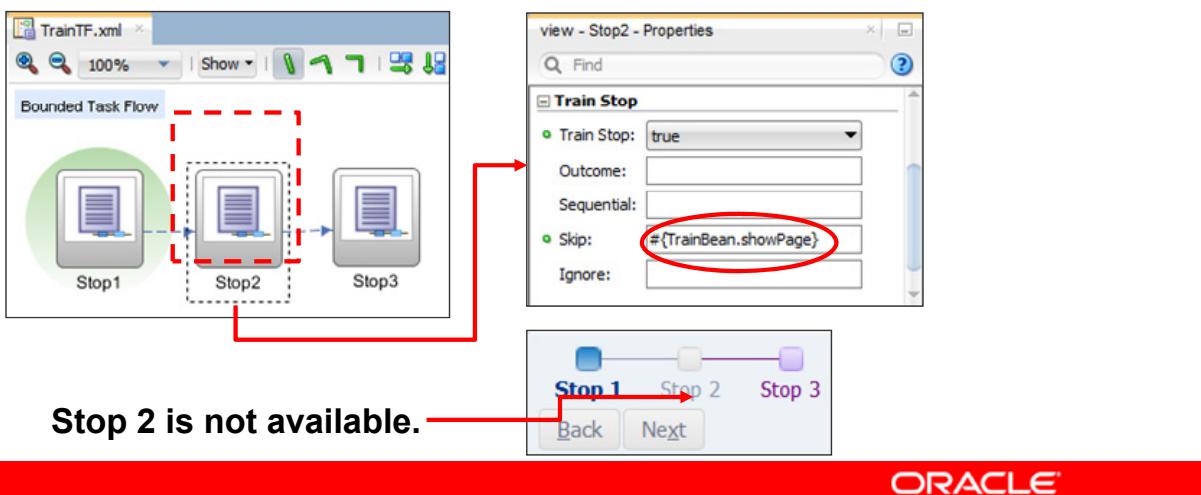
4. **Optional:** If you want to display a label for the navigation items in the train, add a display name to each train stop. To add a display name, right-click each train stop, insert a Display Name inside of the train stop, and specify the value that you want to display. Otherwise, there will be no labels on the navigation items.

Notice that the navigation items are created automatically for you when you use the train model. You do not need to define the navigation separately.

Skipping a Train Stop

Controlled by the Skip property of the train view activity:

- If true, users cannot navigate to the train stop.
- The property is typically set to an expression that evaluates to true or false, such as a managed bean method or property.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can set a train stop to be skipped by setting the Skip property of a train view activity to true. You would typically set this property to an EL expression that evaluates to true or false to conditionally skip the train stop.

The example in the slide shows the Properties window for a view in the task flow of the train. The skip property is set to a managed bean method that is coded to perform some evaluation and conditionally return either true or false.

At run time, the user cannot navigate to a skipped train stop.

Summary

In this lesson, you should have learned how to:

- Describe the difference between bounded and unbounded task flows
- Create routers for conditional navigation
- Call methods and other task flows
- Create menu items, menu bars, pop-up menus, context menus, and navigation panes
- Define breadcrumbs and trains
- Create a page fragment and use it in a bounded task flow
- Use a bounded task flow as a region



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Practice 10 Overview: Adding Advanced Features to Task Flows

This practice covers the following topics:

- Creating a bounded task flow
- Calling a task flow from another task flow
- Creating a train
- Converting bounded task flows to use page fragments
- Using a bounded task flow in a region on a page



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.